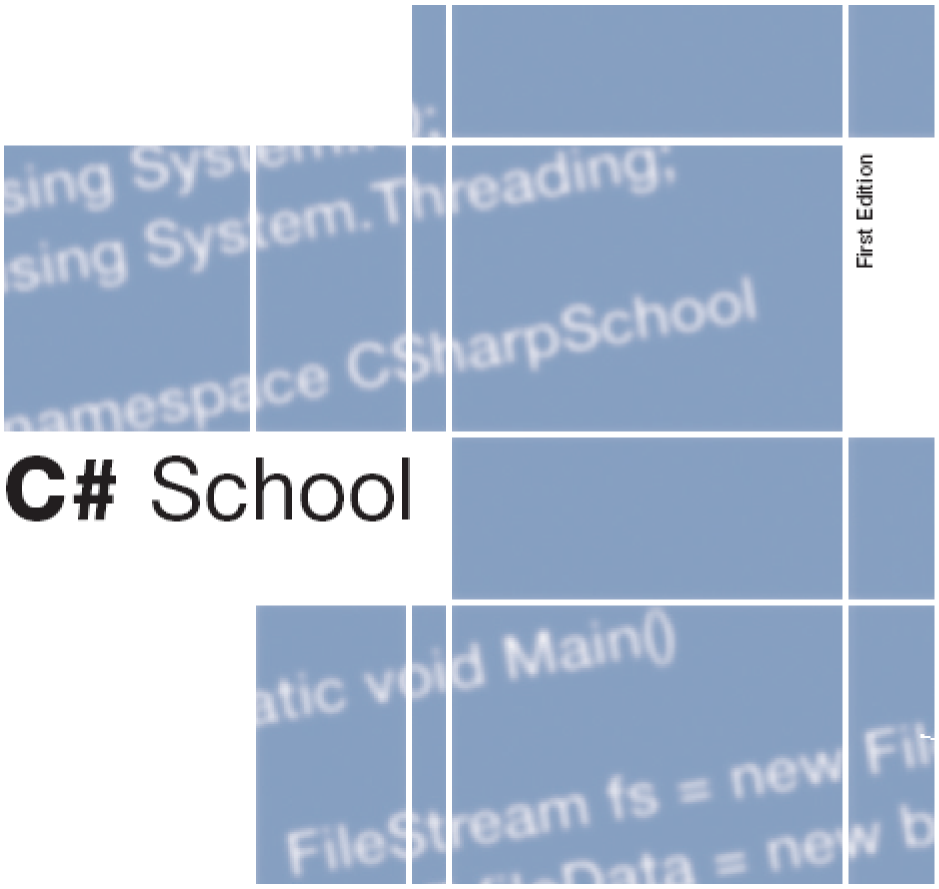


First Edition

C# School

14 lessons to get you started with C# and .NET

Author: Faraz Rasheed

Editors: Tore Nestenius
Jonathan Worthington
Lee Addy

www.programmersheaven.com

Programmer's Heaven

C# School

First Edition

Faraz Rasheed

Edited By:

Tore Nestenius

Jonathan Worthington

Lee Addy Wright

Synchron Data S.L
Av. Clemente Diaz Ruiz
Urb. Puebla Lucia local 7, 12-20
296 40 Fuengirola, Spain

To obtain the latest updates to this book, to suggest corrections or to make other comments, visit:
http://www.programmersheaven.com/2/CSharpBook

Foreword

from Jonathan Worthington

I approached the .NET platform and the C# language with skepticism in the beginning. A number of things have turned me to view both of them as generally good technologies; editing the Programmer's Heaven C# School, the lessons of which are collected together in this e-book, was one of them.

In many ways the .Net platform has asked "what do developers waste time doing" and tried to improve developer performance. For example, the .NET virtual machine provides memory management, a task that takes up much developer time when it has to be done manually. A large and well-documented class library helps avoid re-inventing the same wheel many times over. Inter-operability between code in a number of languages is made trivial.

The C# language was created alongside the .NET platform. It could be considered the "native" language of .NET, providing access to the vast majority of language features that the .NET runtime is optimized to support. It takes the best bits of Java, C and C++, producing a language with the clear object oriented programming constructs of Java along with useful features such as enumerations and structures from C. The initial version of C#, as taught in the original C# School, is mostly focused on the object oriented programming paradigm. C# 2.0 has added support for parametric polymorphism (known as generics) as well as a range of other features, and I have written an additional chapter for this book to cover some of these. The future C# 3.0 is even more adventurous, bringing in ideas from both declarative and functional programming.

I hope that this e-book helps you get to grips with the C# programming language and the .Net platform and proves a useful reference for the future. It is the first edition, but hopefully not the last – your feedback will help us in that sense, so please do not hesitate to send your comments and especially information about any mistakes to info@programmersheaven.com.

Have fun,

Jonathan Worthington
Programmer's Heaven C# School Editor

About Programmer’s Heaven

Started by Tore Nestenius in 1998, the Programmer’s Heaven website has grown to be one of the leading developer resource sites on the net. Taking its name from a range of developer resource CDs published by Tore in the years before the site began, it now features over 30,000 resources spanning a wide range of technologies, from assembly programming to XML

Today Programmer’s Heaven is more than just a massive resource directory. It features many message boards where hundreds of thousands of messages have been posted on a vast range of topics, with experts answering questions for those getting started. Recent years have seen a great deal of original content published by Programmer’s Heaven too. The C# School, now collected together into this e-book, was one very successful example of this. The latest developments on the site include a Usenet archive and a range of “Web Tools”, essential web-based utilities designed to assist those who are building web sites and web-based applications.

Since 2000, Tore has worked full time on Programmer’s Heaven. A range of freelance experts from around the world, including England, the USA, China, Korea and India, have also contributed to the site.

So, why not see how Programmer’s Heaven can help you with your development work today? 650.000 unique visitors a month can't be wrong!

http://www.programmersheaven.com/

.NET newsletter
The Programmer's Heaven .Net newsletter, sent out up to four times a month, contains the latest .Net news along with information on new .Net articles and resources on Programmer's Heaven. The .Net platform is evolving fast, and signing up for our newsletter is a great way to be kept in the picture. In the immediate future, expect coverage of C# 3.0 and WCF.

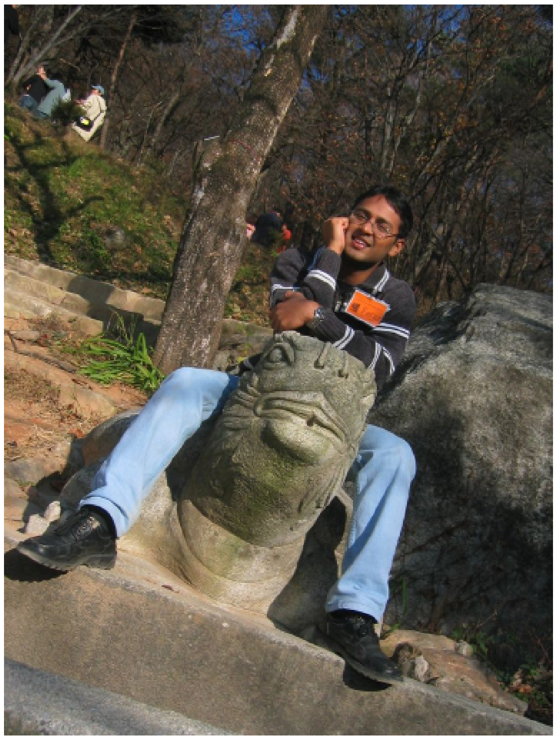
Sign up today for free at http://www.programmersheaven.com/2/DotNet-Newsletter

About Faraz Rasheed

Faraz Rasheed is currently working towards his MS (Computer Engineering) from Kyung Hee University, South Korea, and working as a research assistant in the Ubiquitous Computing Group (UCG) of Realtime & Multimedia Lab (www.oslab.khu.ac.kr). His current research area is 'Information management in context aware & ubiquitous computing environments'. He completed his BS (Computer Science) at the Department of Computer Science, University of Karachi, Pakistan.

He has a strong interest in the object oriented software development process, specifically using .Net and Java based technologies. He has done various projects using C#, VB.Net, ASP.Net on standard and compact edition of .Net, which also involve technologies like ADO.Net, web services, .Net remoting, reflection and strong named shared assemblies.

He can be contacted by email at either frazrasheed@hotmail.com or faraz@oslab.khu.ac.kr.



C# 3.0
C# 3.0 is the next version of the C# language, currently in BETA and due for release in 2007. We have created an area on the Programmer's Heaven site to provide coverage of the exciting new features in C# 3.0. This will include a series of articles explaining the new features and the concepts behind them in the same plain English that you have found in this book.

Contents In Summary

Contents In Detail

1. Introduction

The Way

The book is divided in to three progressive levels. In the first beginner stage we will be discussing the .Net Framework, C# Language Fundamentals and Object Oriented Programming.

In the second intermediate section we will go into depth with Object Oriented constructs such as inheritance, polymorphism, abstract classes, interfaces, structures, enumerations and exceptions.

In the third and final advanced section we will delve into what is required to implement real world applications using C# with Base Libraries, focusing on topics such as Collections, Delegates, Events and Windows Programming with a number of controls, as well as Data Access with ADO.Net, Threads and Streams.

Tools of the trade

Our examples have been written within the standard IDE (Visual Studio.Net). We recommend that you obtain this. Microsoft are currently offering Visual Studio.Net 2005 Express Edition for free, and this is available at http://msdn.microsoft.com/vstudio/express/. An alternative open source IDE, SharpDevelop, is available from http://www.icsharpcode.com/OpenSource/SD/. It is also possible to use any text editor (such as Notepad) to write the C# code.

You will need to download and install the .Net Framework SDK, which can be obtained freely from http://msdn.microsoft.com/netframework/downloads/howtoget.asp. This is needed to run .NET applications and, more importantly for us, contains the C# Compiler that you will need to compile programs you write.

Finally, for non-Windows users, the Mono Project supplies an open source C# compiler, .Net runtime and class library implementation. For more information, see http://www.mono-project.com/.

The code examples in this book were written and tested with the .Net Framework version 1.1 and Visual Studio 2003, but should work fine with later versions. They have not been tested with Mono.

The C# Language

C# (pronounced C-Sharp) is no doubt the language of choice in the .Net environment. It is a whole new language free of the backward compatibility curse with a whole bunch of new, exciting and promising features. It is an Object Oriented Programming language and has at its core, many similarities to Java, C++ and VB. In fact, C# combines the power and efficiency of C++, the simple and clean OO design of Java and the language simplification of Visual Basic.

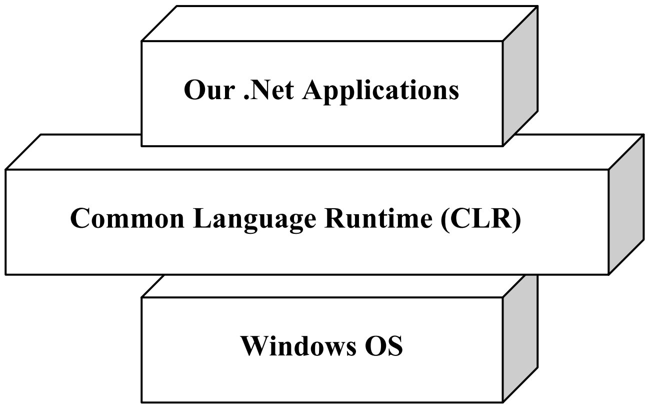
Like Java, C# also does not allow multiple inheritance or the use of pointers (in safe/managed code), but does provide garbage memory collection at runtime, type and memory access checking. However, contrary to JAVA, C# maintains the unique useful operations of C++ like operator overloading, enumerations, pre-processor directives, pointers (in unmanaged/un-safe code), function pointers (in the form of delegates) and promises to have template support in the next versions. Like VB, it also supports the concepts of properties (context sensitive fields). In addition to this, C# comes up with some new and exciting features such as reflections, attributes, marshalling, remoting, threads, streams, data access with ADO.Net and more

The .Net Architecture and .Net Framework

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

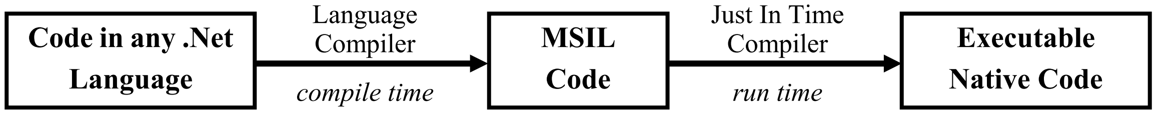
The Common Language Runtime (CLR)

The most important concept of the .Net Framework is the existence and functionality of the .Net Common Language Runtime (CLR), also called .Net Runtime for 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 (such as C#, VB.Net or C++.Net) our source code does not get converted into the executable binary code, but to an intermediate code known as MSIL which is interpreted by the Common Language Runtime. MSIL is operating system and hardware independent code. Upon program execution, this MSIL (intermediate code) is converted to binary executable code (native code). Cross language relationships are possible as the MSIL code is similar for each .Net language.



Just In Time Compilers (JITers)

When our IL compiled code needs to be executed, the CLR invokes the JIT compiler, 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 some time (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).

The 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.

The 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.

The Common Type System (CTS)

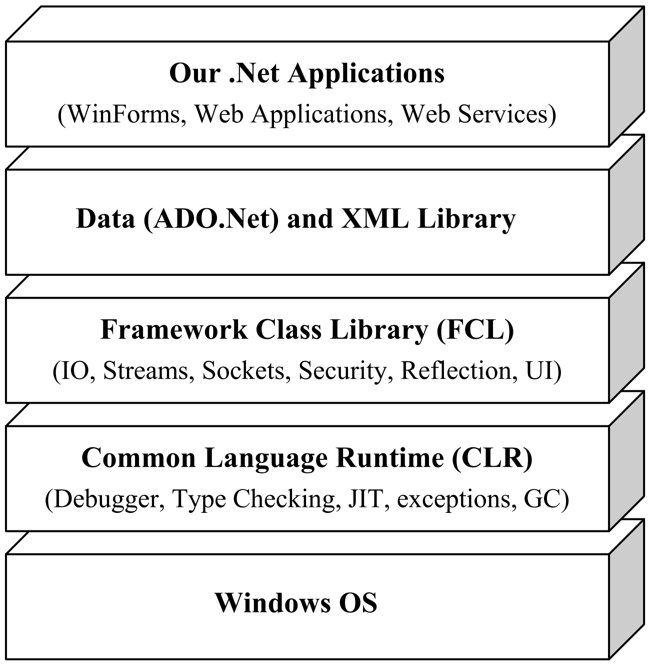
.Net also defines a Common Type System (CTS). Like CLS, CTS is also a set of standards. CTS defines the basic data types that IL understands. Each .Net compliant language should map its data types to these standard data types. This makes it possible for the 2 languages to communicate with each other by passing/receiving parameters to and 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 Collection (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. 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!

The .Net Framework

The .Net Framework is the combination of layers of CLR, FCL, Data and XML Classes and our Windows, Web applications and Web Services. A diagram of the .Net Framework is presented below for better understanding.


Our .Net Applications
(WinForms, Web Applications, Web Services)
Data (ADO.Net) and XML Library
Framework Class Library (FCL)
(IO, Streams, Sockets, Security, Reflection, UI)
Common Language Runtime (CLR)
(Debugger, Type Checking, JIT, exceptions, GC)
Windows OS

C# compared to C++

In terms of performance and efficiency in the use of memory and other resources, C++ does outclass C#. But are performance and speed the only measure when it comes to choosing a development environment? No! C++ is no doubt a very complex, abstract and low-level language to program. It burdens programmer with many responsibilities and less support. Another problem with C++ is that it is a vast language at its core, with too many language constructs, and many of those are very repetitive; e.g., to handle the data in memory you can either use variable, pointer or reference. C# at its core is very simple yet powerful modern language which has been made while keeping in mind the experience of developers over the years and their current needs. One of the biggest concerns in the development market is to make programs that are re-usable, maintainable, scalable, portable and easy to debug. C# comes right with these issues. Every single C++ developer knows how difficult it is to manage bigger C++ program and debug them. It can be a nightmare to find the reason why a program crashes randomly. The sole reason for this, to me, is the backwards-compatibility curse. They made C++ on the structure of C, a structured programming language, so it never became a true object oriented programming language only, and if the compiler allows me to go in the structured way of solving problem, who are you to make me take an object oriented approach? Also Bjarne Stroustrup, the founder of C++ said, "C++ is a multi-paradigm language not only OO language". The prime advantages of using C# include support for the common language runtime, Framework class library, and a new, clean object oriented design free of the backwards-compatibility curse.

The Visual Studio.Net IDE

Microsoft Visual Studio.Net is an Integrated Development Environment (IDE), which is the successor of Visual Studio 6. It eases the development process of the .Net Applications (VC#.Net, VB.Net, VC++.Net, JScript.Net, J#.Net, ASP.Net, and more). The revolutionary approach in this new improved version is that for all the Visual Studio.Net Compliant Languages use 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: The IDE provides various useful development tools such as:

- Keyword and syntax highlighting
- Intellisense (auto complete), which helps by automatically completing the syntax as you type a dot (.) with objects, enumerations, namespaces and when you use the “New” keyword.
- Project and solution management with solution explorer that helps to manage applications consisting of multiple files.
- Help building user interface with simple drag and drop support.
- Properties tab that allow you to set different properties for multiple windows and web controls.
- Standard debugger that allows you to debug your program using putting break points for observing run-time behavior.
- Hot compiler that checks the syntax of your code as you type it and error notification.
- Dynamic Help on a number of topics using the Microsoft Development Network (MSDN) library.
- Compiling and building applications.
- Program Execution with or without the debugger.
- Deploying your .Net application over the Internet or to disk.

Projects 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 '.csproj' where 'cs' represents C-Sharp. Similarly, VB.Net projects are stored as '.vbproj' 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.

Writing Your First Hello World Console Application in C#

In the following text, we will build our first C# application with, and then without, Visual Studio.Net. We will see how to write, compile, and execute the C# application. Later in the chapter, we will explain the different concepts in the program.

Working Without Visual Studio.Net

Open Notepad, or any other text editor, and write the following code:

```
 using System;

namespace MyHelloWorldApplication
{
    class HelloWorld
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

Save this with any file name with the extension ".cs". Example: 'MyFirstApplication.cs' To compile this file, go to command prompt and write:

```
    csc MyFirstApplication.cs
```

This will compile your program and create an .exe file (MyFirstApplication.exe) in the same directory and will report any errors that may occur.

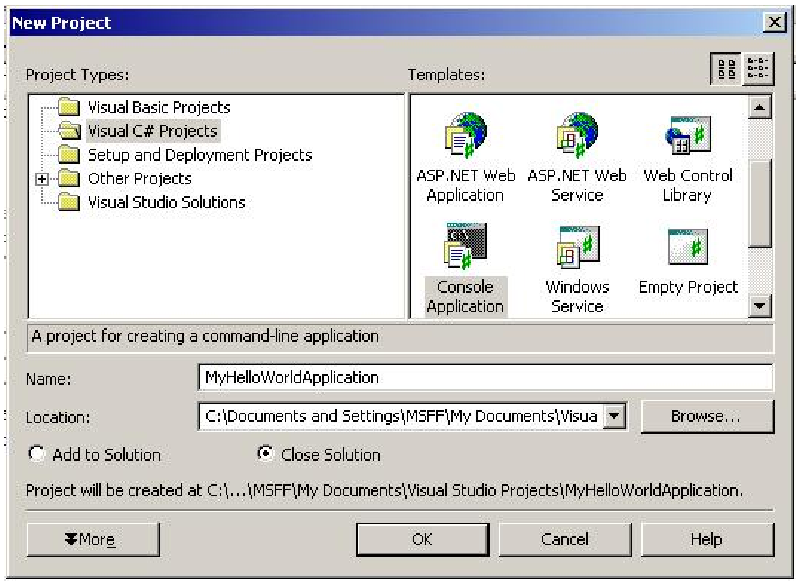
To run your program, type:

```
MyFirstApplication
```

This will print Hello World as a result on your console screen. Simple, isn't it? Let's do the same procedure with Visual Studio.Net:

With Visual Studio.Net

Start Microsoft Visual Studio.Net and select File - New - Project; this will show the open file dialog. Select Visual C# Project from Project Type and select Console Application from Template. Write MyHelloWorldApplication in the name text box below and click OK.



A project for creating a command-line application

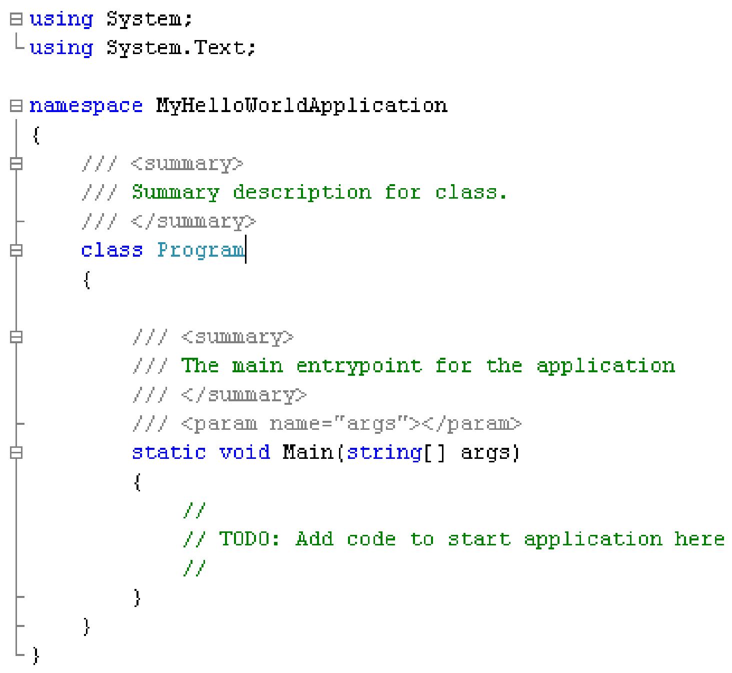
Name: MyHelloWorldApplication

Location: C:\Documents and Settings\MSFF\My Documents\Visua

Add to Solution [✓ Close Solution]

Project will be created at C:\...\MSFF\My Documents\Visual Studio Projects\MyHelloWorldApplication.

This will show you the initial default code for your hello world application:



```
using System;
using System.Text;

namespace MyHelloWorldApplication
{
    /// <summary>
    /// Summary description for class.
    /// </summary>
    class Program
    {

        /// <summary>
        /// The main entrypoint for the application
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start application here
            //
        }
    }
}
```

Remove the documentation comments (lines starting with ///), change the name of class from Class1 to HelloWorld and write

```
Console.WriteLine("Hello World"); in place of //TODO: Add code....
```

to make the picture look like

```
using System;
namespace MyHelloWorldApplication
{
    class HelloWorld
    {
        [STAThread]
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

Now to compile and execute your application, select Debug - Start Without Debugging or press Ctrl+F5. This will open a new Console Window with Hello World written in it. Once you press any key, the window will close, terminating the program.

Understanding the Hello World Application Code:

The first line of our program (using System;) appears in virtually all the C# programs. It gives us access to the core functionality of our computer system. We will discuss this a bit later. Let's first see what the second line (namespace MyHelloWorldApplication) means.

Namespaces in C#

A Namespace is simply a logical collection of related classes in C#. We bundle our related classes (like those related with database activity) in some named collection calling it a namespace (e.g., DataActivity). As C# does not allow two classes with the same name to be used in a program, the sole purpose of using namespaces is to prevent name conflicts. This may happen when you have a large number of classes, as is the case in the Framework Class Library (FCL). It is very much possible that our Connection Class in DataActivity conflicts with the Connection Class of InternetActivity. To avoid this, these classes are made part of their respective namespace. So the fully qualified name of these classes will be DataActivity.Connection and InternetActivity.Connection, hence resolving any ambiguity for the compiler.

So, in the second line of our program we are declaring that the following classes (within { } block) are part of MyHelloWorldApplication namespace.

```
namespace MyHelloWorldApplication
{
...
}
```

The C# namespaces have NO physical mapping as is the case in Java. Classes with same namespace can be in different folders. The C# concept of mapping is similar to packages in Java and namespaces in standard C++. The namespace may contain classes, events, exceptions, delegates and even other namespaces called 'Internal namespace'.

These internal namespaces can be defined like this:

```
namespace Parent
{
    namespace Child
    {
    ...
    }
}
```

The using Keyword

The first line of our program was:

```
using System;
```

The using keyword above allows us to use the classes in the following 'System' namespace. By doing this, we can now access all the classes defined in the System namespace like we are able to access the Console class in our Main method later. One point to remember here is using allows you to access the classes in the referenced namespace only and not in its internal/child namespaces. Hence we might need to write

```
using System.Collections;
```

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

The class Keyword

All of our C# programs contain at least one class. The Main() method resides in one of these classes. Classes are a combination of data (fields) and functions (methods) that can be performed on this data in order to achieve the solution to our problem. We will see the concept of class in more detail in the coming days. Classes in C# are defined using the class keyword followed by the name of class.

The Main() Method

In the next line we defined the Main() method of our program:

```
static void Main(string[] args)
```

This is the standard signature of the Main method in C#. The Main method is the entry point of our program, i.e., our C# program starts its execution from the first line of Main method and terminates with the termination of Main method. The Main method is designated as static as it will be called by the Common Language Runtime (CLR) without making any object of our HelloWorld Class (which is the definition of static methods, fields and properties). The method is also declared void as it does not return anything. Main is the (standard) name of this method, while string [] args is the list of parameters that can be passed to main while executing the program from command line. We will see this later.

One interesting point here is that it is legitimate to have multiple Main() methods in C# program. But, you have to explicitly identify which Main method is the entry point at the run-time. C++ and Java Programmers, take note that Main starts with capital 'M' and the return type is void.

Printing on the Console

Our next line prints Hello World on the Console screen:

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

Here we called WriteLine(), a static method of the Console class defined in the System namespace. This method takes a string (enclosed in double quotes) as its parameter and prints it on the Console window.

C#, like other Object Oriented languages, uses the dot (.) operator to access the member variables (fields) and methods of a class. Also, braces () are used to identify methods in the code and string literals are enclosed in double quotation marks ("). Lastly, each statement in C# (like C, C++ and Java) ends with a semicolon (;), also called the statement terminator.

Comments

Comments are the programmer's text to explain the code, are ignored by the compiler and are not included in the final executable code. C# uses syntax for comments that is similar to Java and C++. The text following double slash marks (// any comment) are line comments. The comment ends with the end of the line:

```
// This is my main method of program
static void Main()
{
...
}
```

C# also supports the comment block. In this case, the whole block is ignored by the compiler. The start of the block is declared by slash-asterisk (/*) and ends with asterisk-slash mark (*/):

```
static void Main()
{
    /* These lines of text
       will be ignored by the compiler */
    ...
    }
```

C# introduces another kind of comment called 'documentation comments'. C# can use these to generate the documentation for your classes and program. These are line comments and start with triple slash mark (///):

```
    /// These are documentation comments
```

We will discuss these in detail in coming issues.

Important points to remember

- Your C# executable program resides in some class.
- The entry point to program is the static method Main() with void return type
- C# is a case sensitive language so void and Void are different
- Whitespaces (enter, tab, space) are ignored by the compiler between the code. Hence, the following is also a valid declaration of the Main() method although it is not recommended:

```
static        void
    Main   (   )
{
...
}
```

- You DON'T need to save your program with same file name as of your class containing Main() method
- There can be multiple Main() methods in your program
- The boundaries of namespace, class and method are defined by opening and closing curly brackets { }
- A namespace is only logical collection of classes with no physical mapping on disk (unlike Java)
- The using keyword is used to inform compiler where to search for the definition of classes (namespaces) that you are about to use in your C# program.
- The three types of comments exist in C#; line, block and documentation. These are ignored by the compiler and are used only to enhance the readability and understandability of program for the developers.
- Enclosing your class in some namespace is optional. You can write program where your class is not enclosed by any namespace
- It is not mandatory that Main Method of program takes 'string [] args' as parameter. It is perfectly valid to write Main method as:

```
static void Main()
{
...
}
```

A more interactive Hello World Application

Up until now, we have seen a very static hello world application that greets the whole world when it is executed. Let’s now make a more interactive hello world that greets the current user of it. This program will ask the user their name and will greet using his/her name, like 'Hello Faraz' when a user named 'Faraz' runs it. Let’s see the code first:

```
    static void Main(string[] args)
    {
        Console.Write("Please enter your name: ");
        string name = Console.ReadLine();
        Console.WriteLine
          ("Hello {0}, Good Luck in C#", name);
    }
```

Author's Note: In the above code, we haven't shown the complete program but only the Main Method to save space. We will follow this strategy in the rest of the course when appropriate.

Discussing a more interactive Hello World Application

In the first line, we have used another method, Write(), of the Console class. This is similar to the WriteLine() method, discussed in the previous program, but does not change the line after printing the string on the console.

In the second line, we declared a variable of the type string and called it 'name'. Then, we took a line of input from the user through the ReadLine() method of the Console class and stored the result in the 'name' variable. The variables are placeholders in memory for storing data temporarily during the execution of program. Variables can hold different types of data depending on their data-type, e.g., int variables can store integers while string variables can store a string (series) of characters. The ReadLine() method of the Console class (contrary to WriteLine()) reads a line of input given at the Console Window. It returns this input as string data, which we stored in our string variable 'name'.

Author's Note:
A string is implicit data-type in C# contrary to other languages. It starts with small 's'.

In the third line, we printed the name given by user in line 2, along with some greeting text using the WriteLine() method of the Console class. Here we used the substitution parameter {0} to state where in the line the data in the variable 'name' should be written when the WriteLine() method is called.

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

When the compiler finds a substitution parameter, {n}, it replaces it with the (n+1)th variable following the string. The string is delimited with double quotation marks and each parameter is separated by a comma. Hence, in our case when the compiler finds {0}, it replaces it with (0+1)th, i.e., the 1st variable ('name') following the string. So at run-time, the CLR will read it as:

```
        Console.WriteLine
          ("Hello Faraz, Good Luck in C#");
```

if the value of 'name' is Faraz at run-time. Alternatively, it can also be written as:

```
        Console.WriteLine
          ("Hello " + name + ", Good Luck in C#");
```

removing the substitution parameter. 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:

```
Please enter your name: Faraz
Hello Faraz, Good Luck in C#
```

2. C# Language Fundamentals

Lesson Plan

Today we will learn the language fundamentals of C#. We will explore the data types in C#, using variables, operators, flow control statements like if.. else, looping structure and how to use arrays.

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

There are two kinds of data types in C#.

- Value Types (implicit data types, structs and enumeration)
- Reference Types (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 issue, implicit data types in .Net compliant languages are mapped to types in the Common Type System (CTS) and CLS (Common Language Specification). Hence, each implicit data type in C# has its corresponding .Net type. The implicit data types in C# are:

C# type	.Net type	Size in bytes	Description
Integral Types			
byte	Byte	1	May contain integers from 0-255
sbyte	SByte	1	Signed byte from -128 to 127
short	Int16	2	Ranges from -32,768 to 32,767
ushort	UInt16	2	Unsigned, ranges from 0 to 65,535
int (default)	Int32	4	Ranges from -2,147,483,648 to 2,147,483,647
uint	UInt32	4	Unsigned, ranges from 0 to 4,294,967,295
long	Int64	8	Ranges from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

ulong	UInt64	8	Unsigned, ranges from 0 to 18,446,744,073,709,551,615
Floating Point Types			
float	Single	4	Ranges from ±1.5 × 10^{-45} to ±3.4 × 10^{38} with 7 digits precision. Requires the suffix 'f' or 'F'
double (default)	Double	8	Ranges from ±5.0 × 10^{-324} to ±1.7 × 10^{308} with 15-16 digits Precision
Other Types			
bool	Boolean	1	Contains either true or false
char	Char	2	Contains any single Unicode character enclosed in single quotation mark such as 'c'
decimal	Decimal	12	Ranges from 1.0 × 10^{-28} to 7.9 × 10^{28} with 28-29 digits precision. Requires the suffix 'm' or 'M'

Implicit data types are represented in language using keywords, so each of the above is a keyword in C# (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 C#, so string is a keyword in C#. The last point about implicit data types is that they are value types and thus stored on the stack, while user defined types or referenced types are stored using the heap. A stack is a data structure that store items in a first in first out (FIFO) fashion. It is an area of memory supported by the processor and its size is determined at the compile time. A heap consists of memory available to the program at run time. Reference types are allocated using memory available from the heap dynamically (during the execution of program). The 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 a program, data is temporarily stored in memory. A variable is the name given to a memory location holding a particular type of data. So, each variable has associated with it a data type and a value. In C#, variables are declared as:

```
<data type> <variable>;
```

e.g.,

```
int i;
```

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

```
    bool  isReady       = true;
    float percentage    = 87.88, average = 43.9;
    char  digit         = '7';
```

In C# (like other modern languages), you must declare variables before using them. Also, there is the concept of "Definite Assignment" in C# which says "local variables (variables defined in a method) must be initialized before being used". The following program won't compile:

```
    static void Main()
    {
        int age;
        // age = 18;
        Console.WriteLine(age);    // error
    }
```

But, if you un-comment the 2nd line, the program will compile. C# does not assign default values to local variables. C# is also a type safe language, i.e., values of particular data type can only be stored in their respective (or compatible) data type. You can't store integer values in Boolean data types like we used to do in C/C++.

Constant Variables or Symbols

Constants are variables whose values, once defined, can not be changed by the program. Constant variables are declared using the const keyword, like:

```
    const double PI = 3.142;
```

Constant variables must be initialized as they are declared. It is a syntax error to write:

```
    const int MARKS;
```

It is conventional to use capital letters when naming constant variables.

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 variables and methods should start with a 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 that you strictly follow the convention. Microsoft no longer supports Hungarian notation, like using iMarks for integer variable. Also, using the underscore _ in identifiers is not encouraged.

Operators in C#

Arithmetic Operators

Several common arithmetic operators are allowed in C#.

Operand	Description
+	Add
-	Subtract
*	Multiply
/	Divide
%	Remainder or modulo
++	Increment by 1
--	Decrement by 1

The program below uses these operators.

```
using System;

namespace CSharpSchool
{
    class ArithmeticOperators
    {
        // The program shows the use of arithmetic operators
        // + - * / % ++ --
        static void Main()
        {
            // result of addition, subtraction,
            // multiplication and modulus operator
            int sum=0, difference=0, product=0, modulo=0;
            float quotient=0;        // result of division
            int num1 = 10, num2 = 2; // operand variables

            sum          = num1 + num2;
            difference   = num1 - num2;
            product      = num1 * num2;
            quotient     = num1 / num2;

            // remainder of 3/2
            modulo       = 3 % num2;

            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);

            num1++;         // increment num1 by 1
            num2--;         // decrement num2 by 1

            Console.WriteLine("num1 = {0}, num2 = {1}", num1, num2);
        }
    }
}
```

Although the program above is quite simple, I would like to discuss some concepts here. 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 the num1, num2 and sum variables. In {i}, i specifies that (i+1)th variable after double quotes will replace it when printed to the Console. Hence, {0} will be replaced by the first one, {1} will be replaced by the second variable and so on...

Another point to note is that num1++ has the same meaning as:

```
    num1 = num1 + 1;
```

Or:

```
    num1 += 1;
```

(We will see the description of second statement shortly)

Prefix and Postfix notation

Both the ++ and -– operators can be used as prefix or postfix operators. In prefix form:

```
num1 = 3;
num2 = ++num1;   // num1 = 4, num2 = 4
```

The compiler will first increment num1 by 1 and then will assign it to num2. While in postfix form:

```
num2 = num1++;   // num1 = 4, num2 = 3
```

The compiler will first assign num1 to num2 and then increment num1 by 1.

Assignment Operators

Assignment operators are used to assign values to variables. Common assignment operators in C# are:

Operand	Description
=	Simple assignment
+=	Additive assignment
-=	Subtractive assignment
*=	Multiplicative assignment
/=	Division assignment
%=	Modulo assignment

The equals (=) operator is used to assign a value to an object. Like we have seen

```
bool    isPaid = false;
```

assigns the value 'false' to the isPaid variable of Boolean type. The left hand and right hand side of the equal or any other assignment operator must be compatible, otherwise the compiler will complain about a syntax error. Sometimes casting is used for type conversion, e.g., to convert and store a value in a variable of type double to a variable of type int, we need to apply an integer cast.

```
double  doubleValue = 4.67;


// intValue will be equal to 4
int     intValue    = (int) doubleValue;
```

Of course, when casting there is always a danger of some loss of precision; in the case above, we only got the 4 of the original 4.67. Sometimes, the casting may result in strange values:

```
int      intValue    = 32800;
short   shortValue    = (short) intValue;
// shortValue would be equal to -32736
```

Variables of type short can only take values ranging from -32768 to 32767, so the cast above can not assign 32800 to shortValue. Hence shortValue took the last 16 bits (as a short consists of 16 bits) of the integer 32800, which gives the value -32736 (since bit 16, which represents the value 32768 in an int, now represents -32768). If you try to cast incompatible types like:

```
bool   isPaid      = false;
int    intValue    = (int) isPaid;
```

It won't get compiled and the compiler will generate a syntax error.

Relational Operators

Relational operators are used for comparison purposes in conditional statements. Common relational operators in C# are:

Operand	Description
==	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

```
int num1 = 5, num2 = 6;
```

Then:

```
num1 == num2  // false
num1 != num2  // true
num1 >  num2  // false
num1 <  num2  // true
num1 <= num2  // true
num1 >= num2  // false
```

Only compatible data types can be compared. It is invalid to compare a bool with an int, so if you have

```
    int     i = 1;
    bool    b = true;
```

you cannot compare i and b for equality (i==b). Trying to do so will result in a syntax error.

Logical and Bitwise Operators

These operators are used for logical and bitwise calculations. Common logical and bitwise operators in C# are:

Operand	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
!	Bitwise NOT
&&	“Logical” or “short circuit” AND
||	“Logical” or “short circuit” OR

The operators &, | and ^ are rarely used in usual programming practice. The NOT operator is used to negate a Boolean or bitwise expression like:

```
    bool    b     = false;
    bool    bb    = !b;
    // bb would be true
```

Logical Operators && and || are used to combine comparisons like

```
    int  i=6, j=12;
    bool firstVar  = i>3 && j<10;
    // firstVar would be false
    bool secondVar = i>3 || j<10;
    // secondVar would be true
```

In the first comparison: i>3 && j<10 will result in true only if both the conditions i>3 and j<10 result in true.
In the second comparison: i>3 || j<10 will result in true if any of the conditions i>3 and j<10 result in true. You can, of course, use both && and || in single statement like:

```
    bool firstVar   = (i>3 && j<10) || (i<7 && j>10)  // firstVar would be true
```

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

You can use & and | operators in place of && and || but for combining conditional expressions, && and || are more efficient because they use "short circuit evaluation". For example, if in the expression (i>3 && j<10), i>3 evaluates to false, the second expression j<10 won't be checked and false will be returned (when using AND, if one of the participant operands is false, the whole operation will result in false). Hence, one should be very careful when using assignment expressions with && and || operators. The & and | operators don't do short circuit evaluation and do execute all the comparisons before returning the result.

Other Operators

There are some other operators present in C#. A short description of these is given below:

Operand	Description
<<	Left shift bitwise operator
>>	Right shift bitwise operator
.	Member access for objects
[]	Index operator used in arrays and collections
()	Cast operator
?:	Ternary operator

Operator Precedence

All operators are not treated equally. There is a concept of "operator precedence" in C#. For example:

```
int i = 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

The if...else statement

Condition checking has always been the most important construct in any language right from the time of the assembly language days. C# provides conditional statements in the form of the if...else statement. The structure of this statement is:

```
if(Boolean expression)
    Statement or block of statements
else
    Statement or block of statements
```

The else clause above is optional. A typical example is:

```
if(i==5)
    Console.WriteLine("Thank God, I finally became 5.");
```

In the above example, the console message will be printed only if the expression i==5 evaluates to true. If you would like to take some action when the condition does not evaluate to true, then you can use else clause:

```
if(i==5)
    Console.WriteLine ("Thank God, I finally became 5.");
else
    Console.WriteLine("Missed...When will I become 5?");
```

Only the first message will be printed if i is equal to 5. In any other case (when i is not 5), the second message will be printed. If you want to use a block of statements (more than one statement) under if or else, you can enclose your block in {} brackets:

```
if(i==5)
{
    j = i*2;
    Console.WriteLine("Thank God, I finally became 5.");
}
else
{
    j = i/2;
    Console.WriteLine("Missed...When will I become 5?");
}
```

I would always recommend to use { } brackets to enclose the statements following if and else even if you only have a single statement. It increases readability and prevents many bugs that otherwise can result if you neglect the scope of if and else statements.

You can also have if after else for further conditioning:

```
if(i==5)        // line 1
{
    Console.WriteLine("Thank God, I finally became 5.");
}
else if(i==6)    // line 5
{
    Console.WriteLine("Ok, 6 is close to 5.");
}
```

```
else            // line 9
{
    Console.WriteLine("Missed...When will I become 5 or be close to 5?");
}
```

Here else if(i==6) is executed only if the first condition i==5 is false, and else at line 9 will be executed only if the second condition i==6 (line 5) executes and fails (that is, both the first and second conditions fail). The point here is else at line 9 is related to if on line 5.

Since if...else is also a statement, you can use it under other if...else statement (nesting), like:

```
if(i>5)         // line 1
{
    if(i==6)    // line 3
    {
        Console.WriteLine("Ok, 6 is close to 5.");
    }
    else        // line 7
    {
        Console.WriteLine("Oops! I'm older than 5 but not 6!");
    }
    Console.WriteLine("Thank God, I finally became older than 5.");
}
else            // line 13
{
    Console.WriteLine("Missed...When will I become 5 or close to 5?");
}
```

The else on line 7 is clearly related to if on line 3 while else on line 13 belongs to if on line 1. Finally, do note (C/C++ programmers especially) that if statements expect only Boolean expressions and not integer values. It's an error to write:

```
int flag = 0;
if(flag = 1)
{
    // do something...
}
```

Instead, you can either use:

```
int flag = 0;
if(flag == 1)    // note ==
{
    // do something...
}
```

or,

```
    bool flag = false;
    if(flag = true)    // Boolean expression
    {
        // do something...
    }
```

The keys to avoiding confusion in the use of any complex combination of if...else are:

- Habit of using {} brackets with every if and else.
- 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 indentation for you. Otherwise, you have to take care of this yourself.

I strongly recommend you follow the above two guidelines.

The switch...case statement

If you need to perform a series of specific checks, switch...case is present in C# just for this. The general structure of the switch...case statement is:

```
    switch(integral or string expression)
    {
        case constant-expression:
            statements
            breaking or jump statement
        // some other case blocks
        ...
        default:
            statements
            breaking or jump statement
    }
```

It takes less time to use switch...case than using several if...else if statements. Let's look at it with an example:

```
using System;
```

```
// To execute the program write "SwitchCaseExample 2" or
// any other number at command line,
// if the name of .exe file is "SwitchCaseExample.exe"
namespace CSharpSchool
{
    class SwitchCaseExample
    {
        // Demonstrates the use of switch...case statement along with
        // the use of command line argument
        static void Main(string [] userInput)
        {
            int input = int.Parse(userInput[0]);
            // convert the string input to integer.
            // Will throw a run-time exception if there is no input at run-time or if
            // the input is not castable to integer.
            switch(input)      // what is input?
            {
                case 1:        // if it is 1
                    Console.WriteLine("You typed 1 (one) as the first command line argument");
                    break;     // get out of switch block
                case 2:        // if it is 2
                    Console.WriteLine("You typed 2 (two) as the first command line argument");
                    break;     // get out of switch block
                case 3:        // if it is 3
                    Console.WriteLine("You typed 3 (three) as the first command line argument");
                    break;     // get out of switch block
                default:       // if it is not any of the above
                    Console.WriteLine("You typed a number other than 1, 2 and 3");
                    break;     // get out of switch block
            }
        }
    }
}
```

The program must be supplied with an integer command line argument. First, compile the program (at the command line or in Visual Studio.Net). Suppose we made an exe with name "SwitchCaseExample.exe", we would run it at the command line like this:

```
C:>SwitchCaseExample 2
You typed 2 (two) as command line argument
```

Or:

```
C:>SwitchCaseExample 34
You typed a number other than 1, 2 and 3
```

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

```
C:>SwitchCaseExample
Unhandled Exception: System.IndexOutOfRangeException:
  Index was outside the bounds of the array.
   at CSharpSchool.SwicthCaseExample.Main(String[] userInput) in
   c:\visual studio projects\SwitchCaseExample\
     SwitchCaseExample.cs :line 9
```

Let's get to internal working. First, we converted the first command line argument (userInput[0]) into an int variable input. For conversion, we used the static Parse() method of the int data type. This method takes a string and returns the equivalent integer or raises an exception if it can't. Next we checked the value of input variable using a switch statement:

```
switch(input)
{
...
}
```

Later, on the basis of the value of input, we took specific actions under respective case statements. Once our case specific statements end, we mark it with the break statement before the start of another case (or the default) block.

```
case 3:       // if it is 3
    Console.WriteLine("You typed 3 (three) as first command line argument");
    break;    // get out of switch block
```

If all the specific checks fail (input is none of 1,2 and 3), the statements under default executes.

```
default:
     // if it is not any of the above
    Console.WriteLine ("You typed a number other than 1, 2 and 3");
    break; // get out of switch block
```

There are some important points to remember when using the switch...case statement in C#:

- You can use either integers (enumeration) or strings in a switch statement
- The expression following case must be constant. It is illegal to use a variable after case:

```
case i:        // incorrect, syntax error
```

- A colon : is used after the case statement and not a semicolon ;
- You can use multiple statements under single case and default statements:

```
case "Pakistan":
    continent = "Asia";
    Console.WriteLine("Pakistan is an Asian Country");
    break;
default:
    continent = "Un-recognized";
    Console.WriteLine
      ("Un-recognized country discovered");
    break;
```

- The end of the case and default statements is marked with break (or goto) statement. We don't use {} brackets to mark the block in switch...case as we usually do in C#
- C# does not allow fall-through. So, you can't leave case or default without break statement (as you can in Java or C/C++). The compiler will detect and complain about the use of fall-through in the switch...case statement.
- The break statement transfers the execution control out of the current block.
- Statements under default will be executed if and only if all the case checks fail.
- It is not necessary to place default at the end of switch...case statement. You can even place the default block before the first case or in between cases; default will work the same regardless of its position. However, making default the last block is conventional and highly recommended. Of course, you can't have more than one default block in a single switch...case.

Loops In C#

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

The for Loop

The most common type of loop in C# is the for loop. The basic structure of a for loop is exactly the same as in Java and C/C++ and is:

```
for(assignment; condition; increment/decrement)
    statements or block of statements
    enclosed in {} brackets
```

Let's see a for loop that will write the integers from 1 to 10 to the console:

```
    for(int i=1; i<=10; i++)
    {
        Console.WriteLine("In the loop, the value of i is {0}.", i);
    }
```

At the start, the integer variable i is initialized with the value of 1. The statements in the for loop are executed while the condition (i<=10) remains true. The value of i is incremented (i++) by 1 each time the loop starts.

Some important points about the for loop

All three statements in for(), assignment, condition and increment/decrement are optional. You can use any combination of these and even decide not to use any one at all .This would make what is called and 'Indefinite or infinite loop' (a loop that will never end until or unless the break instruction is encountered inside the body of the loop). But, you still have to supply the appropriate semi colons:

```
for(; ;)
for( ; i<10; i++)
for(int i=3; ; i--)
for( ; i>5; )
```

If you don't use the {} brackets, the statement immediate following for() will be treated as the iteration statement. The example below is identical to the one given above:

```
for(int i=1; i<=10; i++)
    Console.WriteLine("In the loop, value of i is {0}.", i);
```

I will again recommend that you always use {} brackets and proper indentation.

If you declare a variable in for()'s assignment, its life (scope) will only last inside the loop and it will die after the loop body terminates (unlike some implementations of C++). Hence if you write:

```
for(int i=1; i<=10; i++)
{
    Console.WriteLine("In the loop, value of i is {0}.", i);
}
i++;         // line 1
```

The compiler will complain at line 1 that "i is an undeclared identifier".

You can use break and continue in for loops or any other loop to change the normal execution path. break terminates the loop and transfers the execution to a point just outside the for loop:

```
for(int i=1; i<=10; i++)
{
    if(i>5)
    {
        break;
    }
    Console.WriteLine("In the loop, value of i is {0}.", i);
}
```

The loop will terminate once the value of i gets greater than 5. If some statements are present after break, break must be enclosed under some condition, otherwise the lines following break will become unreachable and the compiler will generate a warning (in Java, it's a syntax error).

```
for(int i=3; i<10; i++)
{
    break;    // warning, Console.WriteLine (i); is unreachable code
    Console.WriteLine(i);
}
```

continue ignores the remaining part of the current iteration and starts the next iteration.

```
for(int i=1; i<=10; i++)
{
    if(i==5)
    {
        continue;
    }
    Console.WriteLine("In the loop, value of i is {0}.", i);
}
```

Console.WriteLine will be executed for each iteration except when the value of i becomes 5. The sample output of the above code is:

```
    In the loop, value of i is 1.
    In the loop, value of i is 2.
    In the loop, value of i is 3.
    In the loop, value of i is 4.
    In the loop, value of i is 6.
    In the loop, value of i is 7.
    In the loop, value of i is 8.
    In the loop, value of i is 9.
    In the loop, value of i is 10.
```

The do...while Loop

The general structure of a do...while loop is

```
    do
        Statement or block of statements
    while(boolean expression);
```

The statements under do will execute the first time and then the condition is checked. The loop will continue while the condition remains true.
The program for printing the integers 1 to 10 to the console using the do...while loop is:

```
int i=1;
do
{
    Console.WriteLine("In the loop, value of i is {0}.", i);
    i++;
} while(i<=10);
```

Some important points here are:

- The statements in a do...while() loop always execute at least once.
- There is a semicolon ; after the while statement.

while Loop

The while loop is similar to the do...while loop, except that it checks the condition before entering the first iteration (execution of code inside the body of the loop). The general form of a while loop is:

```
    while(Boolean expression)
        statements or block of statements
```

Our program to print the integers 1 to 10 to the console using while will be:

```
    int i=1;
    while(i<=10)
    {
        Console.WriteLine("In the loop, value of i is {0}.", i);
        i++;
    }
```

Arrays in C#

Array Declaration

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

```
    <data type> [] <identifier> = new <data type>[<size of array>];
```

Let's define an array of type int to hold 10 integers.

```
    int [] integers = new int[10];
```

The size of an array is fixed and must be defined before using it. However, you can use variables to define the size of the array:

```
    int size = 10;
    int [] integers = new int[size];
```

You can optionally do declaration and initialization in separate steps:

```
    int [] integers;
    integers = new int[10];
```

It is also possible to define arrays using the values it will hold by enclosing values in curly brackets and separating individual values with a comma:

```
    int [] integers = {1, 2, 3, 4, 5};
```

This will create an array of size 5, whose successive values will be 1, 2, 3, 4 and 5.

Accessing the values stored in an array

To access the values in an Array, we use the indexing operator [int index]. We do this by passing an int to indicate which particular index value we wish to access. It's important to note that index values in C# start from 0. So if an array contains 5 elements, the first element would be at index 0, the second at index 1 and the last (fifth) at index 4. The following lines demonstrate how to access the 3rd element of an array:

```
    int [] intArray = {5, 10, 15, 20};
    int j = intArray[2];
```

Let's make a program that uses an integral array.

```
    // demonstrates the use of arrays in C#
    static void Main()
    {
        // declaring and initializing an array of type integer
        int [] integers = {3, 7, 2, 14, 65};
        // iterating through the array and printing each element
        for(int i=0; i<5; i++)
        {
            Console.WriteLine(integers[i]);
        }
    }
```

Here we used the for loop to iterate through the array and 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 had to hard-code the size of the array in the for loop. As we mentioned earlier, arrays in C# are reference type and are a sub-class of the System.Array Class. This class has lot of useful properties and methods that can be applied to any instance of an array that we define. Properties are very much like the combination of getter and setter methods in common Object Oriented languages. Properties are context sensitive, which means that the compiler can un-ambiguously identify whether it should call the getter or setter in any given context. We will discuss properties in detail in the coming lessons. System.Array has a very useful read-only property named Length that can be used to find the length, or size, of an array programmatically. Using the Length property, the for loop in the above program can be written as:

```
    for(int i=0; i<integers.Length; i++)
    {
        Console.WriteLine(integers[i]);
    }
```

This version of looping is much more flexible and can be applied to an array of any size and of any data-type. Now we can understand the usual description of Main(). Main is usually declared as:

```
static void Main(string [] args)
```

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

foreach Loop

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

```
    foreach(<type of elements in collection> <identifier> in <array or collection>)
        <statements or block of statements>
```

Let's now make our previous program to iterate through the array with a foreach loop:

```
    // demonstrates the use of arrays in C#
    static void Main()
    {
        // declaring and initializing an array of type integer
        int [] integers = {3, 7, 2, 14, 65};
        // iterating through the array and printing each element
        foreach(int i in integers)
        {
            Console.WriteLine(i);
        }
    }
```

Simple and more readable, isn't it? In the statement:

```
    foreach(int i in integers)
```

We specified the type of elements in the collection (int in our case). We declared the variable (i) to be used to hold the individual values of the array 'integers' in each iteration.

Important points to note here:

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

```
    static void Main()
    {
        string name = "Faraz Rasheed";
        foreach(char ch in name)
        {
            Console.WriteLine(ch);
        }
    }
```

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

3. Classes and Objects

Lesson Plan

Today we will start Object Oriented Programming (OOP) in C#. We will start with learning classes, objects, and their basics. Then we will move to constructors, access modifiers, properties, method overloading and static methods.

Concept of a Class

A class is simply an abstract model used to define a new data types. A class may contain any combination of encapsulated data (fields or member variables), operations that can be performed on 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 C# is declared using the keyword class and its members are enclosed in parenthesis

```
class MyClass
{
    // fields, operations and properties go here
}
```

where MyClass is the name of class or new data type that we are defining here.

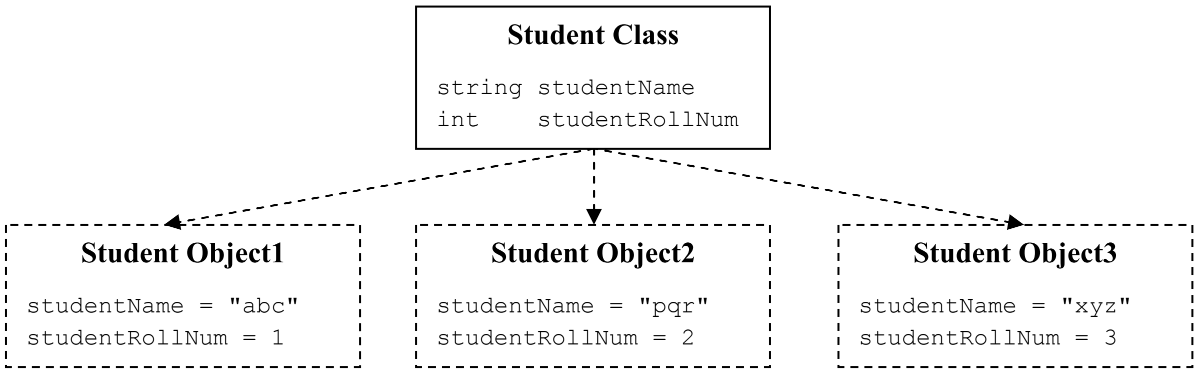
Objects

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

```
MyClass myObjectReference = new MyClass();
```

In the line above, we made an object of type MyClass which is referenced by an identifier myObjectReference.

The difference between classes and implicit data types is that objects are reference types (passed by reference) while implicit data types are value type (passed by making a copy). Also, objects are created at the heap while implicit data types are stored on 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
    string name;
    int     age;
    int     marksInMaths;
    int     marksInEnglish;
    int     marksInScience;
    int     totalMarks = 300;    // initialization
    int     obtainedMarks;
    double  percentage;
}
```

You can also initialize the fields with the initial values as we did in totalMarks in the example above. If you don't initialize the members of the class, they will be initialized with their default values.

Default values for different data types are shown below:

Data Type	Default Value
int	0
long	0
float	0.0
double	0.0
bool	False
char	'\0' (null character)
string	"" (empty string)
Objects	null

Methods

Methods are the operations performed on the data. A method may take some input values through its parameters and may return a value of a particular data type. The signature of the method takes the form

```
<return type> <name of method>(<data type> <identifier>,  <data type> <identifier>,...)
{
    // body of the method
}
```

For example,

```
int FindSum(int num1, int num2)
{
    int sum = num1 + num2;
    return sum;
}
```

Here, we defined a method named FindSum which takes two parameters of int type (num1 and num2) and returns a value of type int using the keyword return. If a method does not return anything, its return type would be void. A method can also optionally take no parameter (a parameterless method)

```
void ShowCurrentTime()
{
    Console.WriteLine("The current time is: " + DateTime.Now);
}
```

The above method takes no parameter and returns nothing. It only prints the Current Date and Time on the console using the DateTime Class in the System namespace.

Instantiating the class

In C# a class is instantiated (making its objects) using the new keyword.

```
Student theStudent = 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 line

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

Note that it is very similar to using implicit data types except for the object is created with the new operator while implicit data types are created using literals

```
int i;
i = 4;
```

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

```
Student theStudent;
```

only declares the reference theStudent of type Student which at this point does not contain any object (and points to the default null value) so if you try to access the members of class (Student) through it, it will throw a compile time error 'Use of unassigned variable theStudent'. When we write

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

then a new object of 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 dot '.' operator against the reference of the object like this:

```
Student theStudent = new Student();
theStudent.marksOfMaths = 93;
theStudent.CalculateTotal();
Console.WriteLine(theStudent.obtainedMarks);
```

Let's now make our Student class with some related fields, methods and then instantiate it in the Main() method.

```
using System;
namespace CSharpSchool
{
    // Defining a class to store and manipulate students information
    class Student
    {
        // fields
        string    name;
        int        age;
        int        marksOfMaths;
        int        marksOfEnglish;
        int        marksOfScience;
        int        totalMarks = 300;
        int        obtainedMarks;
        double    percentage;

        // methods
        void CalculateTotalMarks()
        {
            obtainedMarks = marksOfMaths + marksOfEnglish + marksOfScience;
        }

        void CalculatePercentage()
        {
            percentage = (double) obtainedMarks / totalMarks * 100;
        }

        double GetPercentage()
        {
            return percentage;
        }

        // Main method or entry point of program
        static void Main()
        {
            // creating new instance of Student
            Student st1 = new Student();
            // setting the values of fields
            st1.name = "Einstein";
            st1.age = 20;
            st1.marksOfEnglish = 80;
            st1.marksOfMaths = 99;
            st1.marksOfScience = 96;
            // calling functions
            st1.CalculateTotalMarks();
            st1.CalculatePercentage();
            double st1Percentage = st1.GetPercentage();
            // calling and retrieving value
            // returned by the function

            Student st2 = new Student();
            st2.name = "Newton";
            st2.age = 23;
            st2.marksOfEnglish = 77;
            st2.marksOfMaths = 100;
            st2.marksOfScience = 99;
            st2.CalculateTotalMarks();
            st2.CalculatePercentage();
            double st2Percentage = 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);
        }
    }
}
```

Here, 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 totalMarks and percentage, then we retrieved and stored the percentage in a variable and finally printed these on a console window.

We repeated the same steps again to create another object of type Student, set and printed its attributes. Hence in this way, you can create as many object 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, everyone 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 getting beyond 100%, which in most cases we like to restrict. C# provides access modifiers or accessibility levels just for this purpose, i.e., restricting access to a particular member. There are 5 access modifiers that can be applied to any member of the class. We are listing these along with short description in the order of decreasing restriction

Access Modifier	Description
private	private members can only be accessed within the class that contains them
protected internal	This type of member can be accessed from the current project or from the types inherited from their containing type
internal	Can only be accessed from the current project
protected	Can be accessed from a containing class and types inherited from the containing class
public	public members are not restricted to anyone. Anyone who can see them can also access them.

In Object Oriented Programming (OOP) it is always advised and recommended to mark all your fields as private and allow the user of your class to access only 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 public.

```
class Student
{
    // fields
    private string    name;
    private int        age;
    private int        marksOfMaths;
    private int        marksOfEnglish;
    private int        marksOfScience;
    private int        totalMarks = 300;
    private int        obtainedMarks;
    private double    percentage;

    // methods
    public void CalculateTotalMarks()
    {
        obtainedMarks = marksOfMaths +  marksOfEnglish + marksOfScience;
    }

    public void CalculatePercentage()
    {
        percentage = (double) obtainedMarks / totalMarks * 100;
    }

    public double GetPercentage()
    {
        return percentage;
        }
    }
```

If you don't mark any member of class with an 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 C# such as the class, interface, struct, enum, delegate and event. For top-level types (types not bound by any other type except namespace) like class, interface, struct and enum you can only use public and internal access modifiers with the same meaning as described above. In fact other access modifiers don't make sense to these types. Finally you can not apply access modifiers to 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. C# is the first language to provide 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 the value) and setters (to assign the value) were defined like if we have a private field name

```
    private string name;
```

then, the getters and setters would be like

```
    // getter to name field
    public string GetName()
    {
        return name;
    }

    // setter to name field
    public void SetName(string theName)
    {
        name = theName;
    }
```

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

```
    private int totalMarks;
    public int GetTotalMarks()
    {
        return totalMarks;
    }
```

Hence outside the class, one can only read the value of totalMarks and can not modify it. You can also decide to check some condition before assigning a value to your field

```
    private int marksOfMaths;
    public void SetMarksOfMaths(int marks)
    {
        if(marks >= 0 && marks <=100)
        {
            marksOfMaths = marks;
        }
        else
        {
            marksOfMaths = 0;
            // or throw some exception informing user marks out of range
        }
    }
```

This procedure gives you a lot of control over how fields of your classes should be accessed and dealt in a program. But, the problem is this you need to define two methods and have to prefix the name of your fields with Get or Set. C# provides the 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 control over how the fields can be accessed.

Using Properties

The general Syntax for Properties is

```
    <access modifier> <data type> <name of property>
    {
        get
        {
            // some optional statements
            return <some private field>;
        }
        set
        {
            // some optional statements;
            <some private field> = value;
        }
    }
```

Didn't understand it? No problem. Let's clarify it with an example: we have a private field name

```
    private string name;
```

We decide to define a property for this providing both getters and setters. We will simply write

```
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
```

We defined a property called 'Name' and provided both a getter and a setter in the form of get { } and set { } blocks. Note that we called our property 'Name' which is accessing the private field 'name'. It is becoming convention to name the property the same as the corresponding field but with first letter in uppercase (for name->Name, for percentage->Percentage). As properties are accessors to certain fields, they are mostly marked as public while the corresponding field is (and should be) mostly private. Finally note in the set { } block, we wrote

```
    name = value;
```

Here, value is a keyword and contains the value passed when a property is called. In our program we will use our property as

```
Student theStudent = new Student();
theStudent.Name = "Faraz";
string myName = theString.Name;
theStudent.name = "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 { } block of the properties passing "Faraz" as a value (which is a keyword). In the next line when we write

```
string myName = 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 { } block of property Name which will return the contents of the private field name ("Faraz" in this case, as we assigned in line 2) which will be stored in the local string variable name. Hence, when compiler finds the use of a property, it checks in which context it is called and takes appropriate action with respect to the context.

The last line

```
theStudent.name = "Someone not Faraz";    // error
```

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

You can give the definition of either of get { } or set { } block. If you miss one of these, and user tries to call it, he/she will get compile time error. For example the Length property in String class is read only; that is, the implementers have only given the definition of get { } block. You can write statements in the get { }, set { } blocks as you do in methods.

```
    private int marksOfMaths;

    public int MarksOfMaths
    {
        set
        {
            if(value >= 0 && value<=100)
            {
                marksOfMaths = value;
            }
            else
            {
                marksOfMaths = 0;
                // or throw some exception informing user marks out of range
            }
        }
    }
```

Precautions when using properties

- Properties don't have argument lists; set, get and value are keywords in C#
- The data type of value is the same as the type of property you declared when declaring the property
- ALWAYS use proper curly brackets { } and proper indentation while using properties.

- DON'T try to write the set { } or get { } block in a single line
- UNLESS your property only assigns and retrieve values from the private fields like

```
get { return name; }
set { name = value; }
```

Each object has a reference this which points to itself. Suppose in some method call, our object needs to pass 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 FileSystem class which takes the object to store as its parameter.

```
    class Student
    {
        string name = "Some Student";
        int age;

        public void Store()
        {
            FileSystem fs = new FileSystem();
            fs.save(this);
        }
    }
```

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

```
    class Test
    {
        public static void Main()
        {
            Student theStudent = new Student();
            theStudent.Store();
        }
    }
```

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

```
    class Student
    {
        private string name;
        private int age;

        public Student(string name, int age)
        {
            this.name = name;
            this.age = age;
        }
    }
```

Here in the constructor when we use name or age, we actually get the variables passed in the method which overshadow the instance members (fields) with same name. Hence, to use our fields, we had to use this to distinguish our instance members (fields) with the members passed through the parameters.

This is an extremely useful, widely and commonly used construct. I recommend you practice with "this" for some time until you feel comfortable with it.

Static Members of the class

All the members of the classes that we have seen up till now are instance members, meaning 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 are called static. Static members belong to the whole class rather than to individual object. For example, if you have a static phoneNumber field in your Student class, then there will be the 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. Static members are defined using keyword static

```
    class Student
    {
        public static int phoneNumber;
        public int rollNumber;
    }
```

Static members are accessed with the name of class rather than reference to objects. Let’s make our Test class containing Main method

```
    class Test
    {
        public static void Main()
        {
            Student st1 = new Student();
            Student st2 = new Student();
            st1.rollNumber = 3;
            st2.rollNumber = 5;
            Student.phoneNumber = 4929067;
        }
    }
```

Here you can see that the phoneNumber is accessed without any reference to the object but with the name of the class it belongs. Static methods are very useful while programming. In fact, the WriteLine() and ReadLine() methods that we are using from the start are static methods of Console class. That is the reason why we used to call them with reference to their class rather than making an object of the Console class. I hope now you are able to understand the syntax of the Main method in C# in full. It is declared static as CLR calls it without making any instance of our class. Static variables are useful when you want to cache data that should be available to all objects of the class. You can use static fields, methods, properties and even constructors which will be called before any instance of the class is created. Static constructor are declared like

```
    static Student()
    {
        name="unknown";
    }
```

As static methods may be called without any reference to object, you can not use instance members inside static methods or properties, while you may call a static member from a non-static context. The reason for being able to call static members from non-static context is that static 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 static int phoneNumber;
        public int rollNumber;

        public void DoWork()
        {
            // legal, static method called in non-static context
            MyMethod();
        }

        public static void MyMethod()
        {
            // legal, static field used in static context
            phoneNumber++;
            // illegal, non-static field used in static context
            rollNumber++;
            // illegal, non-static method used in static context
            DoWork();
        }
    }
```

Some precautionary points in the end

- Don't put too many static methods in your class as it is against the object oriented design principles and makes your class less extensible.
- Don't try to make a class with only the static methods and properties unless you have very good reason for doing this.
- You tend to loose a number of object oriented advantages while using static methods, as static methods can't be overridden which means it can not be used polymorphically, something widely used in the Object Oriented Paradigm of programming.

Some More about Methods

We mentioned earlier that there are two kinds of 'types' in C#: Value types and Reference types. Value types, such as implicit data types, are passed to methods by value. Reference types, like objects and arrays, are passed by reference.

Constructors

Constructors are a special kind of method. A Constructor has the following properties:

- It has the same name as its containing class
- It has no return type
- It is automatically called when a new instance or object of a class is created, hence why it's called a constructor.
- The constructor contains initialization code for each object, like assigning default values to the fields.

Let us see some examples.

```
using System;

class Person
{
    // field
    private string name;

    // constructor
    public Person()
    {
        name = "unknown";
        Console.WriteLine("Constructor called...");
    }


    // property
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

In the Person class above, we have a private field name, a public constructor which initializes the name field with string "unknown" and prints that it has been called, then we have a public property to read/write the private field name. Lets make another class Test which contains the Main() method and which uses the Person class

```
class Test
{
    public static void Main()
    {
        Person thePerson = 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);
    }
}
```

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

```
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 string "unknown". In fact, when we create a new object, we actually call the constructor of the class:

```
    Person thePerson = new Person();
```

That is why constructor is usually made public. If you make your constructor private, no one would be able to make an object of your class outside of it (though a method in the class of course could). That is, if the Person class is defined as:

```
class Person
{
    private Person()
    {
    }
}
```

then it would cause an error to write:

```
    class Test
    {
        public static void Main()
        {
            Person thePerson = new Person(); // error
        }
    }
```

The constructors shown so far have been parameter-less, i.e. they do not take any parameters. We can define constructors which take some parameters.

```
class Person
{
    private string name;

    public Person(string theName)
    {
        name = theName;
        Console.WriteLine("Constructor called...");
    }
}
```

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

```
    Person thePerson = new Person("Faraz");
```

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

Finalize() Method of Object class

Each class in C# is automatically (implicitly) inherited from the Object class which contains a 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 code for freeing resources that you reserved when using the object. For example,

```
protected override void Finalize()
{
    try
    {
        Console.WriteLine("Destructing object...");
        // put some code here
    }
    finally
    {
        base.Finalize();
    }
}
```

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

Destructors

Destructors are just the opposite of constructors. These are methods with the following properties
It has the same name as the containing class but prefixes it with the ~ (tilde) sign. It is called automatically when the object is about to be destructed (when garbage collector is about to destroy your object). It has no return type. We declare the destructor as

```
class Person
{
    // constructor
    public Person()
    {
    }

    // destructor
    ~Person()
    {
        // put resource freeing code here.
    }
}
```

As a matter of fact, the C# compiler internally converts the destructor to the Finalize() method, we just saw above. Destructors are not used very much in common C# programming practice (that is why Java dropped the idea of destructors). In the days of C++, programmers had to manage memory allocation and de-allocation. Destructors were used there to free the memory allocated by the object dynamically. 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 with the same name and return type but with a different number and type of arguments (parameters). This is called method overloading. For example it is perfectly legal to write:

```
class Checker
{
    // 1st overloaded form
    public bool isDefaultValue(bool val)
    {
        if(val == false)
            return true;
        else
            return false;
    }

    // 2nd overloaded form
    public bool isDefaultValue(int val)
    {
        if(val == 0)
            return true;
        else
            return false;
    }

    // 3rd overloaded form
    public bool isDefaultValue(int intVal, bool booleanVal)
    {
        if(intVal == 0 && booleanVal == false)
            return true;
        else
            return false;
    }
}
```

In the checker class above we defined three methods with the name isDefaultValue(). The return type of all these is bool but all differ from each other in parameter list. The first two differ in the data type of the parameters while the third one differs in the number of parameters. When isDefaultValue() is called, the compiler will decide (on the basis of the types and number of parameters being passed) which one of these three to actually call. For example, in our Main() method:

```
    Checker check = new Checker();
    Console.WriteLine(check.isDefaultValue(5));            // calls the first one
    Console.WriteLine(check.isDefaultValue(false));      // calls the second one
    Console.WriteLine(check.isDefaultValue(0, true));   // calls the third one
```

Remember that methods are overloaded depending on the parameter list and not on the return type. The WriteLine() method of Console class in the System namespace has 19 different overloaded forms! See the .Net Framework Documentation or MSDN for all of these.

Overloading Constructors

Since constructors are a special type of method, we can overload constructors similarly.

```
    class Person
    {
        private string name;

        public Person()
        {
            name = "uknown";
        }

        public Person(string theName)
        {
            name = theName;
        }
    }
```

Now, if we create an object like

```
    Person thePerson = new Person();
```

the first constructor will be called initializing name with "unknown". If we create an object like

```
    Person thePerson = new Person("Faraz");
```

the second constructor will be called initializing 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(), who would have had to come up with 19 names!)

Value types (out & ref Keywords)

When we pass a variable of an implicit data type to a method, conceptually the runtime generates a copy and passes that copy to the method. It is actually a copy of the variable that is available inside the method. Hence if you modify a value type variable (passed as a parameter) in a method, the actual value of the variable would not be changed outside the method. Let us have in our test class a Main() method and a DoWork() method:

```
class Test
{
    public static void Main()
    {
        int a = 3;
        DoWork(a);
        Console.WriteLine("The value of a is " + a);
    }

    public static void DoWork(int i)
    {
        i++;
    }
}
```

The program will result in

```
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 a local variable in Main(). Hence, they can be accessed within their containing methods only. In fact, we may define int a; in different methods and each will have its own variable a and none would have correspondence with any other implicitly.

C# provides a keyword, ref, which means that the value type will be passed by reference instead of the default by value behavior. Hence, changes done inside the method would be reflected back after the method has been called and terminated. Both the method signature and method calling should be declared as ref in order to override the by value characteristic to by ref.

```
class Test
```

```
{
    public static void Main()
    {
        int a = 3;          // must be initialized
        DoWork(ref a);    // note ref
        Console.WriteLine("The value of a is " + a);
    }

    public static void DoWork(ref int i) // note ref
    {
        i++;
    }
}
```

The program will give the following result:

```
The value of a is 4
```

In the case of the ref keyword, the variable must be initialized before passing it to the method by reference. C# also provides the out keyword. This is used for passing a variable for output purposes. This will again be passed by reference. However, when using the out keyword, it is not necessary to initialize the variable.

```
class Test
{
    public static void Main()
    {
        int a;                // may be left un-initialized
        DoWork(out a);    // note out
        Console.WriteLine("The value of a is " + a);
    }

    public static void DoWork(out int i) // note out
    {
        i=4;
    }
}
```

The program will give the result

```
The value of a is 4
```

Reference types

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

```
class Test
{
    public static void Main()
    {
        int [] nums = { 2, 4, 8 } ;
        DoWork(nums);
        int count =0;
        foreach(int num in nums)
            Console.WriteLine("The value of a[{0}] is {1}", count++, num);
    }

    public static void DoWork(int [] numbers)
    {
        for(int i=0; i<numbers.Length; i++)
            numbers[i]++;
    }
}
```

The program will result in

```
The value of a[0] is 3
The value of a[1] is 5
The value of a[2] is 9
```

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

Some more about references and objects

A reference is just a pointer or handle to the object in memory. It is possible to create an object without giving its handle to any reference:

```
new Student();
```

The above line is a valid statement. It will create an object of Student class without any reference pointing to it. An object is actually eligible to be garbage collected when there is no reference to point it. So, in the above case, the new Student object will instantly be eligible to be garbage collected after its creation. Experienced programmers often call methods on these unreferenced objects like

```
    int pc = ( new Student(87, 94, 79) ).CalculatePercentage();
```

In the above line, a new object of class Studrent is created and the CalculatePercentage() method is called on it. This newly created, unreferenced object will be eligible to be garbage collected just after the method CalculatePercentage() completes its execution. I personally won't encourage you to write such statements. The above line is similar to

```
    Student theStduent = new Student(87, 94, 79);

    int pc = theStudent.CalculatePercentage();

    theStudent = null;
```

We assigned null to theStudent so the object above will be destroyed after method call terminates as in the case of previous example. When you write:

```
    Student student1 = new Student("Faraz");
```

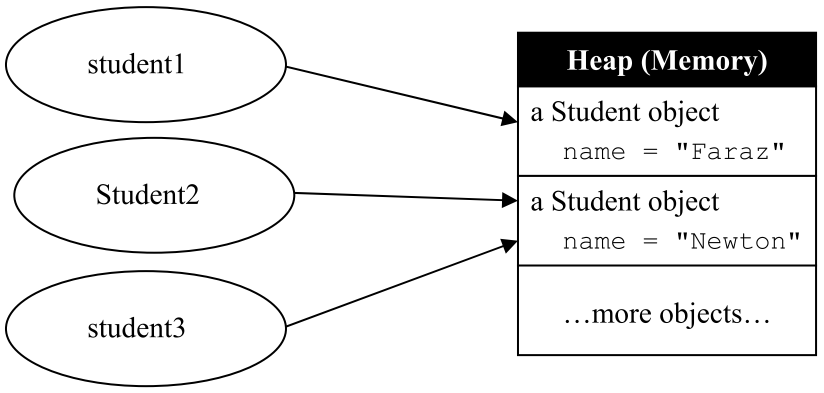
a new object of type Student is created at the heap and its handle is given to the reference student1. Let us make another object and give its handle to the reference student2.

```
    Student student2 = new Student("Newton");
```

Now, if we write

```
    Student student3 = student2;
```

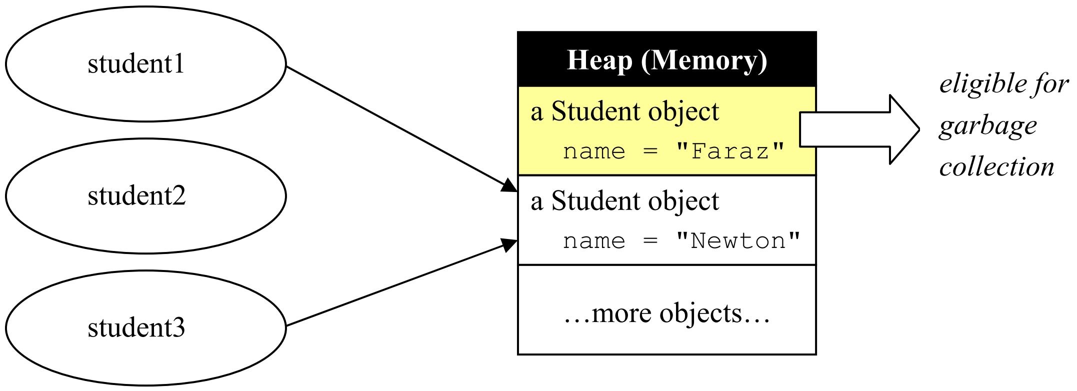
The new reference student3 will also start pointing the student (Newton) already pointed by student2. Hence both student2 and student3 will be pointing to same student and both can make changes to same object.



Now if we write,

```
    student1 = student3;
```

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 anytime.



If you want a reference to reference nothing, you can set it to null, which is a keyword in C#.

```
    student1 = null;
```

4. Inheritance & Polymorphism

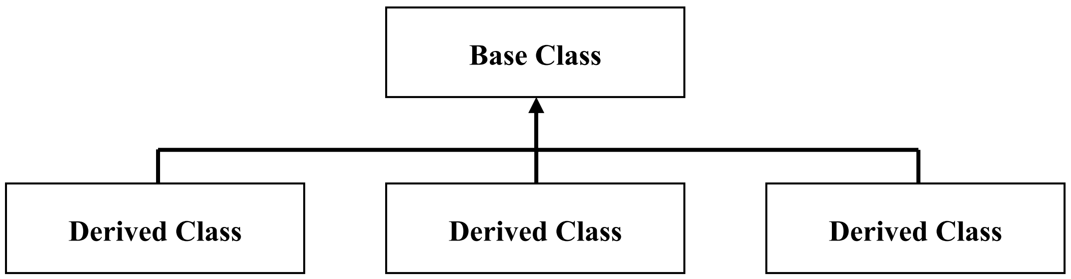
Lesson Plan

Today we will learn about fundamental object oriented features like inheritance and polymorphism in C#. We will start by building on our understanding of inheritance and then we will move towards understanding how C# supports inheritance. We will spend some time exploring the object class and then we will go towards polymorphism and how polymorphism is implemented in C#. We will finish this lesson by understanding how boxing, un-boxing and type-casting mechanisms work in C#.

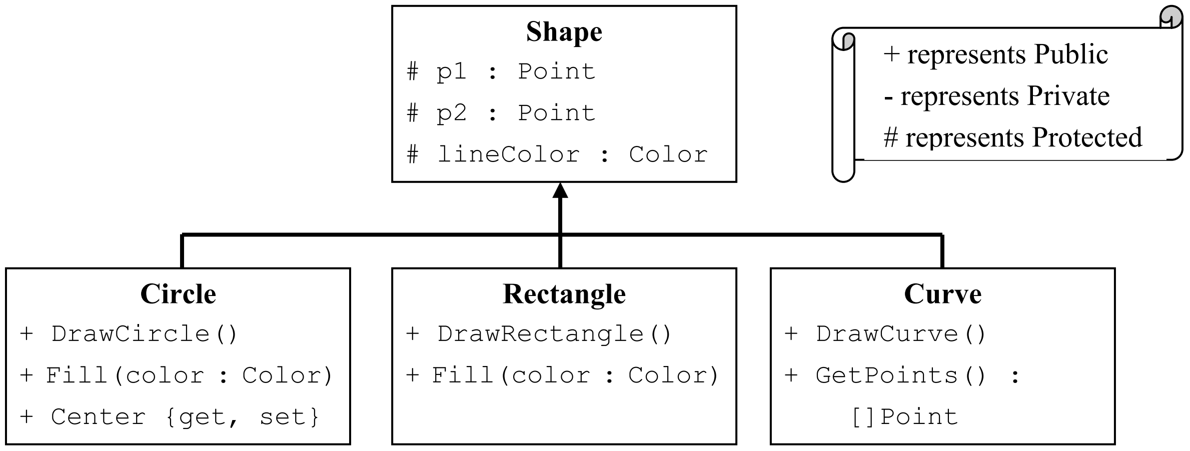
Inheritance

Unless this is your first time at 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. The class that inherits the functionality of the base class and extends it in its own way is called the sub, child, derived or inherited class.



For example:



In the figure above, we have used a UML (Unified Modeling Language) class diagram to show Inheritance. Here, Shape is the base class while Circle, Rectangle and Curve are its sub-classes. A base class usually has general functionality, while sub-classes possess specific functionality. So, when sub-classing or inheriting, we go *'from specialization to generalization'*.

If a class B (sub class) inherits a 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 the private members) of class A. Private members of a base class do get inherited in a 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 you are not getting the idea, don't worry, things will get clearer when we implement inheritance in the following sections)

Inheritance in C#

Before we go on to implementation, here are some key-points about inheritance in C#

- C#, like Java and contrary to C++, allows only single class inheritance. Multiple inheritance of classes is not allowed in C#.
- The Object class defined in the System namespace is implicitly the ultimate base class of all the classes in C# (and the .NET framework)
- Interfaces in C# can inherit more than one interface.

So, multiple inheritance of interfaces is allowed in C# (again similar to Java). We will look at interfaces in detail in the coming lessons. Structures (struct) in C# can only inherit (or implement) interfaces and can not be inherited.

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

Implementing inheritance in C#

C# uses the colon ':' operator to indicate inheritance. Suppose we have a class Student with the fields registrationNumber, name and dateOfBirth, along with the corresponding properties. The class also has a method GetAge() which calculates and returns the age of the student.

```
class Student
{
    // private Fields
    private int registrationNumber;
    private string name;
    private DateTime dateOfBirth;

    // Student Constructor
    public Student()
    {
        Console.WriteLine("New student created. Parameterless constructor called...");
    }

    public Student(int registrationNumber, string name, DateTime dateOfBirth)
    {
        this.registrationNumber = registrationNumber;
        this.name = name;
        this.dateOfBirth = dateOfBirth;
        Console.WriteLine("New Student Created. Parameterized constructor called...");
    }

    // Public Properties
    public int RegisterationNumber
    {
        get { return registrationNumber; }
    }

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public DateTime DateOfBirth
    {
        get { return dateOfBirth; }
        set { dateOfBirth = value; }
    }

    // Public Method
    public int GetAge()
    {
        int age = DateTime.Now.Year - dateOfBirth.Year;
        return age;
    }
}
```

The Student class above is very simple. We have defined three private fields, their accessor properties and one method to calculate the age of the student. We have defined two constructors: the first one takes no parameters and the other takes three parameters. Note that we have only defined the get { } property of registrationNumber since we do not want the user of the Student class to change registrationNumber once it has been assigned through the constructor.

Also note that we did not make the GetAge() a property, but a method. The reason for this is that properties are generally supposed to be accessors for getting/setting the values of fields and not for calculating/processing the data. Hence, it makes sense to declare GetAge() as a method.

Now, let us declare another class named SchoolStudent that inherits the Student class, but with additional members like marks of different subjects and methods for calculating total marks and percentage.

```
class SchoolStudent : Student
{
    // Private Fields
    private int totalMarks;
    private int totalObtainedMarks;
    private double percentage;

    // Public Constructors
    public SchoolStudent()
    {
        Console.WriteLine("New school student created. Parameterless constructor called...");
    }

    public SchoolStudent(int regNum, string name, DateTime dob, int totalMarks, int totalObtainedMarks)
        : base(regNum, name, dob)
    {
        this.totalMarks = totalMarks;
        this.totalObtainedMarks = totalObtainedMarks;
        Console.WriteLine("New school student created. Parameterized  constructor called...");
    }

    // Public Properties
    public int TotalMarks
    {
        get { return totalMarks; }
        set { totalMarks = value; }
    }

    public int TotalObtainedMarks
    {
        get { return totalObtainedMarks; }
        set { totalObtainedMarks = value; }
    }

    // Public Method
    public double GetPercentage()
    {
        percentage = (double) totalObtainedMarks / totalMarks * 100;
        return percentage;
    }
}
```

The SchoolStudent class inherits the Student class by using the colon operator.

```
class SchoolStudent : Student
```

The SchoolStudent class inherits all the members of the Student class. In addition, it also declares its own members: three private fields (totalMarks, totalObtainedMarks and percentage) with their corresponding properties, two constructors (one without parameters and one with parameters) and one instance method (GetPercentage()). For now, forget about the second (parameterized) constructor of our SchoolStudent class. Let us write our Test class and Main() method.

```
// Program to Demonstrate Inheritance
class Test
{
    static void Main(string [] args)
    {
        Student st = new Student(1, "Fraz", new DateTime(1980, 12, 19));
        Console.WriteLine("Age of student, {0}, is {1}\n", st.Name, st.GetAge());
        SchoolStudent 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());
    }
}
```

In the Main() method, first we made an object of the Student class (st) and printed the name and age of student st, then we made an object of the SchoolStudent class (schStd). Since we used a parameterless 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 the 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 23
New student created. Parameterless constructor called...
New school student created. Parameterless constructor called...
Age of student, Newton, is 22. Newton got 95.2% marks.
Press any key to continue
```

The output of the first two lines is as expected, but see the output when we created the SchoolStudent object. First the parameterless constructor of Student is called and then the constructor of SchoolStudent is called.

```
New student created. Parameterless constructor called...
New school student created. Parameterless constructor called...
```

Constructor calls in Inheritance

When we instantiate the sub-class (SchoolStudent), the compiler first instantiates the base-class (Student) by calling one of its constructors and then calling 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 base keyword in SchoolStudent's second construction declaration and make other changes like:

```
public SchoolStudent(int regNum, string name, DateTime dob,  int totalMarks, int totalObtainedMarks)
//    : base(regNum, name, dob)
```

```
{
    this.Name = name;
    this.DateOfBirth = dob;
    this.totalMarks = totalMarks;
    this.totalObtainedMarks = totalObtainedMarks;
    Console.WriteLine("New school student created. Parameterized constructor called...");
}
```

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

```
static void Main(string [] args)
{
    SchoolStudent schStd = 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());
}
```

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

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

Consider the constructor calls shown in the output; first the parameterless constructor of the base-class (Student) is called and then the parameterized constructor of the sub-class (SchoolStudent) class is called. It shows that the compiler creates an object of the base class before it instantiates the sub-class. Now, let us comment out the parameterless constructor of the Student class.

```
class Student
{
    ...
    /*public Student()
    {
        Console.WriteLine("New student created. Parameterless constructor called...");
    }*/
    ...
}
```

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

```
No overload for method 'Student' takes '0' arguments
```

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

The base keyword - Calling Constructors of the base-class explicitly

We can explicitly call the constructor of a base-class using the keyword base. base must be used with the constructor header (or signature or declaration) after a colon ':' operator like:

```
class SubClass : BaseClass
{
    SubClass(int id) : base() // explicit constructor call
    {
        // some code goes here
    }
}
```

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

```
class SubClass : BaseClass
{
    SubClass(int id) : base(id)
    {
        // some code goes here
    }
}
```

Now, the constructor of SubClass(int) will explicitly call the constructor of the base-class (BaseClass) that takes an int argument. Let us practice with our SchoolStudent class again. Revoke the changes we made to its second constructor and make it look like:

```
public SchoolStudent(int regNum, string name, DateTime dob, int totalMarks, int totalObtainedMarks)
               : base(regNum, name, dob)
{
    this.totalMarks = totalMarks;
    this.totalObtainedMarks = totalObtainedMarks;
    Console.WriteLine("New school student created. Parameterized constructor called...");
}
```

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

```
public Student(int registrationNumber, string name, DateTime dateOfBirth)
{
    this.registrationNumber = registrationNumber;
    this.name = name;
    this.dateOfBirth = dateOfBirth;
    Console.WriteLine("New Student Created. Parameterized constructor called...");
}
```

This is also important, as the field registrationNumber is private and provides only the get {} public property. So, in no way can we assign registrationNumber of a student, except when calling a constructor. base fits best in this scenario. The Main() method would be like:

```
static void Main(string [] args)
{
    SchoolStudent schStd = 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());
}
```

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

```
New Student Created. Parameterized constructor called...
New school student 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 Student (base-class) is called and then the parameterized constructor of SchoolStudent (sub-class) is called, setting the appropriate values, which is exactly what we wanted to do. It then simply prints the name, age and percentage of SchoolStudent.
Note that although we write base 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: C#'s base keyword is similar to Java's super keyword, but C# has adopted the syntax of base from C++ and it is identical to C++'s base keyword.

Protected Access Modifier

In our Student class, registrationNumber is made private and has only the get {} property so that no user of the Student class can modify the registrationNumber outside the class. It might be possible that we need to modify this field (registrationNumber) in our SchoolStudent class, but we can not do this. C# provides the protected access modifier just for this. protected members of the 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 would complain and generate an error.

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

```
class A
    {
        protected void DoWork()
        {
            Console.WriteLine("DoWork called...");
        }
    }
```

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

```
class Test
    {
        static void Main(string [] args)
        {
            A a = new A();
            a.DoWork();
        }
    }
```

When we try to compile it, the compiler says:

```
'CSharpSchool.A.DoWork()' is inaccessible due to its protection level
```

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

```
    class B : A
    {
        public B()
        {
            DoWork();
        }
    }
```

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

```
static void Main(string [] args)
{
    B b = new B();
}
```

The result will be:

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

It shows that we can access protected members of a class inside its sub-classes. Note that it is still wrong to write

```
static void Main(string [] args)
{
    B b = new B();
    b.DoWork();     // error
}
```

We can not access protected members even with the reference of the sub-class.

The Protected internal Access Modifier

In a similar way, the protected internal 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 the containing class.

Hence, protected internal acts like 'protected OR internal', i.e., either protected or internal.

The sealed keyword

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

```
sealed class A
{
    ...
    }
```

If one tries to inherit another class B with class A

```
    class B : A
    {
        ...
    }
```

The compiler will generate the following error:

```
'CSharpSchool.B' : cannot inherit from sealed class 'CSharpSchool.A'
```

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

Object class - the base of all classes

In C# (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 a reference of type Object can hold any type of object. The Object class has following methods:

Method Name	Description
Equals(Object)	Compares two objects for equality. The default implementation only supports reference equality, that is, it 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.
Static Equals(Object, Object)	Same as above except, this method is static.
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.
static ReferenceEquals (Object,Object)	Returns true if both the references passed point 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 a shallow copy of the current object. A shallow copy contains a copy of all the instance fields (state) of the current objects.

C# also provides an object keyword which maps to the System.Object class in the .NET framework class library (FCL).

Polymorphism

It is said that a programmer goes through three stages when learning an object oriented programming language. e first phase is when a programmer uses non-object oriented constructs (like for, if...else, switch...case) of the object oriented programming language. The second phase is when a programmer writes classes, inherits them and creates their objects. 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 the base type 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. Let a class A be inherited by a class B:

```
class A
{
    public void MethodA()
    {
        ...
    }
}

class B : A
{
    public void MethodB()
    {
        ...
    }
}
```

Then it is legal to write:

```
A a = new B();
```

Here a reference of type A is holding an 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:

```
    a.MethodA();
```

But it is incorrect to write:

```
    a.MethodB();        // error
```

Although we have an object of type B (contained in the reference of type A), we can not access any member of type B since the apparent type here is A and not B.

Using methods with the same name in the Base and the Sub-class

In our Shape class let us define a method Draw() that draws a shape on the screen:

```
    class Shape
    {
        public void Draw()
        {
            Console.WriteLine("Drawing Shape...");
        }
    }
```

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

```
    class Circle : Shape
    {
        public void Draw()
        {
            Console.WriteLine("Drawing Circle...");
        }
    }
```

Here, we have the Draw() method with the same signature in both the Shape and the Circle classes. Now, if in our Main() method, we write

```
    static void Main()
    {
        Circle theCircle = new Circle();
        theCircle.Draw();
    }
```

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 hides the inherited Draw() method of Shape class. If in our Main() method we write

```
    Shape theShape = new Circle();
    theShape.Draw();
```

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

```
Drawing Shape...
```

Overriding the methods - virtual and override keywords

Now, 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 virtual and the Draw() method in the Circle (sub) class as override.

```
    class Shape
    {
        public virtual void Draw()
        {
            Console.WriteLine("Drawing Shape...");
        }
    }

    class Circle : Shape
    {
        public override void Draw()
        {
            Console.WriteLine("Drawing Circle...");
        }
    }
```

Now, in our Main() method we write

```
static void Main()
    {
        Shape theShape = new Circle();
        theShape.Draw();
    }
```

Here we have used the reference of the base type (Shape) to refer to an object of the sub type (Circle) and call the Draw() method through it. As we have overridden the Draw() method of Shape in the Circle class and since the Draw() method is marked virtual 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 compiler uses the actual object type referenced by the base type reference to decide which of the methods to call. When we compile the above program, the result will be:

```
Drawing Circle...
```

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

We mark that method in the base class on which we want to achieve polymorphism as virtual. By marking a method virtual, we allow a method to be overridden and be used polymorphically. In the same way, we mark the method in the sub-class as override when it is overriding the virtual method in the base class. By marking the method as override, we announce that we are deliberately overriding the corresponding virtual method in the base class.

Author's Note: All the methods in C# are non-virtual by default unlike Java (where all the methods are implicitly virtual). We have to explicitly mark a method as virtual (like C++). Unlike C++ and Java, we also have to declare the overriding method in the sub-class as override in order to avoid any unconscious overriding. In fact, C# also introduces the new 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 the previous line which object the reference (theShape) holds.

```
        Shape theShape = new Circle();
        theShape.Draw();
```

Yes, it is possible for the compiler to conclude it in this particular case, but 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 inherit the Shape class and override its Draw() method to have their own specific implementations.

```
    class Rectangle : Shape
    {
        public override void Draw()
        {
            Console.WriteLine("Drawing Rectangle...");
        }
    }

    class Curve : Shape
    {
        public override void Draw()
        {
            Console.WriteLine("Drawing Curve...");
        }
    }
```

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

```
    static void Main()
    {
        Shape [] shapes = {new Circle(), new Rectangle(), new Curve()};
        Random random = new Random();
        for(int i=0; i<5; i++)
        {
            int randNum = random.Next(0, 3);
            shapes[randNum].Draw();
        }
    }
```

Here, we have made an array of type Shape and stored an object of the Circle, Rectangle and Curve classes in it. Next, we used the Random class to generate a random number between 0 and 2 (both 0 and 2 inclusive). We use this randomly generated number as an index in the shapes array to call the Draw() method. Now, 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 would be called at runtime in each iteration of the loop. When we compile and run the above program, we will see a different output with each run. For example, when I execute the above program, I get the following output:

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

The new keyword

Suppose, in our Circle class, we don't want to override the Draw() method, but we need to have a Draw() method, what we can do? In Java, we can't do this. In C++, even if we mark the method in the base class as virtual, it is impossible. But, C# introduces a keyword new to mark a method as a non-overriding method and as the one which we don't want to use polymorphically. Let us define the Shape and Circle classes as:

```
class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Drawing Shape...");
    }
}

class Circle : Shape
{
    public new void Draw()
    {
        Console.WriteLine("Drawing Circle...");
    }
}
```

Note that we marked the Draw() method in Circle with the new keyword to avoid polymorphism. In our method we write

```
Shape theShape = new Circle();
theShape.Draw();
```

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

```
Drawing Shape...
```

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

```
'CSharpSchool.Circle.Draw()' hides inherited member
'CSharpSchool.Shape.Draw()'.
```

To make the current member override that implementation, add the override keyword. Otherwise add the new keyword.

Type casting the objects - Up-casting and Down-casting

Type Casting means 'making an object behave like' or 'changing the apparent type of an object'. In C#, you can cast an object in an inheritance hierarchy either from the bottom towards the top (up-casting) or from the top towards the bottom (down-casting).

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

```
    Parent theParent = new Child();
```

On the contrary, down-casting is un-safe and explicit. By un-safe, we mean it may throw an exception – that is, fail. Consider the following line of code:

```
    Shape [] shapes = { new Circle(), new Rectangle(), new Curve() };
```

Where Circle, Rectangle and Curve are sub-classes of Shape class. Now, 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 as

```
    Rectangle rect = (Rectangle) shapes[1];
```

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.

Also, note that down-casting can be unsuccessful and if we attempt to write

```
    Rectangle rect = (Rectangle) shapes[2];
```

Since shapes[2] contains an object of type Curve which can not be casted to the Rectangle type, the CLR would raise the following exception at run-time:

```
System.InvalidCastException: Specified cast is not valid.
```

The is and as keywords

To check the run-time type of an object, you can use either the is or the as keyword. is compares the type of the object with the given type and returns true if it is cast-able; otherwise, it returns false. For example,

```
    Console.WriteLine(shapes[1] is Rectangle);
```

would print true on the Console Window, while

```
    Console.WriteLine(shapes[2] is Rectangle);
```

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

```
    Shape [] shapes = { new Circle(), new Rectangle(), new Curve() };
    Rectangle rect=null;
    if(shapes[1] is Rectangle)
    {
        rect = (Rectangle) shapes[1];
    }
```

Alternatively, we can also use the as operator to check the run-time type of the object. The as operator returns null if the object is not cast-able otherwise, it casts the object to the specified type as

```
    Shape [] shapes = { new Circle(), new Rectangle(), new Curve() };
    Rectangle rect = shapes[1] as Rectangle;
    if(rect != null)
        Console.WriteLine("Cast successful");
    else
        Console.WriteLine("Cast unsuccessful");
```

Although is and as perform similar functionality, is just checks the runtime type while as in addition to this, also casts the object to the specified type.

Boxing and Un-boxing

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

```
    int i = 5;
```

when we write

```
    i.ToString();
```

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

```
    int i = 5;
    Object obj = i;    // implicit boxing
    obj.ToString();
```

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

```
    Object obj = (Object) i;    // unnecessary explicit boxing
```

On the other hand, un-boxing is explicit conversion from object type to value type. The following lines show how un-boxing is implemented in C#

```
    int i = 5;
    Object obj = i;        // implicit boxing
    int j = (int) obj;    // explicit un-boxing
```

Like down-casting, un-boxing can be un-safe and throw an 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 are the transformations between value type and object, while casting just transforms the apparent (reference) type of objects.

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

5. Structures, Enumeration, Garbage Collection & Nested Classes

Lesson Plan

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

Structures (struct)

Structures, denoted in C# by the struct keyword, are lightweight objects. Structures are very similar to classes in C#, but with the following properties:

- A struct is useful for creating types that are used to hold data like Point, Rectangle, Color types.
- A struct is of value type, contrary to classes which are of reference type. This means that structures are allocated on the stack and passed to methods by value, that is, by making their copies.
- A struct may contain constructors (except for the no-argument constructor), fields, methods and properties just like in classes.
- Like all value types, structs can neither inherit another class, nor can they be inherited.
- A struct can implement interfaces.
- Like every other type in C#, a struct is also implicitly inherited from the System.Object class.
- Instances of a struct can be created with and without using the new keyword.

Most of the .Net framework types like System.Int32 (for int), System.Double (for double), System.Boolean (for bool), System.Byte (for byte),... are implemented as a struct. When kept small in size, a struct is more efficiently used by the system than a class.

Defining a struct

A struct is defined just like a class, using the struct keyword. Suppose, in a drawing application, we need a Point data type. Since this is a simple and lightweight type, we implement it as a struct.

```
struct Point
{
    public double x;
    public double y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public override string ToString()
    {
        return "(" + x + ", " + y + ")";
    }
}
```

Here, we have declared a struct named Point. Point contains two public fields, x and y, that represent the location of a Point in the coordinate system. We provide a public constructor to initialize the location of the point and we also override the ToString() method from the Object class, so that our point can be printed easily using the Console.WriteLine() method in the Main() method.

Instantiating the struct

A struct 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 we mentioned earlier, we can not provide the no-argument constructor in a struct and if we try to do so, the compiler will generate an error. The compiler implicitly provides a default no-argument constructor for each struct which initializes the fields of a struct with their default values. In the following Main() method, we instantiate Point using the above mentioned three ways

```
    class Test
    {
        static void Main()
        {
            Point pt = new Point();
            Point pt1 = new Point(15, 20);
            Point pt2;    // instantiation without using the new keyword
            pt2.x = 6;
            pt2.y = 3;

            Console.WriteLine("pt = {0}", pt);
            Console.WriteLine("pt1 = {0}", pt1);
            Console.WriteLine("pt2 = {0}", pt2);
        }
    }
```

The output of the program is:

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

Here, we have instantiated three Point objects. The first one (referenced by pt) is instantiated using new and a default no-argument constructor (implemented by the compiler) which zeroed all the fields. Thus, the first Point (pt) is printed as (0, 0). The second one (referenced by pt1) is instantiated using the new keyword and a custom (double, double) constructor. The point (pt1) is initialized with the specified value and thus printed as (15, 20) on the console. The third object (referenced by pt2) is 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 struct created without the new keyword, all its fields must be explicitly initialized. Hence, the Point (pt2) printed out as (6, 3). Note that we wrote:

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

instead of:

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

Console.WriteLine() expects a string, but since we have overridden the ToString() method in our Point struct, the compiler will implicitly call the ToString() method when it expects a string in the

```
Console.WriteLine() method.
```

Let us play with our program to increase our understanding of a struct. If we don't initialize any of the fields of Point then:

```
        static void Main()
        {
            Point pt2;
            pt2.x = 6;
            // pt2.y = 3;        // line 1

            Console.WriteLine("pt2 = {0}", pt2);
        }
```

The compiler will generate an error on 'line 1'

```
Use of unassigned local variable 'pt2'
```

Now, let us make the fields of the point private and provide public properties to access them.

```
    struct Point
    {
        private double x;
        private double y;

        public Point(int x, int y)
        {
            this.x = x;
            this.y = y;
        }

        public double X
        {
            get { return x; }
            set { x = value; }
        }

        public double Y
        {
            get { return y; }
            set { y = value; }
        }

        public override string ToString()
        {
            return "(" + x + ", " + y + ")";
        }
    }
```

Now, let us try to create an instance of the Point without using the new keyword

```
    static void Main()
    {
        Point pt2;
        pt2.X = 6;
        pt2.Y = 3;

        Console.WriteLine("pt2 = {0}", pt2);
    }
```

Here, we have created an instance of Point (pt2), initialized its fields through their accessor properties and then attempted to use it in a Console.WriteLine() method. When we try to compile the program, the compiler generates an error:

```
Use of unassigned local variable 'pt2'
```

We did initialize the fields through properties, but why is the compiler still complaining? In fact, when instantiating a struct without the new keyword, we must first initialize its fields explicitly, without using any properties or methods. This means that you can't instantiate and use a struct without the new keyword unless all its fields are public (or accessible to you) and explicitly initialized.

Finally, let us try to define a no-argument constructor in a struct

```
    struct Point
    {
        public double x;
        public double y;

        public Point()
        {
            x = 3;
            y = 2;
        }

        public Point(int x, int y)
        {
            this.x = x;
            this.y = y;
        }
    }
```

When we try to compile the above program, the compiler says:

```
Structs cannot contain explicit parameterless constructors
```

Here, the compiler says that it is illegal to define parameterless (no-argument) constructors in a struct.

Author's Note: struct is not a new concept. In C++, the struct is another way of defining a class (why?). Even though most of the time C++ developers use struct for lightweight objects holding just data, the C++ compiler does not impose this. C# structs are more restricted than classes because they are value types and can not take part in inheritance. Java does not provide struct at all.

structs as Value Types

We will now demonstrate the use of struct as a value-type through a simple program which passes a struct to some method that modifies it. We will use the Point struct that we defined earlier. The Test class containing the Main() method is:

```
    class Test
    {
        static void Main()
        {
            Point pt = new Point(19, 12);
            Console.WriteLine("Before calling ChangeIt(pt): pt    = {0}", pt);
            ChangeIt(pt);
            Console.WriteLine("After  calling ChangeIt(pt): pt    = {0}", pt);
        }
        static void ChangeIt(Point point)
        {
            point.x = 6;
            point.y = 11;
            Console.WriteLine("In the method  ChangeIt(pt): point = {0}", point);
        }
    }
```

We initialized the Point (pt) with (19, 12) and called the method ChangeIt(), which modifies the location of the Point. We then printed the Point before, after and inside the ChangeIt() method. The output of the program is:

```
Before calling ChangeIt(pt): pt = (19, 12)
In the method ChangeIt(pt): point = (6, 11)
```

```
After calling ChangeIt(pt): pt = (19, 12)
Press any key to continue
```

We can clearly see from the output that the value of Point did get changed inside the method (ChangeIt()), but this change did not reflect back on to the original struct, which printed it un-changed in the Main() method after the ChangeIt() method. This supports the fact that structs are treated as value types by C#.

Enumeration

Enumeration is a very good concept originally found in C++, but not in Java. C# supports enumerations using the enum keyword. Enumerations, 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 here while its contents are called enumerator identifiers. Enumerations are of integral types like int, short, byte,… (except the char type).

The Need for Enumeration

Before we go into the details of how to define and use enumerations, let us first consider why we need enumerations in the first place. Suppose we are developing a Windows Explorer type of program that allows the users to do different tasks with the underlying file system. In View Properties, we allow the 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 void Sort(string criteria)
    {
        switch(criteria)
        {
            case "by name":
                // code to sort by name
                break;
            case "by type":
                // code to sort by type
                break;
            case "by size":
                // code to sort by size
                break;
            case "by date":
                // code to sort by modification date
                break;
            default:
                throw new Exception("Invalid criteria for sorting passed...");
        }
    }
```

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

```
    Explorer explorer = new Explorer();
    // some code goes here
    explorer.Sort("by name");
```

Here 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 the switch…case block and will take appropriate action (sort items by name). But, if someone writes:

```
    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 the coming lessons. Right now, just consider that the 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. Enumeration provides the solution for this type of a problem.

Using Enumeration (enum)

Enumerations are defined in C# using the enum keyword. The enumerator identifiers (named constants) are separated using a comma ','.

```
    enum SortCriteria
    {
        ByName,
        ByType,
        BySize,
        ByDate
    }
```

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

```
    public void Sort(SortCriteria criteria)
    {
        switch(criteria)
        {
            case SortCriteria.ByName:
                // code to sort by name
                Console.WriteLine("Files/Folders sorted by name");
                break;
            case SortCriteria.ByType:
                // code to sort by type
                Console.WriteLine("Files/Folders sorted by type");
                break;
            case SortCriteria.BySize:
                // code to sort by size
                Console.WriteLine("Files/Folders sorted by size");
                break;
            case SortCriteria.ByDate:
                // code to sort by modification date
                Console.WriteLine("Files/Folders sorted by modification date");
                break;
        }
    }
```

Note that we did not have a default clause in the switch…case, because now it is impossible to pass un-allowed criteria either by mistake or intentionally as the compiler will check the criteria at compile time. Now, in order to call the Sort() method, we need to pass an instance of an enumeration of type SortCriteria, which can only take input from of the allowed four criterion, hence providing a compile time check for illegal criteria. We now call the Sort() method as:

```
    Explorer explorer = new Explorer();
    explorer.Sort(SortCriteria.BySize);
```

Alternatively, we can create an instance of SortCriteria and then pass it as an argument to Sort() as:

```
    Explorer explorer = new Explorer();
    SortCriteria criteria = SortCriteria.BySize;
    explorer.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 like days of the week (Sunday, Monday,…), name of the months (January, February,…), connection type (TCPIP, UDP), file open mode (ReadOnly, WriteOnly, ReadWrite, Append), etc.

More about Enumerations

Enumerations internally use integral values to represent the different named constants. The underlying type of an enumeration can be any integral type (e.g. int, byte, etc). The default type is int. 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 an integral value by applying an appropriate cast.

```
    static void Main()
    {
        SortCriteria criteria = SortCriteria.ByType;
        int i = (int) criteria;
        Console.WriteLine("the integral value of {0} is {1}", criteria, i);
}
```

Here, we created an instance of SortCriteria (named criteria), type-casted it to int and then printed it using the Console.WriteLine() method. The output is:

```
the integral value of ByType is 1
Press any key to continue
```

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

```
    enum Temperature
    {
        BoilingPoint = 100,
        FreezingPoint = 0
    }
```

Now, when we try to 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).

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

Will display:

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

Consider the following enumeration

```
    enum Days : byte
    {
        Monday = 1,
        Tuesday,
        Wednesday,
        Thursday,
        Friday,
        Saturday,
        Sunday
    }
```

Here 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 a byte instead of the default int type. Now, the compiler will use the byte type to represent days internally.

```
    enum Days : byte
    {
        ...
    }
```

In the Main() method, we can print the integral value of days as:

```
    static void Main()
    {
        int i = (byte) Days.Tuesday;
        Console.WriteLine("the integral value of {0} is {1}", Days.Tuesday, i);
    }
```

The output is:

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

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

Garbage Collection in .Net

Memory management is no longer the responsibility of a programmer when writing managed code in the .Net environment. .Net provides its own Garbage Collector to manage the memory. The Garbage Collector is a program that runs in the background as a low-priority thread and keeps track of the un-referenced objects (objects that are no longer referenced by any reference) by marking them as 'dirty objects'. The Garbage collector is invoked by the .Net Runtime at regular intervals and removes the dirty objects from memory.
Before re-claiming the memory of any object (before removing the object from memory), the garbage collector calls the Finalize() method or destructor (discussed in lesson 4) of the object, so as to allow an object to free its resources. Hence, you should provide the destructor (which internally overrides 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's memory, we never know exactly when it will be called. It may be called instantly, after an object is found to have no references to it, or 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, I will give an example of implementing the IDisposable interface, as you will have to use this method quite often in your programs. I recommend that you come back and read this again after you become familiar with interfaces.

```
using System;

namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            GarbageCollection garCol = new GarbageCollection();
            garCol.DoSomething();
            garCol.Dispose();
        }
    }

    class GarbageCollection : IDisposable
    {
        public void DoSomething()
        {
            Console.WriteLine("Performing usual tasks...");
        }

        public void Dispose()
        {
            GC.SuppressFinalize(this);
            Console.WriteLine("Disposing object...");
            Console.WriteLine("Freeing resources captured by this object...");
        }

        ~GarbageCollection()
        {
            Console.WriteLine("Destructing object...");
            Console.WriteLine("Freeing resources captured by this object...");
        }
    }

}
```

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 GrabageCollection class can decide when to free-up the resources held by the object, by calling the Dispose() method. The common usage of the class is demonstrated in the Test class' Main() method. Note that we called the SuppressFinalize() method of the System.GC class in the Dispose() method.

```
    public void Dispose()
    {
        GC.SuppressFinalize(this);
        Console.WriteLine("Disposing object...");
        Console.WriteLine("Freeing resources captured by this object...");
    }
```

The System.GC class can be used to control the Garbage Collector. If we pass the current object in the GC.SuppressFinalize() method, the Finalize() method (or destructor) 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.

```
    static void Main()
    {
        GarbageCollection garCol = new GarbageCollection();
        garCol.DoSomething();
        //    garCol.Dispose();
    }
```

The runtime would call the Destructor (or Finalize() method) when the object is no longer referenced.

Now the output would be like this:

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

Destructors and Performance Overhead

Having a destructor or Finalize() method in your class creates some performance overhead, therefore, it is recommended that you do not declare these in your classes unless your class is holding un-managed resources like file handles, databases or network connections.

System.GC.Collect() method

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

Nested Classes in C#

So far, we have only seen top-level classes (classes only bound by namespace) in our C# programs. We can also define nested classes in C#. Nested classes (also called inner classes) are defined inside another class, like this:

```
using System;

namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            Outer.Inner inner = new Outer.Inner();
            inner.InnerFun();
        }
    }
    class Outer
    {
        private static string name="Faraz";

        public Outer()
        {
            Console.WriteLine("Constructor of Outer called...");
        }

        public class Inner
        {
            public Inner()
            {
                Console.WriteLine("Constructor of Inner called...");
            }

            public void InnerFun()
            {
                Console.WriteLine("InnerFun() called...");
                Console.WriteLine("Name in Outer is {0}", name);
            }
        }
    }

}
```

Here, we have defined a nested class named Inner, contained inside the top-level class Outer. Both the Outer class and the Inner class have no-argument constructors. While the top-level classes can only be marked as public or internal, nested classes can be marked with all the access modifiers: private, public, protected, internal, internal OR protected, with the same meaning as described in lesson 5. Like other class members, the default access modifier for nested classes is private, which means that the nested class can not be referenced and accessed outside the containing class. In order to reference and instantiate the Inner class in the Main() method of the Test class, we mark 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:

```
static void Main()
    {
        Outer.Inner inner = new Outer.Inner();
        inner.InnerFun();
    }
```

Another important point about nested classes is that they can access all the (private, internal, protected, public) static 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 void InnerFun()
    {
        Console.WriteLine("InnerFun() called...");
        Console.WriteLine("Name in Outer is {0}", name);
    }
```

When we compile and run the above program, 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 C# are similar to Java's static inner classes, but there is no concept like Java's non-static inner classes in C#.

Since the 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 internal and the access level of the nested class is public, the nested class would only be accessible in the current assembly (project).

6. Abstract Classes & Interfaces

Lesson Plan

Today we will explore abstract classes and interfaces. We will explore the idea behind abstract methods, abstract classes, interfaces and how they are implemented in C#. Later we will see how to cast to and from interface references by using the is and as operators.

Abstract Classes

Abstract classes can be simply defined as incomplete classes. Abstract classes contain one or more incomplete methods called abstract methods. The abstract class only provides the signature or declaration of the abstract methods and leaves the implementation of these methods to derived or sub-classes. Abstract methods and abstract classes are marked with the abstract keyword. An abstract class itself must be marked with the abstract 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 sealed. A class inheriting an abstract class must implement all the abstract methods in the abstract class, or it too must be declared as an abstract class.

A class inheriting an abstract class and implementing all its abstract methods is called the concrete class of the abstract class. We can declare a reference of the type of abstract class and it can point to the objects of the classes that have inherited the abstract class. Let us declare an abstract class with two concrete properties and an incomplete (abstract) method.

```
abstract class TaxCalculator
{
    protected double itemPrice;
    protected double tax;

    public abstract double CalculateTax();

    public double Tax
    {
        get { return tax; }
    }

    public double ItemPrice
    {
        get { return itemPrice; }
    }
}
```

The TaxCalculator class contains two fields: itemPrice and applied tax. It contains an abstract method CalculateTax() which calculates the tax applied on the itemPrice and stores it in the field tax. The CalculateTax() method 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 used to access the two private fields. If we try to instantiate this abstract class in the Main() method

```
static void Main()
    {
        TaxCalculator taxCalc = new TaxCalculator();
    }
```

The compiler will complain:

```
Cannot create an instance of the abstract class or interface
'CSharpSchool.TaxCalculator'
```

In order to create an instance of the TaxCalculator class, we need to sub-class it. Let us now inherit a class from the abstract TaxCalculator class calling it SalesTaxCalculator.

```
    class SalesTaxCalculator : TaxCalculator
    {
        public SalesTaxCalculator(double itemPrice)
        {
            this.itemPrice = itemPrice;
        }

        public override double CalculateTax()
        {
            tax = 0.3 * itemPrice;
            return itemPrice + tax;
        }
    }
```

The SalesTaxCalculator class inherits the TaxCalculator and overrides its CalculateTax() method. It applies 30% tax on the price of an item (a bit harsh!) and returns the new price of the item. The SalesTaxCalculator class also defines a constructor that takes itemPrice as its parameter. If we don't provide the implementation of the CalculateTax() method in SalesTaxCalculator

```
    class SalesTaxCalculator : TaxCalculator
    {
        public SalesTaxCalculator(double itemPrice)
        {
            this.itemPrice = itemPrice;
        }

    /*  public override double CalculateTax()
        {
            tax = 0.3 * itemPrice;
            return itemPrice + tax;
        }*/
    }
```

We will get a compile time error:

```
'CSharpSchool.SalesTaxCalculator' does not implement inherited abstract member
'CSharpSchool.TaxCalculator.CalculateTax()'
```

Now, 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.

```
class Test
{
    static void Main()
    {
        SalesTaxCalculator salesTaxCalc = new SalesTaxCalculator(225);
        double newPrice = salesTaxCalc.CalculateTax();
         Console.WriteLine("The item price has changed  because of sales tax from {0} $ to {1} $",
            salesTaxCalc.ItemPrice, newPrice);
        Console.WriteLine("Tax applied = {0} $", salesTaxCalc.Tax);
    }
}
```

Here, we have instantiated the SalesTaxCalculator class just like a regular class and accessed its members. The output of the above program will be:

```
The item price has changed because of sales tax from 225 $ to 292.5 $
Tax applied = 67.5 $
Press any key to continue
```

We can also use an abstract class type (TaxCalculator) reference to handle an object of its concrete class (SalesTaxCalculator) in our Main() method.

```
static void Main()
{
    TaxCalculator taxCalc = new SalesTaxCalculator(225);
    double newPrice = taxCalc.CalculateTax();
    Console.WriteLine("The item price has changed because of sales tax from {0} $ to {1} $",
      taxCalc.ItemPrice, newPrice);
    Console.WriteLine("Tax applied = {0} $", taxCalc.Tax);
}
```

We can derive as many concrete classes as we want from the abstract TaxCalculator class, as long as they provide the definition of its abstract methods. Here is another concrete class (WarSurchargeCalculator) of the abstract TaxCalculator class.

```
class WarSurchargeCalculator : TaxCalculator
{
    public WarSurchargeCalculator(double itemPrice)
    {
        this.itemPrice = itemPrice;
    }

    public override double CalculateTax()
    {
        tax = 0.5 * itemPrice;
        return itemPrice + tax;
    }
}
```

The WarSurchargeCalculator can be used similarly in the Main() method.

Interfaces

Interfaces are a special kind of type in C#, used to define the specifications (in terms of method signatures) that should be followed by its sub-types.
An interface is declared using the interface keyword. Interfaces, like abstract classes, can not be instantiated. An interface can contain a signature of the methods, properties and indexers. An interface is a type whose members are all public and abstract by default.

An interface is implemented by a class. A class implementing the interface must provide the body for all the members of the interface. To implement an interface, a class uses the same syntax that is used for inheritance. A colon : is used to show that a class is implementing a particular interface.

A class can implement more than one interface, contrary to class-inheritance where you can inherit only 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 C#, to prefix the name of interfaces with uppercase 'I' like IDisposable, ISerializable, ICloneable, IEnumerator, etc.

Let us define an interface named IWindow as:

```
interface IWindow
{
    Point Position
    {
        get; set;
    }

    string Title
    {
        get; set;
    }

    void Draw();
    void Minimize();
}
```

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 to Minimize() the window. Note the semicolon after each member declaration to show their incomplete nature. Now, all the classes that wish to implement (or realize) this interface must provide the body for these members. The property Position is of type Point, which is a struct type and defined in the program as:

```
struct Point
{
    public int x;
    public int y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

It is important to understand that an interface declares an abstract type and provides the specification which must be followed by all the implementing classes. The compiler enforces this specification and does not compile any concrete class which inherits the interface, but does not implement (provide the body of) all the members of the interface.

Now let us define a RectangularWindow class that implements our IWindow interface.

```
class RectangularWindow : IWindow
{
    string title;
    Point position;

    public RectangularWindow(string title, Point position)
    {
        this.title = title;
        this.position = position;
    }

    public Point Position
    {
        get { return position; }
        set { position = value; }
    }
    public string Title
    {
        get { return title; }
        set { title = value; }
    }
    public void Draw()
    {
                   Console.WriteLine("Drawing Rectangular Window");
    }
    public void Minimize()
    {
        Console.WriteLine("Minimizing Rectangular Window");
    }
}
```

The RectangularWindow class implements the IWindow interface using the colon operator.

```
class RectangularWindow : IWindow
```

Since RectangularWindow is implementing the IWindow interface and is not declared as an abstract class, it provides the body for all the members of the IWindow interface. It contains two private fields: title and position. The properties Title and Position specified in the IWindow interface are implemented as an accessor to the title and position fields respectively. The two methods, Draw() and Minimize(), are implemented to print a simple message.

Note that while all the implementing members in RectangularWindow are declared as public, they are not declared public in the IWindow interface.

If we don't mark these members as public, like

```
    void Draw()
    {
           Console.WriteLine("Drawing Rectangular Window");
    }
```

We will get the compile-time error:

```
'CSharpSchool.RectangularWindow' does not implement interface member
'CSharpSchool.IWindow.Draw()'.
'CSharpSchool.RectangularWindow.Draw()' is either static, not public, or has
the wrong return type.
```

The compiler is quite supportive here and suggests some possible reasons for the error. In fact, as we stated earlier, all the members of the interface are abstract and public by default and we can not decrease the accessibility level of the original member during polymorphism, hence we must declare all the members as public when implementing an interface.

Also note that there is no override keyword when overriding the abstract methods of the interface (IWindow) in the implementing class (RectangularWindow), like we used to have in C# during polymorphism. If you write override before a member, you will get the compile time error:

'CSharpSchool.RectangularWindow.Draw()': no suitable method found to override.

The reason for not applying the override keyword here is that we do not actually override any default implementation, but provide our own specific implementation for the members.

Implementing More Than One Interface

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, like this:

```
    interface ILoggable
    {
        void Log(string filename);
    }

    interface IFile
    {
        string ReadLine();
         void Write(string s);
    }
```

The ILoggable interface contains a single method Log(string) which logs the activities in the specified file while the interface IFile contains two methods: ReadLine() and Write(string) . The method ReadLine() reads a line from the file and the method Write(string) writes the supplied string to the file. Let us define a class (MyFile) which implements these interfaces.

```
class MyFile : ILoggable, IFile
{
    private string filename;

    public MyFile(string filename)
    {
        this.filename = filename;
    }

    public void Log(string filename)
    {
        Console.WriteLine("Logging activities in file: {0}", filename);
    }

    public string ReadLine()
    {
        return "A line from MyFile";
    }

    public void Write(string s)
    {
        Console.WriteLine("Writing `{0}' in the file", s);
    }
}
```

MyFile implements both the ILoggable and the IFile interfaces and uses a comma ',' to separate the list of interfaces:

```
    class MyFile : ILoggable, IFile
```

The class gives 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 operate on the file whose name is supplied in the constructor. The Log() method logs the activities of the process to the specified method. The Test class containing the Main() method is implemented as:

```
    class Test
    {
        static void Main()
        {
            MyFile aFile = new MyFile("myfile.txt");
            aFile.Log("c:\\csharp.txt");
            aFile.Write("My name is Faraz");
            Console.WriteLine(aFile.ReadLine());
        }
    }
```

Here, we have created an instance of MyFile and named it aFile. We then call different methods of the class MyFile using its object. The sample output of the program is:

```
Logging activities in file: c:\csharp.txt
Writing `My name is Faraz' in the file: myfile.txt
A line from MyFile: myfile.txt
Press any key to continue
```

Up to this point, there should be no confusion about implementing multiple interfaces, but what if the name of method in ILoggable was Write(string) rather than Log(string)? The class MyFile would then be implementing two interfaces (ILoggable and IFile) and both have a method Write(string) with similar signatures.

```
    interface ILoggable
    {
        void Write(string filename);
    }

    interface IFile
    {
        string ReadLine();
            void Write(string s);
    }
```

Explicit implementation of methods

If a class is implementing more than one interface and at least two of them have methods with similar signatures, then we can provide explicit implementation of the particular method by prefixing its name with the name of the interface and the . operator. Consider the case defined above where we have two interfaces (ILoggable and IFile) and both contain a Write(string) method with identical signatures. Let's now change our class MyFile accordingly to provide explicit implementation of the Write() method.

```
class MyFile : ILoggable, IFile
{
    private string filename;

    public MyFile(string filename)
    {
        this.filename = filename;
    }

    void ILoggable.Write(string filename)
    {
        Console.WriteLine("Logging activities in file: {0}", filename);
    }

    public string ReadLine()
    {
        return "A line from MyFile: " + filename;
    }

    public void Write(string s)
    {
        Console.WriteLine("Writing `{0}' in the file: {1}", s, filename);
    }
}
```

Here we have defined the Write() method of ILoggable explicitly by prefixing it with the interface name:

```
void ILoggable.Write(string filename)
```

Now for the compiler, the two Write() methods are distinguishable in the class definition. Now, if we write our Main() method as

```
static void Main()
{
        MyFile aFile = new MyFile("myfile.txt");
        aFile.Write("c:\\csharp.txt");
        aFile.Write("My name is Faraz");
        Console.WriteLine(aFile.ReadLine());
    }
```

we would get the following output:

```
Writing `c:\csharp.txt' in the file: myfile.txt
Writing `My name is Faraz' in the file: myfile.txt
A line from MyFile: myfile.txt
Press any key to continue
```

Note that in both the calls to the Write() method, the implicit version (IFile.Write(string)) is executed. To call the ILoggable.Write() method, we have to type-cast the object in the reference of the ILoggable interface.

```
    ILoggable logger = aFile;
    log.Write("c:\\csharp.txt");
```

Since ILoggable is the parent type of MyFile (as MyFile implements ILoggable), an object of type MyFile (aFile) can be implicitly casted to the ILoggable reference (logger). The modified Main() method would then be:

```
    static void Main()
    {
        MyFile aFile = new MyFile("myfile.txt");
        aFile.Write("My name is Faraz");
        ILoggable logger = aFile;
        logger.Write("c:\\csharp.txt");
        Console.WriteLine(aFile.ReadLine());
    }
```

And the output will be:

```
Writing `My name is Faraz' in the file: myfile.txt
Logging activities in file: c:\csharp.txt
A line from MyFile: myfile.txt
Press any key to continue
```

Casting to an interface using is and as operators

In the last example, we knew that the object (aFile) is cast-able to the ILoggable interface, so casting was straightforward. But, in a case where we are not sure whether a particular object is really cast-able to the interface

type at run-time, we can make use of the is and the as operators (The is and as operators are discussed in more detail in lesson 5 - Inheritance and Polymorphism).
The is operator is used to check the type of a particular object and it returns a boolean result. For example,

```
    MyFile aFile = new MyFile("myfile.txt");
    Console.WriteLine(aFile is ILoggable);
```

Will print True on the standard output since MyFile is a sub-type of ILoggable, while

```
    string s = "faraz";
    Console.WriteLine(s is ILoggable);
```

Will print False. The as operator returns null if the object is not cast-able. Otherwise, it casts the object to the specified type. For example:

```
   MyFile aFile = new MyFile("myfile.txt");
    ILoggable logger = aFile as ILoggable;
    if(logger == null)
        Console.WriteLine("un-successful cast");
    else
        Console.WriteLine("successful cast");
```

The output will be:

```
successful cast
```

While for:

```
    Test t = new Test();
    ILoggable logger = t as ILoggable;
    if(logger == null)
        Console.WriteLine("un-successful cast");
    else
        Console.WriteLine("successful cast");
```

The output will be:

```
un-successful cast
```

Finally, it is worth noting that it is generally preferable 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
    {
        void Write(string s);
    }

    interface IReadable
    {
        string ReadLine();
    }

    interface IFile : IWritable, IReadable
    {
        void Open(string filename);
        void Close();
    }
```

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 us 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 : IFile
    {
        private string filename;

        public void Open(string filename)
        {
            this.filename = filename;
            Console.WriteLine("Opening file: {0}", filename);
        }

        public string ReadLine()
        {
            return "Reading a line from MyFile: " + filename;
        }

        public void Write(string s)
        {
            Console.WriteLine("Writing `{0}' in the file: {1}", s, filename);
        }


        public void Close()
        {
            Console.WriteLine("Closing the file: {0}", filename);
        }
    }
```

The class provides a simple implementation of all the methods by printing a message in the body of each method.

The Test class containing Main() method is:

```
    class Test
    {
        static void Main()
        {
            MyFile aFile = new MyFile();
            aFile.Open("c:\\csharp.txt");
            aFile.Write("My name is Faraz");
            Console.WriteLine(aFile.ReadLine());
            aFile.Close();
        }
    }
```

Here, we have created an instance of MyFile and named the reference aFile. We later call different methods on the object using the reference.

The output of the program is:

```
Opening file: c:\csharp.txt
Writing `My name is Faraz' in the file: c:\csharp.txt
Reading a line from MyFile: c:\csharp.txt
Closing the file: c:\csharp.txt
Press any key to continue
```

The output shows the result of the methods as they are called in order.

7. Arrays, Collections & String Manipulation

Lesson Plan

Today we will explore arrays, collections and string manipulation in C#. First of all, we will explore multidimensional rectangular and jagged arrays. We will also explore how foreach iterates through a collection. Then we will move to collections and see how they are implemented in C#. Later, we will explore different collections like ArrayLists, Stacks, Queues and Dictionaries. Finally, we will see how strings are manipulated in C#. We will explore both the string and the StringBuilder types.

Arrays Revisited

As we have seen earlier, an array is a sequential collection of elements of a similar data type. In C#, an array is an object and thus a reference type, and therefore they are stored on the heap. We have only covered single dimension arrays in the previous lessons, now we will explore multidimensional arrays.

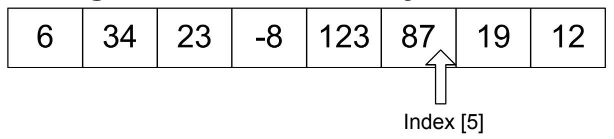
Multidimensional Arrays

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 in a database where each primary element (row) is a collection of 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 C#:

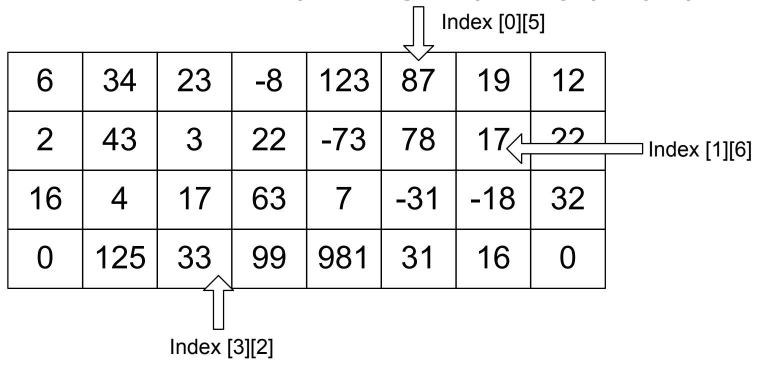
- Rectangular array (one in which each row contains an equal number of columns)
- Jagged array (one in which each row does not necessarily contain an equal number of columns)

The images below show what the different kinds of arrays look like. The figure also shows the indexes of different elements of the arrays. Remember, the first element of an array is always zero (0).

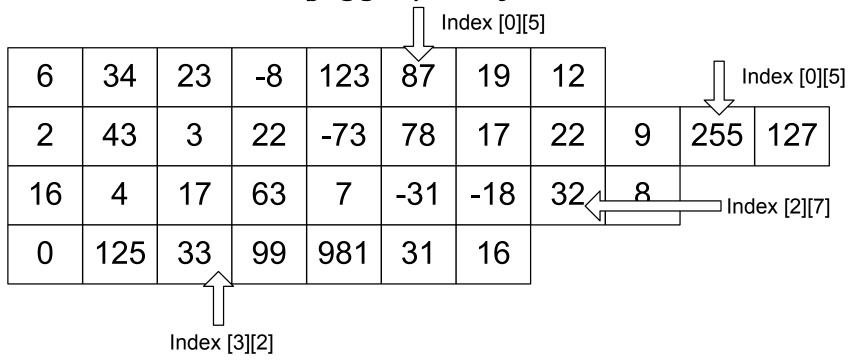
A single Dimensional Array



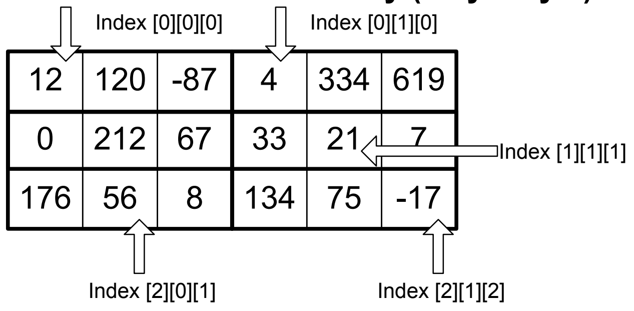
A two-dimensional (rectangular) Array (4 by 4)



A two-dimensional (jagged) Array



A three-dimensional Array (3 by 2 by 3)



Instantiating and accessing the elements of multidimensional arrays

Recall that we instantiate our single dimensional array like this:

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

The above line would instantiate (create) a one dimensional array (intArray) of type int, and with 5 elements. 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 says '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:

```
    int [,] myTable = new int[2,3];
```

All the elements of the array are auto-initialized to their default values; hence all the elements of the myTable array would be initialized with zeroes. We can iterate through this array using either a foreach loop or a for loop.

```
    foreach(int intVal in myTable)
    {
        Console.WriteLine(intVal);
    }
```

When it is compiled and executed, it will print six (2 x 3) zeros.

```
0
0
0
0
0
0
```

Now, let us 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 write the following code:

```
    myTable[0,0] = 32;
```

In the same way we can change the values of other elements of the array:

```
    myTable[0,1] = 2;
    myTable[0,2] = 12;
    myTable[1,0] = 18;
    myTable[1,1] = 74;
    myTable[1,2] = -13;
```

Now, we can use a couple of nested for loops to iterate over the array:

```
    for(int row=0; row<myTable.GetLength(0); row++)
    {
        for(int col=0; col<myTable.GetLength(1); col++)
        {
            Console.WriteLine("Element at {0},{1} is {2}", row, col, myTable[row,col]);
        }
    }
```

Here, we have used two for 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 an array. Note that the Length property will give total number of elements in this two dimensional array, i.e., 6. The output of the above program will be:

```
Element at 0,0 is 3
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
```

Instantiating and accessing Jagged Arrays

A jagged array is one 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 the first row is 3, the second row is 5 and the third row is 2. We can instantiate this jagged array like this:

```
    int [][] myTable = new int[3][];
    myTable[0] = new int[3];
    myTable[1] = new int[5];
    myTable[2] = new int[2];
```

Then we can fill the array like this:

```
    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;
```

Now, we will show how to use the foreach loop to access the elements of the array:

```
    foreach(int []row in myTable)
    {
        foreach(int col in row)
        {
            Console.WriteLine(col);
        }
        Console.WriteLine();
    }
```

The code above is very simple and easily understandable. We picked up each row (which is an int 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:

```
    int [,,] myTable = new int[3,2,4];


    myTable[0,0,0] = 3;
    myTable[1,1,1] = 6;
```

Or in jagged array fashion:

```
    int [][][] myTable = new int[2][][];
    myTable[0]    = new int[2][];
    myTable[0][0] = new int[3];
    myTable[0][1] = new int[4];

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

    myTable[0][0][0] = 34;
    myTable[0][1][1] = 43;
    myTable[1][2][2] = 76;
```

Here, we have created a three dimensional jagged array. It is an array of two 2-dimensional arrays. The first of the 2-dimensional arrays contains 2 rows. The length of the first row is 3, while the length of second row is 4. In a similar fashion, the second two dimensional array is also initialized. In the end, we have accessed some of the elements of the array and assigned them different values. Although higher dimensional jagged arrays are quite difficult to perceive; they may be very useful in certain complex problems. Again, the key to avoid confusion in multidimensional arrays is to perceive them as an 'array of arrays'.

Some other important points about multidimensional arrays

In the examples we just saw, we have used multidimensional arrays of only the integer type. But, you can declare arrays of any data type. For example, you may define an array of strings or even an array of objects of your own class.

```
    string []names = new string[4];
```

You can initialize the elements of an array on the fly in the following ways

```
    string []names = new string[4]{"Faraz", "Gates", "Hejlsberg", "Gosling"};
```

or

```
    string []names = new string[]{"Faraz", "Gates", "Hejlsberg", "Gosling"};
```

or

```
    string []names = {"Faraz", "Gates", "Hejlsberg", "Gosling"};
```

You can also separate the declaration and initialization of an array reference and the array:

```
    string []names;
    names = new string[4]{"Faraz", "Gates", "Hejlsberg", "Gosling"};
```

You can also initialize two-dimensional arrays on the fly (along with the declaration)

```
    string [][]nameLists = {new string[]{"Faraz", "Gates"},
    new string[]{"Hejlsberg", "Gosling", "Bjarne"}};
```

Some of the more important properties & methods that can be applied on arrays are:

- Length gives the number of elements in all dimensions of an array
- GetLength(int) 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 foreach Loop

We have been using the foreach loop for quite a long time now. Let us see how it works and how can we enable our class to be iterated over by the foreach loop. For this, we need to implement the IEnumerable interface which contains only 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:

```
object Current { get; }
```

It returns the current element of the collection which is of object data type. The MoveNext() method is declared as:

```
bool MoveNext();
```

It advances the current selection to the next element of the collection and returns true if the advancement is successful, and 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 which means that the Current property is not valid until MoveNext() has been executed for the first time.

Finally, the Reset() method is declared as

```
    void Reset();
```

This method resets the enumerator and sets it to the initial state. After reset, MoveNext() will again advance the selection to the first element in the collection.

Now we will show how to make a class that can be iterated over using the foreach loop, by implementing the IEnumerable and IEnumerator interfaces.

```
using System;
using System.Collections;

namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            MyList list = new MyList();
            foreach(string name in list)
            {
                Console.WriteLine(name);
            }
        }
    }

    class MyList : IEnumerable
    {
        static string []names = {"Faraz", "Gates", "Hejlsberg", "Gosling", "Bjarne"};
        public IEnumerator GetEnumerator()
        {
            return new MyEnumerator();
        }

        private class MyEnumerator : IEnumerator
        {
            int index = -1;
            public object Current
            {
                get { return names[index]; }
            }

            public bool MoveNext()
            {
                if(++index >= names.Length)
                    return false;
                else
                    return true;
            }

            public void Reset()
            {
                index = -1;
            }
        }
    }
}
```

Here we have declared a class called MyList which contains a private nested class named MyEnumerator. The class MyEnumerator implements IEnumerator by providing the implementation of its public property and methods. The class MyList implements the IEnumerable interface and returns an object of type MyEnumerator in its GetEnumerator() method. The class MyList contains a static array of strings 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 true or false, depending on whether collection's final element has been reached. The Current property returns the indexth element of the collection while the Reset() method resets the variable index to -1 again.

In the Test class we have instantiated the MyList class and iterated through it using a foreach loop, because we have implemented IEnumerable interface on the MyList class. The output of the above program is:

```
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 when using arrays for collections. The size of an array is always fixed and must be defined at the time of instantiation of an array. Secondly, an array can only contain objects of the same data type, which we need to define at the time of its instantiation. Also, an array does not impose any particular mechanism for inserting and retrieving the elements of a collection. For this purpose, the creators of C# and the .Net Framework Class Library (FCL) have provided a number of classes to serve as a collection of 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 that grows dynamically as the number of elements change.
Stack	A collection that works on the Last In First Out (LIFO) principle, i.e., the last item inserted is the first item removed from the collection.
Queue	A collection that works on the 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 the collection (similar to the Length property of an Array)
- The IsSynchronized property returns a boolean value depending on whether 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 array, int index) method copies the elements of the collection to the array, starting from the specified index.
-

All the collection classes also implement the IEnumerable interface, so they can be iterated over using the foreach 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 into 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 like this:

```
    ArrayList list = new ArrayList();
```

We can also specify the initial Capacity of the ArrayList by passing an integer value to the constructor:

```
    ArrayList list = new ArrayList(20);
```

We can also create an ArrayList with some other collection by passing the collection in the constructor:

```
    ArrayList list = new ArrayList(someCollection);
```

We 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 a foreach loop:

```
    static void Main()
    {
        ArrayList list = new ArrayList();
        list.Add(45);
        list.Add(87);
        list.Add(12);
        foreach(int num in list)
        {
            Console.WriteLine(num);
        }
    }
```

which will print out the elements in the 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 it to the Add() method, which expects an instance of type object. The reason for this is that in C# boxing is performed automatically and the compiler boxes the value types to the object implicitly.

The ArrayList class has also implemented the indexer property (or index operator) which allow its elements to be accessed using the [] operators, just as you do with a simple array (we will see how to implement indexers in the next lesson). The following code is similar to the above code but uses the indexers to access the elements of the ArrayList.

```
static void Main()
{
    ArrayList list = new ArrayList();
    list.Add(45);
    list.Add(87);
    list.Add(12);
    for(int i=0; i<list.Count; i++)
    {
        Console.WriteLine(list[i]);
    }
}
```

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(int)	Removes an element at the specified index from the ArrayList.
Insert(int, 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. This method is overloaded and one can specify the range to be copied and also from which index of the array copy should start.
IndexOf(object)	Returns the zero based index of the first occurrence of the object in the ArrayList. If

	the object is not found in the ArrayList, it returns -1.
LastIndexOf(object)	Returns 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 collection 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 onto the stack is termed as a '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 a 'Peek' operation. The System.Collections.Stack class provides the functionality of a Stack in the .Net environment. The Stack class can be instantiated in a manner similar to the one we used for the ArrayList.

```
    Stack stack = 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:

```
    Stack stack = new Stack(12);
```

While the following constructor will initialize the Stack with the supplied collection:

```
    Stack stack = new Stack(myCollection);
```

Now, we can push elements onto the Stack using the Push() method, like this:

```
    stack.Push(2);
    stack.Push(4);
    stack.Push(6);
```

In a similar manner, we can retrieve elements from the stack using the Pop() method. A complete program that pushes 3 elements onto the stack and then pops them off one by one is presented below:

```
using System;
using System.Collections;

namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            Stack stack = new Stack();
            stack.Push(2);
            stack.Push(4);
            stack.Push(6);

            while(stack.Count != 0)
            {
                Console.WriteLine(stack.Pop());
            }
        }
    }
}
```

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 a stack is that the Pop() operation not only returns 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 will be:

```
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 the Peek() operation.

```
    static void Main()
    {
        Stack stack = new Stack();
        stack.Push(2);
        stack.Push(4);
        stack.Push(6);

         Console.WriteLine("The total number of elements on the stack before Peek() = {0}", stack.Count);
         Console.WriteLine("The top element of stack is {0}",
         stack.Peek());
         Console.WriteLine("The total number of elements on the stack after Peek() = {0}", stack.Count);
    }
```

The above program pushes three elements onto the stack and then peeks at the top element on the stack. The program prints the number of elements on the stack before and after the Peek() operation. The result of the program is:

```
The total number of elements on the stack before Peek() = 3
The top element of stack is 6
The total number of elements on the stack after Peek() = 3
Press any key to continue
```

The output of the program shows that Peek() does not affect the number of elements on the stack and does not remove the top element, contrary to the 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 the queue will be the first item removed from it. To 'Enqueue' an item is to insert it into the queue, and removal of an item from the queue is termed 'Dequeue'. Like a stack, there is also a Peek operation, where the item is not removed but only read from the front of the queue.

The System.Collections.Queue class provides the functionality of queues in the .Net environment. The Queue's constructors are similar to those of the ArrayList and the Stack.

```
    // an empty queue
    Queue queue = new Queue();

    // a queue with initial capacity 16
    Queue queue = new Queue(16);

    // a queue containing elements from myCollection
    Queue queue = new Queue(myCollection);
```

The following program demonstrates the use of Queues in C#.

```
    static void Main()
    {
        Queue queue = new Queue();
        queue.Enqueue(2);
        queue.Enqueue(4);
        queue.Enqueue(6);

        while(queue.Count != 0)
        {
            Console.WriteLine(queue.Dequeue());
        }
    }
```

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 items in the order they were inserted. The other methods of a Queue are very similar to those of the ArrayList and Stack classes.

Dictionaries

Dictionaries are a kind of collection that store 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.

The Hashtable class

Hashtable stores items as key-value pairs. Each item (or value) in the hashtable is uniquely identified by its key. A hashtable stores the key and its value as an object type. Mostly the string class is used as the key in a hashtable, but you can use any other class as a key. However, before selecting a class for the key, be sure to override the Equals() and GetHashCode() methods that it inherit 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 for usage as a key in hashtables or other dictionaries. There are many constructors available to instantiate a hashtable. The simplest is:

```
    Hashtable ht =  new Hashtable();
```

Which is a default no argument constructor. A hashtable can also be constructed by passing in the initial capacity:

```
    Hashtable ht =  new Hashtable(20);
```

The Hashtable class also contains some other constructors which allow you to initialize the hashtable with some other collection or dictionary.

Adding items to a Hashtable

Once you have instantiated a hashtable object, then 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 the 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);
```

Here we have used the indexer ([] operator) to retrieve the value 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). 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.

```
    static void Main()
    {
        Hashtable ht =  new Hashtable(20);
        ht.Add("st01", "Faraz");
        ht.Add("sci01", "Newton");
        ht.Add("sci02", "Einstein");

        Console.WriteLine("Size of Hashtable is {0}", ht.Count);
        Console.WriteLine("Removing element with key = st01");
        ht.Remove("st01");
        Console.WriteLine("Size of Hashtable is {0}", ht.Count);
    }
```

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 property, which return an ICollection containing all the keys and values respectively. The following program iterates through all the keys and values and prints them, using a foreach loop.

```
    static void Main()
    {
        Hashtable ht =  new Hashtable(20);
        ht.Add("st01", "Faraz");
        ht.Add("sci01", "Newton");
        ht.Add("sci02", "Einstein");

        Console.WriteLine("Printing Keys...");
        foreach(string key in ht.Keys)
        {
            Console.WriteLine(key);
        }

        Console.WriteLine("\nPrinting Values...");
        foreach(string Value in ht.Values)
        {
            Console.WriteLine(Value);
        }
    }
```

The output of the program will be:

```
Printing Keys...
st01
sci02
sci01

Printing Values...
Faraz
Einstein
Newton
Press any key to continue
```

Checking for the existence of a particular item in a hashtable

You can use the ContainsKey() and the ContainsValue() method to find out whether a particular item with the specified key and value exists in the hashtable or not. Both the methods return a boolean value.

```
    static void Main()
    {
        Hashtable ht =  new Hashtable(20);
        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"));
    }
```

The output is:

```
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 you can with an array. In the case of SortedList, if you want to use your own class as a key then, in addition to the considerations described in Hashtable, you also need to make sure that your class implements the IComparable interface.

The IComparable interface has only one method: int CompareTo(object obj). 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.

- A return value of 0 indicates that this object is equal to the supplied obj.
- A return value greater than zero indicates that this object is greater than the supplied obj
- A return value less than zero indicates that this object is less than the supplied obj.

The string class and other primitive data types provide an implementation of this interface and hence can be used as keys in a SortedList directly.

The SortedList provides similar constructors as provided by the Hashtable and the simplest one is a zero argument constructor.

```
    SortedList sl =  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(object key, object value)	Adds an element (key-value pair) to the SortedList.
GetKey(int index)	Returns the key at the specified index.
GetByIndex(int index)	Returns the value at the specified index.
IndexOfKey(object key)	Returns the zero based index of the specified key.
IndexOfValue(object value)	Returns the zero based index of the specified value.
Remove(object key)	Removes the element with the specified key from the SortedList.
RemoveAt(int)	Removes the element at the specified index from the SortedList.
Clear()	Removes all the elements from the SortedList.
ContainsKey(object key)	Returns a boolean value indicating whether the SortedList contains an element with the supplied key.
ContainsValue(object value)	Returns a boolean value indicating whether the SortedList contains an element with the supplied value.

The following program demonstrates the use of a SortedList.

```
static void Main()
{
    SortedList sl =  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("\t Key \t\t Value");
    Console.WriteLine("\t === \t\t =====");
    for(int i=0; i<sl.Count; i++)
    {
        Console.WriteLine("\t {0} \t\t {1}",  sl.GetKey(i), sl.GetByIndex(i));
    }
}
```

The program 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 program 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 C#

The next topic in today's lesson is about handling strings in the C# programming language. In C#, a string is a built in and primitive data type. The primitive data type string maps to the System.String class. The objects of the String class (or string) are immutable by nature. By immutable it 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 string in the .Net Platform is 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 a

StringBuilder class object, the new string will be appended to (added to the end of) the original object. Let's now discuss the two classes one by one.

The string class and its members

We have been using the string class since our first lesson in the C# school. We have also seen some of its properties (like Length) and methods (like Equals()) in previous lessons. Here, 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(string s)	Compares this instance of the string with the supplied string s and returns an integer on the pattern of the IComparable interface.
Compare(string s1, string s2)	This is a static method and compares the two supplied strings on the pattern of the IComparable interface.
Equals(string s)	Returns true if the supplied string is exactly the same as this string, else returns false.
Concat(string s)	Returns a new string that is the concatenation (addition) of this and the supplied string s.
Insert(int index, string s)	Returns a new string by inserting the supplied string s at the specified index of this string.
Copy(string s)	This static method returns a new string that is the copy of the supplied string.
Intern(string s)	This static method returns the system's reference to the supplied string.
StartsWith(string s)	Returns true if this string starts with the supplied string s.
EndsWith(string s)	Returns true if this string ends with the supplied string s.

IndexOf(string s)
IndexOf(char ch)
	Returns the zero based index of the first occurrence of the supplied string s or supplied character ch. This method is overloaded and more versions are available.

LastIndexOf(string s)
LastIndexOf(char ch)
	Returns the zero based index of the last occurrence of the supplied string s or supplied character ch. This 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 char[])	Identifies those substrings (in this instance) which are delimited by one or more characters specified in an array, then places the substrings into a String array and returns it.

Substring(int i1)
Substring(int i2, int i3)
	Retrieves a substring of this string starting from the index position i1 till the end in the first overloaded form. In the second overloaded form, it retrieves the substring starting from the index i2 and which 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 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 program we have demonstrated the use of most of the above methods of the string class. The program is quite self-explanatory and only includes the method calls and their results.

```
using System;

namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            string s1 = "faraz";
            string s2 = "fraz";
            string s3 = "Faraz";
            string s4 = "C# is a great programming language!";
            string s5 = "    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);

            string []words = s4.Split(' ');
            foreach(string word in words)
            {
                Console.WriteLine("\t {0}", word);
            }

                Console.WriteLine("\nThe substring of \n\t`{0}' \nfrom index 3 of length 10 is \n\t`{1}'",
                  s4, s4.Substring(3, 10));
            Console.WriteLine("\nThe string \n\t`{0}'\nafter trimming is \n\t`{1}'", s5, s5.Trim());
        }
    }
}
```

The output of the program will certainly help you understand the behavior of each member.

```
Length of faraz is 5
Comparison 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 `C# is a great programming language!' are
C#
is
a
great
programming
language!

The substring of
`C# is a great programming language!'
from index 3 of length 10 is
`is a great'

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 that it is mutable; that is, the internal state of its objects can be modified by its operations. Unlike in the string class, you must first call the constructor of a StringBuilder to instantiate its object.

```
    string s = "This is held by string";
    StringBuilder sb = new StringBuilder("This is held by StringBuilder");
```

StringBuilder is somewhat similar to ArrayList and other collections in the way that it grows automatically as the size of the string it contains changes. Hence, the Capacity of a 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(int st, int length)	Removes all characters from the index position st of specified length in the current StringBuilder object.
Equals(StringBuilder)	Checks the supplied StringBuilder object with this instance and returns true if both are identical; otherwise, it returns false.

The following program demonstrates the use of some of these methods:

```
using System;
using System.Text;

namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
                StringBuilder sb = new StringBuilder("The text");
                string s = " 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("\nStringBuilder before appending is `{0}'", sb);
                 Console.WriteLine("StringBuilder after appending `{0}' is `{1}'", s, sb.Append(s));
                 Console.WriteLine("\nStringBuilder after inserting `now' is `{0}'", sb.Insert(11, "
now"));
                 Console.WriteLine("\nStringBuilder after removing 'is ' is `{0}'", sb.Remove(8, 3));
                 Console.WriteLine("\nStringBuilder after replacing all `e' with `x' is {0}",
                   sb.Replace('e', 'x'));
        }
    }
}
```

And the output of the program 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 getting modified, hence StringBuilder objects are mutable compared to the immutable String objects.

8. Exception Handling

Lesson Plan

Today we will explore exception handling mechanisms in C#. We will start out by looking at the idea behind exceptions and will see how different exceptions are caught. Later, we will explore different features provided and constraints applied by the C# compiler in exception handling. Finally, we will learn how to define our own custom exceptions.

Exceptions Basics

Put simply, exception handling is a mechanism to handle run-time errors in .NET that would otherwise cause your software to terminate prematurely.

The need for Exceptions

Consider the following simple code:

```
using System;

namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            double p = Divide(3, 5);
            Console.WriteLine(p);
        }
        static double Divide(double a, double b)
        {
            double c = a/b;
            return c;
        }
    }
}
```

The Divide() method returns the result of the division of the two numbers passed as parameters. The output of the program will be:

```
0.6
```

But what if the method Divide() is called in the Main() method as

```
    double p = Divide(3, 0);
```

There will be no compile time error as both the parameters are cast-able to double. But when the program is executed, the computer will attempt to divide 3 by zero which is not possible. As a result, the program will crash! One way to deal with this is to explicitly check the second parameter and if it is zero return some value to indicate the error

```
    static double Divide(double a, double b)
    {
        if(d == 0)
        {
            return 0;
        }
        double c = a/b;
        return c;
    }
```

The calling code will be:

```
    static void Main()
    {
        double p = Divide(3, 5);
        if(p == 0)
            Console.WriteLine("Error in input");
        else
            Console.WriteLine(p);
    }
```

Here we return a value 'zero' to indicate an error in the input. But the problem is that zero can itself be the result of some division like (0 divided by 4) and in this case the caller may get confused whether the returned value is an error or is a legal result, e.g.

```
    static void Main()
    {
        double p = Divide(0, 3);
        if(p == 0)
            Console.WriteLine("Error in input");
        else
```

```
            Console.WriteLine(p);
    }
```

In this case the program will display that there is an error in the input, which is clearly not the case. So how do we solve this problem?

Exceptions in C# and .Net

People have worked on this problem and 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 exception handling mechanisms in C# and .Net

- All exceptions in .Net are objects. The System.Exception class is the base class for all 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 C#. 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 the exception. We can define which particular class of exception we want to deal in this catch block by mentioning the name of exception after 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, placed just after the try or try...catch blocks. This code is guaranteed to always be executed whether the exception occurs or not.
- When an exception is raised during the execution of the 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's exceptions. Attempting to catch any of the sub-class's exceptions explicitly after the parent class exception will cause a compile time error.
- The finally block is optional. Exception handling requires a combination of either the try...catch, 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, causing your program to terminate.

Author's Note: For Java developers, there is no concept of checked exceptions in C#. All the exceptions are implicitly unchecked. Hence, there is no 'throws' keyword present in C#. The absence of checked exceptions in C# is quite a hot topic of debate since the birth of C#. I personally feel that the idea of checked exceptions is extremely good and it should have been there in C# or in any strongly typed language. See the Microsoft site for the argument of Hejlsberg and other designers of C# for why they haven't included checked exceptions in C#.

http://msdn.microsoft.com/chats/transcripts/vstudio/vstudio_032103.aspx

Handling Exceptions using the try...catch...finally blocks

Use of the try...catch block

A simple demonstration of the use of the try...catch blocks is given below.

```
static void Main()
{
    string s = null;
    try
    {
        Console.WriteLine("In try block... before calling s.ToLower()");
        Console.WriteLine(s.ToLower());
        Console.WriteLine("In try block... after calling s.ToLower()");
    }
    catch(NullReferenceException e)
    {
        Console.WriteLine("In catch block...");
        Console.WriteLine("NullReferenceException Caught");
    }
    Console.WriteLine("After try...catch block");
}
```

The string 's' in Main() is assigned a null value. When we attempt to call the ToLower() method on 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. In parenthesis 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 output is:

```
In try block... before calling s.ToLower()
In catch block...
NullReferenceException Caught
After try...catch block
Press any key to continue
```

Look carefully at the output of the program and compare it with the source code. The call to s.ToLower() raises 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. Lets change the code above a bit and assign an object to the reference 's'

```
        string s = "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(NullReferenceException e)
        {
            Console.WriteLine("In catch block...");
            Console.WriteLine("NullReferenceException Caught");
        }
        Console.WriteLine("After try...catch block");
```

Since this program does not cause any exceptions to be raised, the execution of program will result into:

```
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 Properties

Note that the code under the catch block didn't get executed because of the absence of NullReferenceException. Now lets remove the try...catch block, assign s to null as we did in the first place, and see what happens.

```
    static void Main()
    {
        string s = null;
        Console.WriteLine("Before printing lower case string...");
        Console.WriteLine(s.ToLower());
        Console.WriteLine("After printing lower case string...");
    }
```

When we compile and execute the above program, 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 CSharpSchool.Test.Main() in
c:\documents and settings\administrator\my documents\visual studio
projects\myconsole\class1.cs:line 11
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. Exception Message: this describes the exception

2. 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 of the Test class contained in the CSharpSchool namespace (CSharpSchool.Test.Main()). The Stack trace also points out the filename along with its complete path and the line number which contains the cause for 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 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 property of the Exception class. Consider the following code:

```
    static void Main()
    {
        string s = null;
        try
        {
            Console.WriteLine("In try block... before calling s.ToLower()");
            Console.WriteLine(s.ToLower());
            Console.WriteLine("In try block... after calling s.ToLower()");
        }
        catch(NullReferenceException e)
        {
            Console.WriteLine("\nIn catch block...");
            Console.WriteLine("NullReferenceException Caught");
            Console.WriteLine("\nException Message");
            Console.WriteLine("=============");
            Console.WriteLine(e.Message);
            Console.WriteLine("\nException Stack Trace");
            Console.WriteLine("=============");
            Console.WriteLine(e.StackTrace);
        }
        Console.WriteLine("\nAfter try...catch block");
    }
```

The difference between this one and the previous code is that here we have printed the explanatory message and stack trace of the exception explicitly by using 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 CSharpSchool.Test.Main() in c:\documents and settings\administrator\
my documents\visual studio projects\myconsole\class1.cs:line 14

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 that could not be closed because of the exception. For example, the finally block can be used to close the file, database, socket connections and other important resources opened in the try block. Let's add the finally block to our previous example.

```
static void Main()
    {
        string s = "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(NullReferenceException e)
        {
            Console.WriteLine("\nIn catch block...");
            Console.WriteLine("NullReferenceException Caught");
        }
        finally
        {
            Console.WriteLine("\nIn finally block...");
        }
    }
```

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. Lets cause an exception to occur by setting the string 's' to null in Main()

```
      static void Main()
    {
        string s = null;
        ...
    }
```

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 regardless of if an exception occurred or not.

It is possible to write the try...finally block without any catch block, e.g.:

```
    static void Main()
    {
        string s = "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("\nIn finally block...");
        }
    }
```

The output of the program will be:

```
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 thrown in a try block using multiple (or a series of) catch blocks. For example, we can write three catch blocks; one for catching the NullReferenceException, second for catching the 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 size of the array. The code below demonstrates the use of multiple catch blocks.

```
    static void Main()
    {
        string s = "Faraz";
        int []i = new int[3];
        try
        {
            Console.WriteLine("Entering the try block...\n");

            // 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] = 4/i[0];
            Console.WriteLine("\nLeaving the try block...");
        }
        catch(NullReferenceException e)
        {
            Console.WriteLine("\nIn catch block...");
            Console.WriteLine("NullReferenceException Caught");
        }
        catch(IndexOutOfRangeException e)
        {
            Console.WriteLine("\nIn catch block...");
            Console.WriteLine("IndexOutOfRangeException Caught");
        }
        catch(Exception e)
        {
            Console.WriteLine("\nIn catch block...");
            Console.WriteLine("Exception Caught");
            Console.WriteLine(e.Message);
        }
    }
```

Here we have used a string 's' and an integer array i. The size of the int array is declared as 3. There are three places in the program where we will introduce the occurrence of exceptions:

- 's' and 'i' can be null, causing a NullReferenceException.
- Access to the array 'i' may cause an IndexOutOfRangeException
- The division of 4 by i[0] may cause a DivideByZeroException if the value of i[0] is zero.

We have declared the 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 it.

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 has occurred. Let us first make the string reference s null and see the effect.

```
static void Main()
    {
        string s = null;
        ...
    }
```

The output will be:

```
Entering the try block...

In catch block...
NullReferenceException Caught
Press any key to continue
```

It looks similar and is very much as expected. Now change the array index to some out of bound value (either less than zero or greater than or equal to 3). Also change the string 's' to point to some 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] has caused a DivideByZeroException. The runtime checked for the presence of a catch block. In the third catch block it found an Exception, which is the super type of the DivideByZeroException (and any other exception in .Net). The execution was then transferred to that corresponding catch block.

An important point to remember in 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 a compile time error. For example, consider the following code:

```
    static void Main()
    {
        string s = null;
        try
        {
            Console.WriteLine("Entering the try block...\n");
            // can cause NullReferenceException
            Console.WriteLine("Lower case name is: " + s.ToLower());
            Console.WriteLine("\nLeaving the try block...");
        }
        catch(Exception e)
        {
            Console.WriteLine("\nIn catch block...");
            Console.WriteLine("Exception Caught");
            Console.WriteLine(e.Message);
        }
        catch(NullReferenceException e)
        {
                Console.WriteLine("\nIn catch block...");
                Console.WriteLine("NullReferenceException Caught");
        }
    }
```

Here, the catch block to handle the Exception type exception is placed prior to its sub-class exception (NullReferenceException). Now, 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 "dead code" - it can never be reached. The compiler detects this and produces an error when we attempt to compile the program.

```
A previous catch clause already catches all exceptions of this or a super type
('System.Exception')
```

Hence, one must consider the inheritance hierarchy of different exceptions when using multiple catch blocks.

Other important points about Exception Handling in C#

It is possible to write the name of the exception class only in the catch parenthesis without mentioning any reference, like this:

```
    catch(NullReferenceException)
    {
        ...
    }
```

However, we would suggest that you do not do this.

It is also permissible to write only the catch keyword without parenthesis, which is similar to catching the Exception type exception or catching any exception as

```
    catch
    {
        ...
    }
```

We would again and strongly suggest not to use it, but rather use the Exception class to catch all the exceptions, like this:

```
    catch(Exception e)
    {
        ...
    }
```

Defining your own custom exceptions

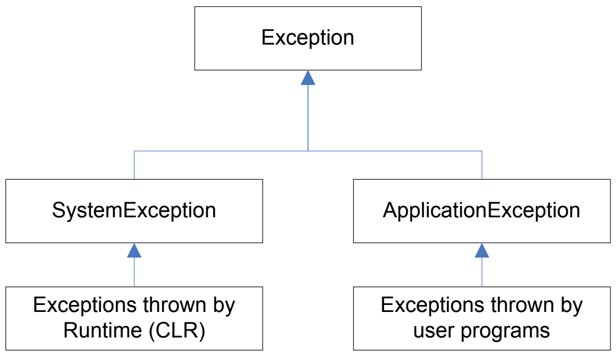
It is possible in C# to define your own custom exceptions to identify the occurrence of unexpected events in your code. As stated earlier, exceptions are implemented in C# as classes and objects. So in order to define a new custom exception, one must define a new class for this. But before understanding 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 the .Net Framework

There are two types of exceptions in .Net:

- Exceptions generated by the runtime (Common Language Runtime) are called System Exceptions.
- Exceptions generated by the user's program are called Application Exceptions.

The simple hierarchy of exceptions in the .Net framework is shown in the following diagram.


Exception
SystemException
ApplicationException
Exceptions thrown by Runtime (CLR)
Exceptions thrown by user programs

Exceptions Hierarchy in the .NET framework

Hence, all the 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, shown earlier in the lesson, when the second argument is zero. The code to define our custom exception, InvalidArgumentException, is:

```
class InvalidArgumentException : ApplicationException
{
    public InvalidArgumentException() : base("Divide By Zero Error")
    {
    }

    public InvalidArgumentException(string message) : base(message)
    {
    }
}
```

Our custom exception class 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 the constructors pass this message to the base class constructor, which initializes the Message property (originally present in the Exception class) with the supplied string. Now that we have defined our own custom exception, it is 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.

```
    static double Divide(double a, double b)
    {
        if(b==0)
            throw new InvalidArgumentException();
        double c = a/b;
        return c;
    }
```

Here, 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 the exception, like:

```
throw new InvalidArgumentException("Error: The second argument of Divide is zero");
```

Now the Main() method will use this method and catch the exception using the try...catch block.

```
static void Main()
{
    try
    {
        Console.WriteLine("In try block...");
        double d = Divide(3, 0);
        Console.WriteLine("\tResult of division: {0}",d);
    }
    catch(InvalidArgumentException e)
    {
        Console.WriteLine("\nIn catch block...");
        Console.WriteLine("\tSystem.InvalidArgumentException caught...");
        Console.WriteLine("\tMessage: " + e.Message);
    }
}
```

The code is very similar to the earlier examples, except that now we are catching exceptions that we have defined. When the program is executed, the following output is generated:

```
In try block...

In catch block...
System.InvalidArgumentException caught...
Message: Divide By Zero Error
Press any key to continue
```

Note that here the Message Property returns our own custom message. Some important points about defining your own exception:

- It is a convention in the .NET framework that the names of all exceptions end with the word 'Exception' like SystemException, NullReferenceException, IndexOutOfRangeException, etc. Hence, we would strongly recommend following this naming convention.
- Always derive your custom exception from the ApplicationException class.
- Catching and throwing exceptions has some performance overhead, so it is not a good programming practice to throw unnecessary exceptions.

9. Delegates & Events

Lesson Plan

Today 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 C#. Later, we will explore multicast delegates and their significance. Finally, we will learn about events and event handling mechanism in C# through delegates.

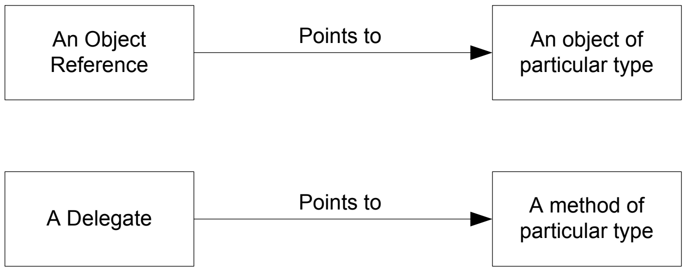
Delegates Basics

Delegates are references to methods. So far we have used references to objects, e.g. Stack st = new Stack(); Here st is a reference to an object of the Stack class type. Hence, each reference has two properties: 1. The type of object (class) the reference can point to. 2. The actual object referenced (or pointed to) by the reference.

Delegates are similar to object references, but are used to reference methods instead of objects. The type of a delegate is the type or signature of the method rather than the class. Hence a delegate has three properties:

1. The type or signature of the method that the delegate can point to
2. The 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++.



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 the method the delegate can reference. The signature of a method includes its return type and the type of parameters which it requires to be passed.

For example:

```
int someMethod(string [] args)
```

Is the common signature of the Main() method of a C# programs defined as:

```
int Main(string [] args)
{
...
}
```

And for the following Add() method:

```
int Add(int a, int b)
{
    return a+b;
}
```

The signature will be:

```
int aMethod(int p, int q)
```

Which is also the signature of following Subtract() method:

```
int Subtract(int c, int d)
{
    return c-d;
}
```

It should be noticed from the above examples that the name of a method is not the part of its signature; the signature only involves its return type and parameters.

In case of delegates, we define the type of a delegate using the delegate keyword, e.g.

```
delegate int MyDelegate(int p, int q);
```

Here we have defined a delegate type with the name 'MyDelegate'. The reference of this delegate type can be used to point to any method which takes two integers as parameters and returns an integer value.

The delegate reference, that can be used to reference a method

Once we have defined a delegate type, we can set it to reference actual methods with matching signatures. A delegate reference can be declared just like an object reference. For example, a reference of type MyDelegate (defined above) can be declared as:

```
MyDelegate arithMethod;
```

The delegate reference arithMethod can now reference any method whose signature is identical to the signature of MyDelegate.

3.The actual method referenced by the delegate

The delegate reference can be made to reference any method with a matching signature by passing its name as the parameter of delegate:

```
arithMethod = new MyDelegate(Add);
```

Here, the arithMethod delegate reference is made to point to the Add() method by passing its name as a parameter to the delegate type (MyDelegate). The last two steps can be merged together in a single statement, like this:-

```
MyDelegate arithMethod = new MyDelegate(Add);
```

Calling the actual method through its delegate

Once the delegate reference 'arithMethod' has been made to point to the Add() method, it can be used to call the actual method like this:-

```
int r = arithMethod(3, 4);
```

The complete source code of the program is presented below.

```
using System;

namespace CSharpSchool
{
    class Test
    {
        delegate int MyDelegate(int p, int q);

         static void Main()
         {
             MyDelegate arithMethod = new MyDelegate(Add);
```

```
            int r = arithMethod(3, 4);
            Console.WriteLine("The result of arithmetic operation `+' on 3 and 4 is: {0}", r);
        }

        static int Add(int a, int b)
        {
            return a + b;
        }
    }
}
```

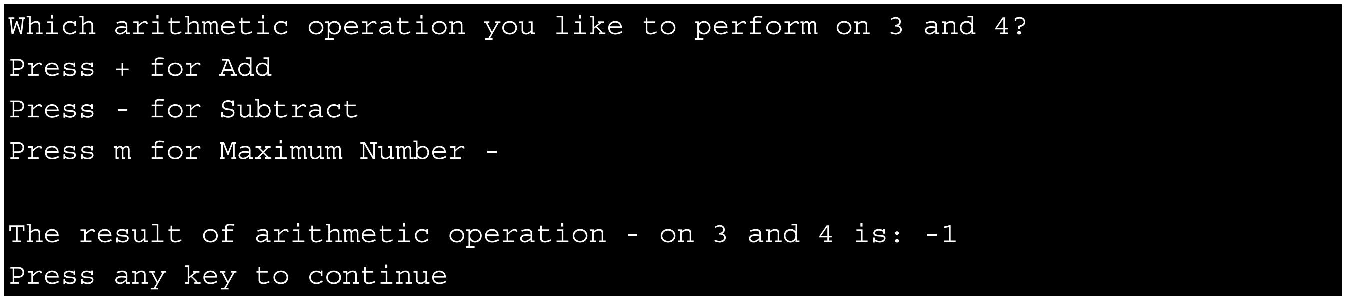
The result of the above program will be

```
The result of arithmetic operation `+' on 3 and 4 is: 7
Press any key to continue
```

The above program can be changed so that the arithmetic operation can be selected by the user.

```
using System;

namespace CSharpSchool
{
    class Test
    {
        delegate int MyDelegate(int p, int q);
        static void Main()
        {
            MyDelegate arithMethod = null;

            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 ");
            char choice = (char) Console.Read();

            switch(choice)
            {
            case '+':
               arithMethod = new MyDelegate(Add);
              break;
          case '-':
              arithMethod = new MyDelegate(Subtract);
              break;
          case 'm':
              arithMethod = new MyDelegate(Max);
              break;
            }
            int r = arithMethod(3, 4);
            Console.WriteLine("\nThe result of arithmetic operation {0} on 3 and 4 is: {1}", choice, r);
        }

        static int Add(int a, int b)
        {
            return a + b;
        }

        static int Subtract(int a, int b)
        {
            return a-b;
        }

        static int Max(int c, int d)
        {
            if(c>d)
              return c;
            else
              return d;
        }
    }
}
```

Here we have defined three methods with 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 matching 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
```

Since, in the output above, the user pressed '-', 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 to various methods as long as their signature is 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 sometimes creates confusion in the reader's mind. For the sake of clarity, we are continuously using the term 'delegate type' and 'delegate reference' and will recommend the readers to also use these.

Delegates in the .Net Framework

Although C# presents delegates as a keyword and as a first class language construct, in .Net delegates are present as a reference type, and all delegates inherit from the System.Delegate type. Hence, technically, our prior definition that said 'a delegate is a reference to a method' is not quite appropriate. A delegate is a reference type derived from System.Delegate and its instances can be used to call methods with matching signatures. 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 of this lesson.

```
class Test
{
    delegate int MyDelegate(int p, int q);
    static void Main()
    {
        MyDelegate arithMethod = null;
        ...
    }
}
```

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 called 'PerformArithOperation()', which takes two integers and a delegate reference of type MyDelegate, and calls the encapsulated method using the two integers.

```
static void PerformArithOperation(int a, int b, MyDelegate arithOperation)
{
    int r = arithOperation(a, b);
    Console.WriteLine("\nThe result of arithmetic operation on 3 and 4 is: {0}", r);
}
```

Now in the Main() method, we will call this method as

```
PerformArithOperation(3, 4, arithMethod);
```

Hence, the task of collecting and printing the result has been delegated (or transferred) to the PerformArithOperation() method. The complete source code of the program is shown below.

```
using System;
namespace CSharpSchool
{
    class Test
    {
        delegate int MyDelegate(int p, int q);
        static void Main()
        {
            MyDelegate arithMethod = null;

            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 ");
            char choice = (char) Console.Read();

            switch(choice)
            {
           case '+':
               arithMethod = new MyDelegate(Add);
              break;
          case '-':
              arithMethod = new MyDelegate(Subtract);
              break;
          case 'm':
             arithMethod = new MyDelegate(Max);
             break;
            }
            PerformArithOperation(3, 4, arithMethod);
        }

        static void PerformArithOperation(int a, int b, MyDelegate arithOperation)
        {
            int r = arithOperation(a, b);
            Console.WriteLine("\nThe result of arithmetic operation on 3 and 4 is: {0}", r);
        }

        static int Add(int a, int b)
        {
            return a + b;
        }

        static int Subtract(int a, int b)
        {
            return a-b;
        }

        static int Max(int c, int d)
        {
            if(c>d)
             return c;
            else
             return d;
        }
    }
}
```

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 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 return type of a multicast delegate type must be void". 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, a multicast delegate type must have the void return type.

Implementing a Multicast Delegate

A multicast delegate is defined in exactly the same way as a simple delegate, with the exception that the return type of a multicast delegate is strictly void.

```
delegate void MyMulticastDelegate(int p, int q);
```

The different methods are added to multicast delegate's invocation list by using '+=' assignment operator, like this:

```
MyMulticastDelegate arithMethod = null;
arithMethod = new  MyMulticastDelegate(Add);
arithMethod += new  MyMulticastDelegate(Subtract);
arithMethod += new  MyMulticastDelegate(Max);
```

The invocation of a multicast delegate is again similar to that of normal delegates and methods except that it in turn calls all the encapsulated methods.

```
arithMethod(3, 4);
```

The complete source code of this example is shown below.

```
using System;
namespace CSharpSchool
{
    class Test
    {
        delegate void MyMulticastDelegate(int p, int q);
        static void Main()
        {
            MyMulticastDelegate arithMethod = null;
            arithMethod = new  MyDelegate(Add);
            arithMethod += new  MyDelegate(Subtract);
            arithMethod += new  MyDelegate(Max);

            arithMethod(3, 4);
        }

        static void Add(int a, int b)
        {
           Console.WriteLine("The sum of 3 and 4 is: {0}", a+b);
        }

        static void Subtract(int a, int b)
        {
            Console.WriteLine("The difference of 3 and 4 is: {0}", a-b);
        }

        static void Max(int c, int d)
        {
            if(c>d)
             Console.WriteLine("The Maximum of 3 and 4 is: {0}", c);
            else
             Console.WriteLine("The Maximum of 3 and 4 is: {0}", d);
        }
    }
}
```

Note that we have changed the Add(), Subtract() and Max() methods so that they have a void return type and print out the result of their respective operations within the body of the method. The output of the program is:

```
The sum of 3 and 4 is: 7
The difference of 3 and 4 is: -1
The Maximum of 3 and 4 is: 4
Press any key to continue
```

Note that the single delegate invocation has invoked all of the encapsulated methods. This concept is used in the event handling mechanism of .Net, described later in the lesson, where each event handler method is called (when the event is fired) through the multicast delegate.

Removing a method from the multicast delegate's invocation list

Just as we can add methods to the multicast delegate's invocation list using '+=' operator, we can remove a method from the multicast delegate's invocation list using the '-=' operator. Consider the revised Main() method of the previous program shown below.

```
static void Main()
{
    Console.WriteLine("Adding 3 methods to the multicast delegate...");
    Console.WriteLine("=================");

    MyMulticastDelegate arithMethod = null;
    arithMethod = new  MyMulticastDelegate(Add);
    arithMethod += new  MyMulticastDelegate(Subtract);
    arithMethod += new  MyMulticastDelegate(Max);

    arithMethod(3, 4);

    Console.WriteLine("\nRemoving Subtract() method from the multicast delegate...");
    Console.WriteLine ("==================================);

    arithMethod -= new MyMulticastDelegate(Subtract);
    arithMethod(3, 4);
}
```

First we have added the three methods (Add(), Subtract() and Max()) to the multicast delegate 'MyMulticastDelegate' and invoked the delegate. Later, we removed the Subtract() method from the multicast delegate and invoked it again. The output of the code will be:

```
Adding 3 methods to the multicast delegate...
=============================================
The sum of 3 and 4 is: 7
The difference of 3 and 4 is: -1
The Maximum of 3 and 4 is: 4

Removing Subtract() method from the multicast delegate...
=========================================================
The sum of 3 and 4 is: 7
The Maximum of 3 and 4 is: 4
Press any key to continue
```

The output shows that the Subtract() method has been removed from the delegate's invocation list and is not called when the delegate is invoked the second time.

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 the coming of a certain time (alarm). An event is basically a message which is said to be fired or triggered when the respective action occurs. A class that raises an event is called an 'event sender', a class that receives an event is called and 'event consumer' and a 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 C# 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 the Visual Studio.Net's IDE. But as TanenBaum, the famous writer of many computer science books, writes in one of his books, "Finally, like eating spinach and learning Latin in high school, some things are considered good for you in some abstract way!"

Event Handling in C#

In .Net, events are implemented as multicast delegates. In C# 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 any class boundary. The conventional signature of a delegate for an event is:

```
public void EventDelegate(object sender, EventArgs e)
```

2.Define a class to generate or raise the event. Define a public event in the class using the event keyword and the public delegate:

```
public event EventDelegate MyEvent
```

Write some logic to raise the event. When raising an event, the first argument is usually the sender or originator of the event. The second argument is a sub-type of System.EventArgs, which holds any additional data to be passed to the event handler.

```
  class SomeEventArgs : EventArgs
    {
        ...
    }
```

An event is generally raised like this:

```
    SomeEventArgs someData = new SomeEventArgs(/*some necessary arguments*/);
    MyEvent(this, someData);
```

Or if no data needs to be sent, the event is raised like this:

```
    MyEvent(this, null);
```

3.Define a class to receive the events. This 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 identical to that of the public delegate created in step 1. The name of the event handler method conventionally starts with the word "On", e.g.

```
public void OnMyEvent(object sender, EventArgs e)
  {
      // handle the event
  }
```

Instantiate the event generator class created in step 2 like this:

```
EventClass eventObj = new EventClass();
```

Add the event handler written in the current class to the event generator class' event.

```
eventObj.MyEvent += new EventDelegate(OnMyEvent);
```

Now the event handler 'OnMyEvent()' will be called automatically whenever the event 'MyEvent' is triggered.

A Clock Timer Example

To help understand how events are implemented and received, let's look at 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':

```
    public delegate void TimerEvent(object sender, EventArgs e);
```

Now we define a class named 'ClockTimer' to generate the event.

```
class ClockTimer
{
    public event TimerEvent Timer;

    public void Start()
    {
        for(int i=0; i<5; i++)
        {
            Timer(this, null);
            Thread.Sleep(1000);
        }
    }
}
```

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. Here, we have used the Sleep() method of the System.Threading.Thread class, which takes the number of milliseconds the current thread will be suspended as its argument. We will explore threading and its issues in coming lessons.

Next we need to define a class that will receive and consume the event, which is defined as:

```
class Test
{
    static void Main()
    {
        ClockTimer clockTimer = new ClockTimer();
        clockTimer.Timer += new TimerEvent(OnClockTick);
        clockTimer.Start();
    }

    public static void OnClockTick(object sender, EventArgs e)
    {
        Console.WriteLine("Received a clock tick event!");
    }
}
```

The class contains an event handler method, 'OnClockTick()', which follows the ClockEvent delegate's signature. In the Main() method of the class, 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 have called the Start() method, which will start the process of generating events in the ClockTimer class. The complete source code of the program is shown below.

```
using System;
using System.Threading;

namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            ClockTimer clockTimer = new ClockTimer();
            clockTimer.Timer += new TimerEvent(OnClockTick);
            clockTimer.Start();
        }

        public static void OnClockTick(object sender, EventArgs e)
        {
            Console.WriteLine("Received a clock tick event!");
        }
    }

    public delegate void TimerEvent(object sender, EventArgs e);

    class ClockTimer
    {
        public event TimerEvent Timer;

        public void Start()
        {
            for(int i=0; i<5; i++)
            {
              Timer(this, null);
              Thread.Sleep(1000);
            }
        }
    }
}
```

Note that we have also included the System.Threading namespace at 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 C#, we can subscribe multiple event handlers to a single event. For example, consider this revised Test class:

```
class Test
{
    static void Main()
    {
        ClockTimer clockTimer = new ClockTimer();
        clockTimer.Timer += new TimerEvent(OnClockTick);
        clockTimer.Timer += new TimerEvent(OnClockTick2);
        clockTimer.Start();
    }

    public static void OnClockTick(object sender, EventArgs e)
    {
        Console.WriteLine("Received a clock tick event!");
    }

    public static void OnClockTick2(object sender, EventArgs e)
    {
        Console.WriteLine("Received a clock tick event in OnClockTick2!");
    }
}
```

Here we have introduced another event handler, 'OnClockTick2', and have subscribed it also to the Timer event in the Main() method using the '+=' operator. 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 above, now both the OnClockTick() and OnClockTick2() are invoked each time the event is raised.

Passing some 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 also contains the sequence number of the clock ticks. First we need to define a new class 'ClockTimerArgs', which inherits from the System.EventArgs class.

```
public class ClockTimerArgs : EventArgs
{
    private int tickCount;
    public ClockTimerArgs(int tickCount)
    {
        this.tickCount = tickCount;
    }
    public int TickCount
    {
        get { return tickCount; }
    }
}
```

The ClockTimerArgs class contains a private variable named 'tickCount' 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 to:

```
public delegate void TimerEvent(object sender, ClockTimerArgs e);
```

The argument type in the delegate is changed from EventArgs to 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 TimerEvent Timer;

    public void Start()
    {
        for(int i=0; i<5; i++)
        {
            Timer(this, new ClockTimerArgs(i+1));
            Thread.Sleep(1000);
        }
    }
}
```

The only change in this class is that instead of passing null as the second argument, we are passing a new object of the ClockTimerArgs type with the sequence number of the current clock tick. Finally, the event handler is written:

```
public static void OnClockTick(object sender, ClockTimerArgs e)
{
    Console.WriteLine("Received a clock tick event. This is clock tick number {0}", e.TickCount);
}
```

Here we have simply printed the clock tick number using the ClockTimerArgs' TickCount Property. The complete source code is shown below.

```
using System;
using System.Threading;

namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            ClockTimer clockTimer = new ClockTimer();
            clockTimer.Timer += new TimerEvent(OnClockTick);
            clockTimer.Start();
        }

        public static void OnClockTick(object sender, ClockTimerArgs e)
        {
            Console.WriteLine("Received a clock tick event. This is clock tick number {0}", e.TickCount);
        }
    }

    public class ClockTimerArgs : EventArgs
    {
        private int tickCount;
        public ClockTimerArgs(int tickCount)
        {
            this.tickCount = tickCount;
        }
        public int TickCount
        {
            get { return tickCount; }
        }
    }
    public delegate void TimerEvent(object sender, ClockTimerArgs e);

    class ClockTimer
    {
        public event TimerEvent Timer;
        public void Start()
        {
            for(int i=0; i<5; i++)
            {
                Timer(this, new ClockTimerArgs(i+1));
                Thread.Sleep(1000);
            }
        }
    }
}
```

When the above program 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.

10. WinForms & Windows Applications

Lesson Plan

Today we will start building Windows Applications in C#. We will start by looking at the architecture of Windows Application 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

C# 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 'WinForm' for short. A windows form may contain text labels, push buttons, text boxes, list boxes, images, menus and 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:

```
class MyForm : System.Windows.Form
{
     ...
}
```

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
    {
        static void Main()
        {
            Application.Run(new MyForm());
        }
    }
```

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:

```
using System;
using System.Windows.Forms;
using System.Drawing;

namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            Application.Run(new MyWindow());
        }
    }

    class MyWindow : Form
    {
        public MyWindow() : base()
        {
            this.Text = "My First Windows Application";
            this.Size = new Size(300, 300);

            Label lblGreeting = new Label();
            lblGreeting.Text = "Hello WinForm";
            lblGreeting.Location = new Point(100, 100);

            this.Controls.Add(lblGreeting);
        }
    }
}
```

Understanding the Code

At the start, we included three namespaces in our application:

```
using System;
using System.Windows.Forms;
using System.Drawing;
```

The System namespace, as we stated in the first lesson, is the necessary ingredient of all C# 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. Form, Label and Button. Finally, including the System.Drawing namespace is necessary as it contains the classes related to the drawing of controls. The Size and Point classes used later in the program are actually defined in the System.Drawing namespace.

Later, we derived a new class, 'MyWindow', from the Form class defined in System.Windows.Forms.

```
    class MyWindow : Form
    {
    ...
    }
```

In the constructor of MyWindow, 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.

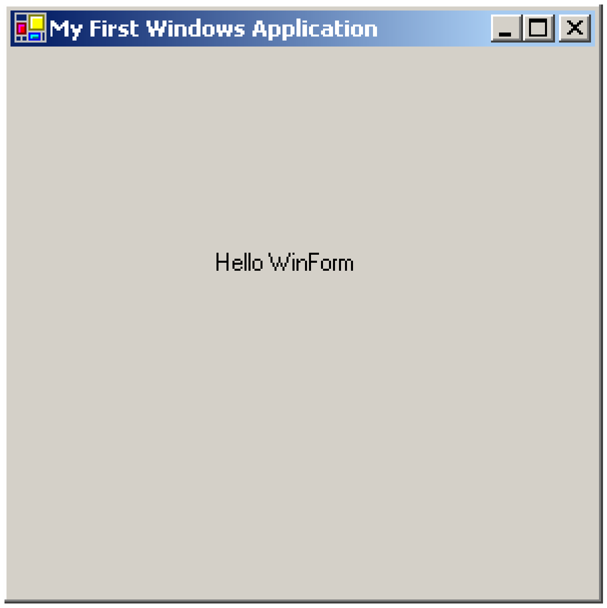
```
    public MyWindow() : base()
    {
        this.Text = "My First Windows Application";
        this.Size = new Size(300, 300);
```

Next 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 MyWindow() : base()
    {
        this.Text = "My First Windows Application";
        this.Size = new Size(300, 300);

        Label lblGreeting = new Label();
        lblGreeting.Text = "Hello WinForm";
        lblGreeting.Location = new Point(100, 100);

        this.Controls.Add(lblGreeting);
    }
```

Finally, we have created a Test class containing the Main() method. In the Main() method, we have instantiated the MyWindow 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

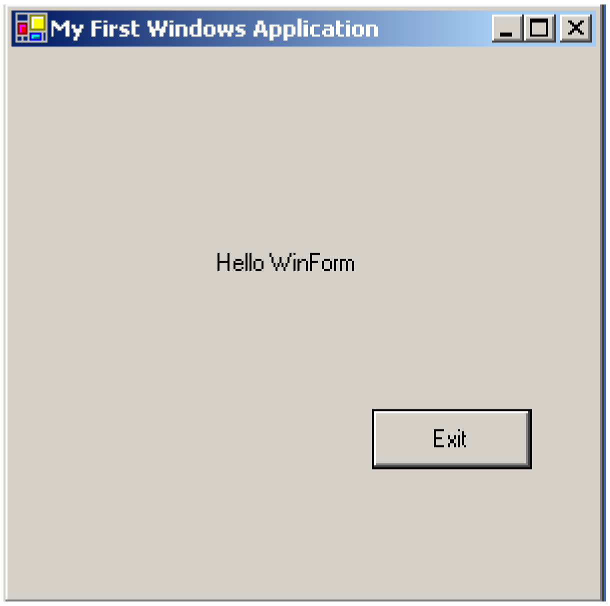
Let's now add a button labeled 'Exit' to the form. The 'Exit' button will close the application when it is clicked. In .Net, Push Buttons are instances of the System.Windows.Forms.Button class. To associate some action with the button click, we need to create an event handler and register (or add) it to the Button's Click event. Below is the code for this application.

```
using System;
using System.Windows.Forms;
using System.Drawing;

namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            Application.Run(new MyWindow());
        }
    }

    class MyWindow : Form
    {
        public MyWindow() : base()
        {
            // Form
            this.Text = "My First Windows Application";
            this.Size = new Size(300, 300);
            this.StartPosition = FormStartPosition.CenterScreen;

            // Label
            Label lblGreeting = new Label();
            lblGreeting.Text = "Hello WinForm";
            lblGreeting.Location = new Point(100, 100);

            // Button
            Button btnExit = new Button();
            btnExit.Text = "Exit";
            btnExit.Location = new Point(180, 180);
            btnExit.Size = new Size(80, 30);
            btnExit.Click += new EventHandler(BtnExitOnClick);

            // Adding controls to Form
            this.Controls.AddRange(new Control[] {lblGreeting, btnExit});
        }

        public void BtnExitOnClick(object sender, EventArgs e)
        {
            Application.Exit();
            }
        }
}
```

In the constructor of MyWindow, first 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. Later, 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 C#, see lesson 10 of the C# 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 form class to add an array of controls to the Controls collection of form. This method takes an array of type Control as its parameter.

When the code is run, the following window will be displayed:



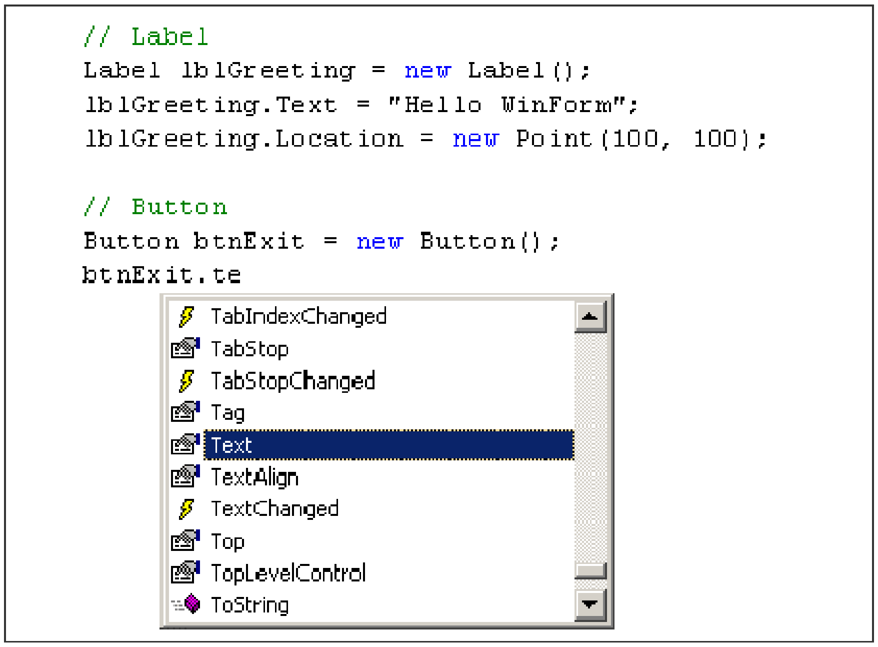
Now you can press either the Exit Button or the close button at title bar to exit the application.

Visual Studio.Net & its IDE (Integrated Development Environment)

Most of the time, you will be using Visual Studio.Net to develop Windows applications in C#. 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 IDE for all .Net applications, along with a standard debugger, project and solution settings, form designer, integrated compiler and lot of other useful tools.

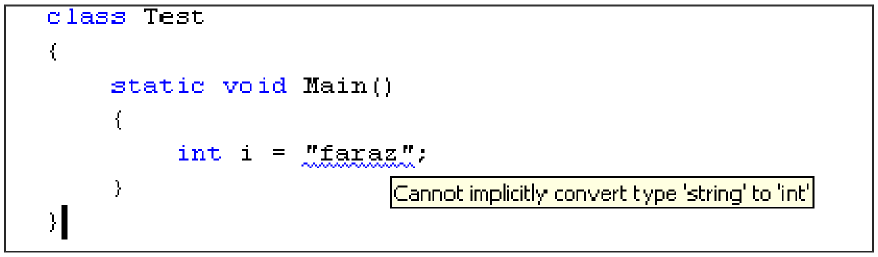
IntelliSense and Hot Compiler

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 in the programming context. 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.



```
// Label
Label lblGreeting = new Label();
lblGreeting.Text = "Hello WinForm";
lblGreeting.Location = new Point(100, 100);

// Button
Button btnExit = new Button();
btnExit.te
```

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 at work in Visual Studio.Net.

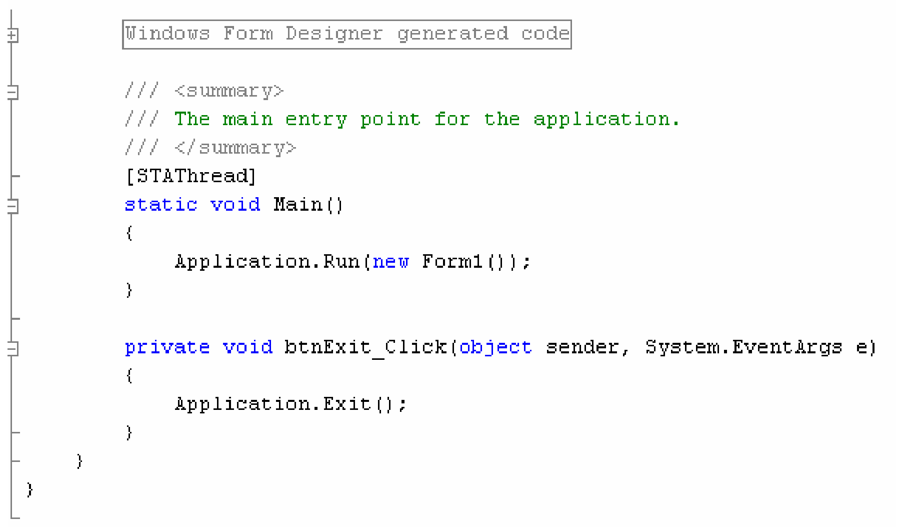


```
class Test
{
    static void Main()
    {
        int i = "faraz";
    }
}
```

Cannot implicitly convert type 'string' to 'int'

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 the region. The region is defined using the #region...#endregion preprocessor directives.



```
        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();
        }
    }
}
```

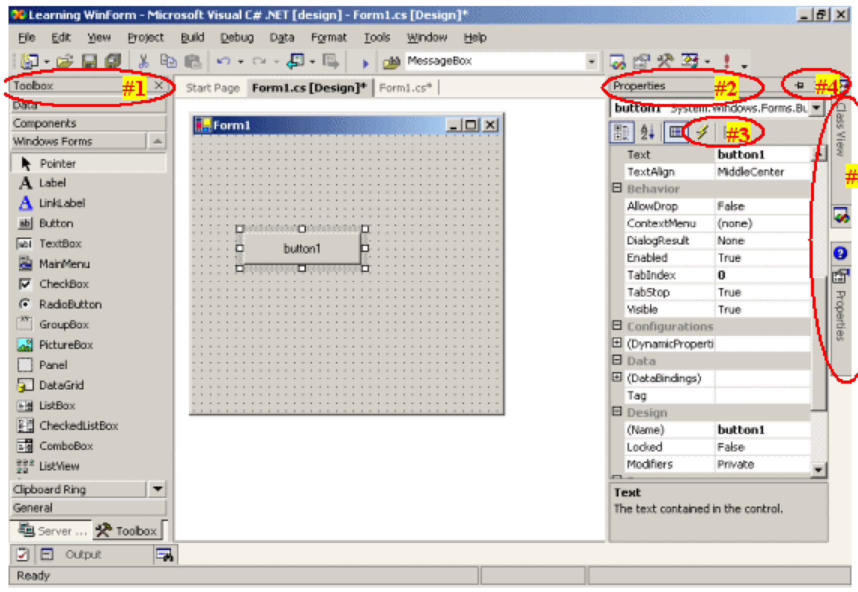
Integrated Compiler, Solution builder and Debugger

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 just by placing the controls on the form from the Toolbox. You can set a lot of 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 change 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 windows.



button1 System.Windows.Forms.Bu

Text	button1
TextAlign	MiddleCenter
Behavior	
AllowDrop	False
ContextMenu	(none)
DialogResult	None
Enabled	True
TabIndex	0
TabStop	True
Visible	True
Configurations	
(DynamicProperti	
Data	
(DataBindings)	
Tag	
Design	
(Name)	button1
Locked	False
Modifiers	Private

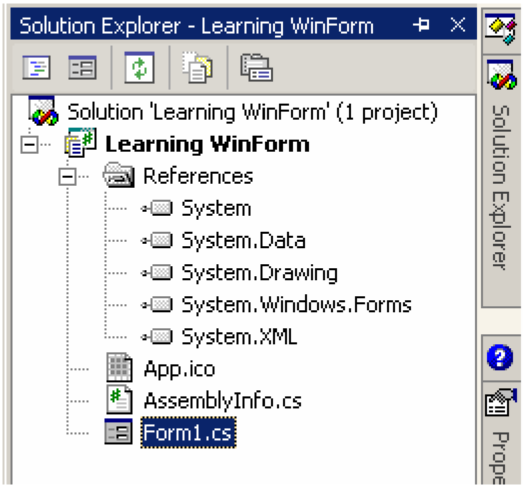
Text
The text contained in the control.

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. Event properties can be changed by switching to the Event Poperties 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 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 receive a mouse click), and hide when they lose focus. A window can be set for auto hide by the button marked #4 in the above figure. The hidden windows are always accessible through the left and right hand panes of the form designer window. The right hand pane is marked with #5 in the above figure and has got the class view, help and solution explorer windows in the 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 standard 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 like VB.Net, VC#.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. 'App.ico' is the icon file for the application. AssemblyInfo.cs is a C# file that contains information about the current assembly. Form1.cs in the above figure is the name of the source file of the program.

A .Net solution is saved in a .sln file, a C# project is saved in a .csproj file and C# source code is saved in a .cs file. It is important to understand here that Projects and Solutions are standards of Visual Studio.Net and are not the requirement of any .Net language. In fact, the language compiler is not even aware of any project or solution.

Menus in the Visual Studio .Net IDE

File Menu: Used to create, open, save and close the project, solution or individual source files.

Edit Menu: Used for text editing and searching in 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 coming lessons).

Build Menu: This menu is used to compile and build the source file, project or solution. 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 program, and a debugger helps make this process 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 and their layout in the Form Designer view.

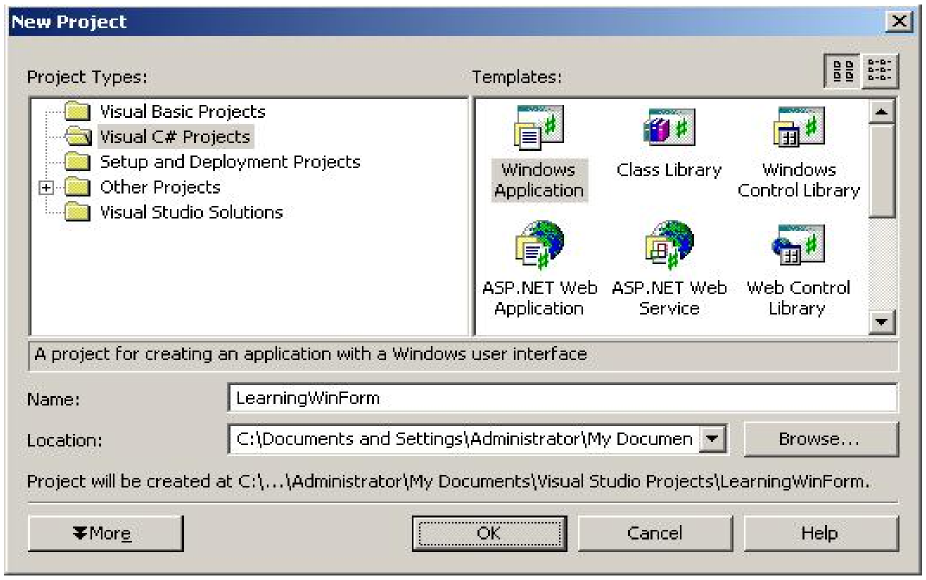
Tools Menu: Provides the access to various useful Visual Studio.Net tools.

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 C# Windows Application Project. For this, start Visual Studio.Net and click File'New'Project. It will show the following screen:



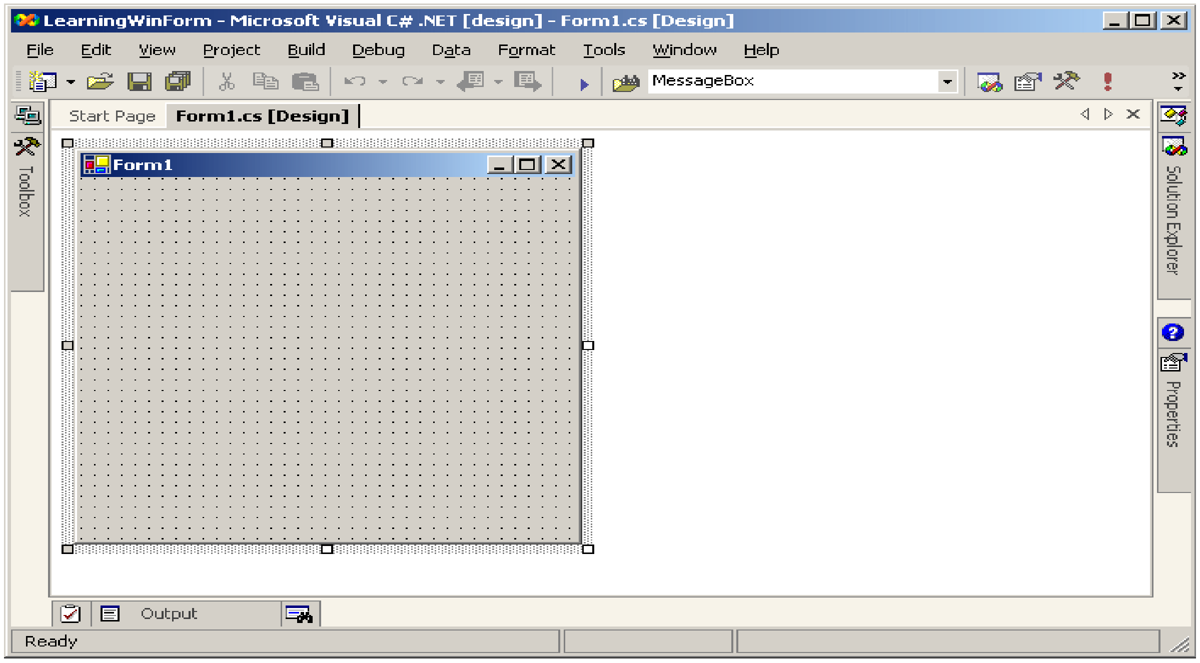
A project for creating an application with a Windows user interface

Name: LearningWinForm

Location: C:\Documents and Settings\Administrator\My Documen

Project will be created at C:\...\Administrator\My Documents\Visual Studio Projects\LearningWinForm.

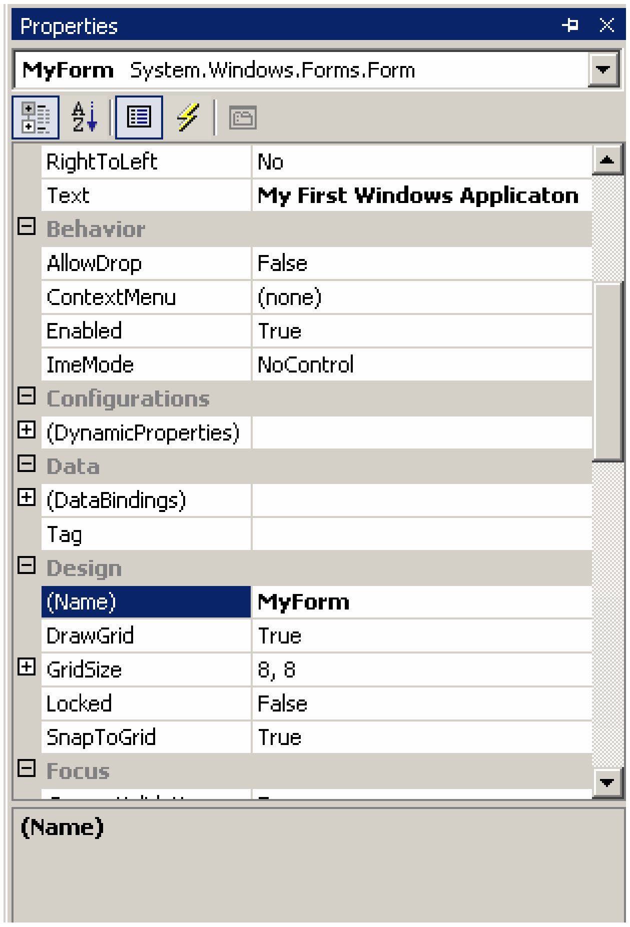
From the above screen, select 'Visual C# Projects' in Project types and 'Windows Application' in Templates. Write the name of the new project ('LearningWinForm' in the above figure) in the text box labeled Name. Select the location where you wish to store the project using the Browse... Button and click OK. It will show you an empty form in the designer view similar to the figure below:


LearningWinForm - Microsoft Visual C# .NET [design] - Form1.cs [Design]
File Edit View Project Build Debug Data Format Tools Window Help
MessageBox
Start Page
Form1.cs [Design]
Form1
Toolbox
Solution Explorer
Properties
Output
Ready

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

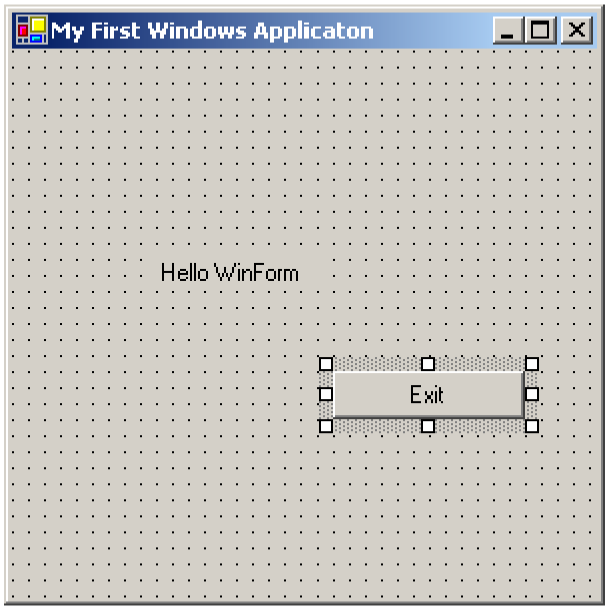
You can change the default properties of the form using the Properties window. For this, select (click) the form and select the properties window (If the properties window is not visible in the right hand pane, select View'Properties Window). Now change the title of the form and the name of the form's class in the code by changing the Text and Name properties, as shown in the following figure.



RightToLeft	No
Text	**My First Windows Applicaton**
Behavior	
AllowDrop	False
ContextMenu	(none)
Enabled	True
ImeMode	NoControl
Configurations	
(DynamicProperties)	
Data	
(DataBindings)	
Tag	
Design	
(Name)	**MyForm**
DrawGrid	True
GridSize	8, 8
Locked	False
SnapToGrid	True
Focus	

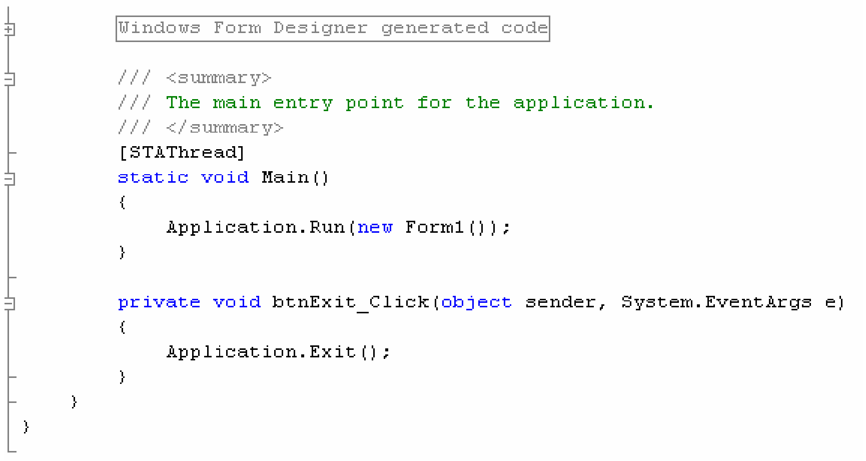
Adding Controls to the Form

Now select the Label control from the toolbox, place it on the form and resize it as appropriate. Select (click) the label on the form and from 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. Now 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

Now we need to add the event handling code for the Exit button. For this, simply double click the Exit button in the designer. This will create a new event handler for the Exit button's Click event and take you to the source code as shown in the following figure.

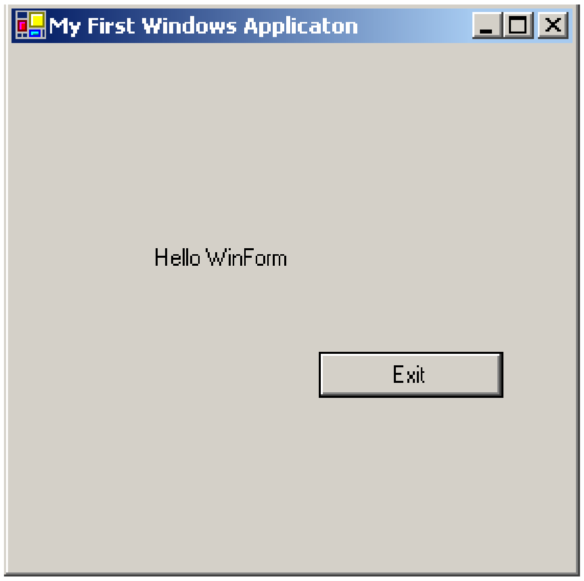


```
        Windows Form Designer generated code

        /// <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 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.

Author's Note: Unfortunately, some builds of Visual Studio.Net do not change the name of the form (from Form1 to MyForm in our case) in the call to the Application.Run() method in the Main() method. You may have to explicitly change the Main() method like this:

```
    static void Main()
    {
        Application.Run(new MyForm());
    }
```

Make sure to change the name of the form in Main() method whenever you change its name in the Properties Window in order to avoid getting a compilation error.

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.

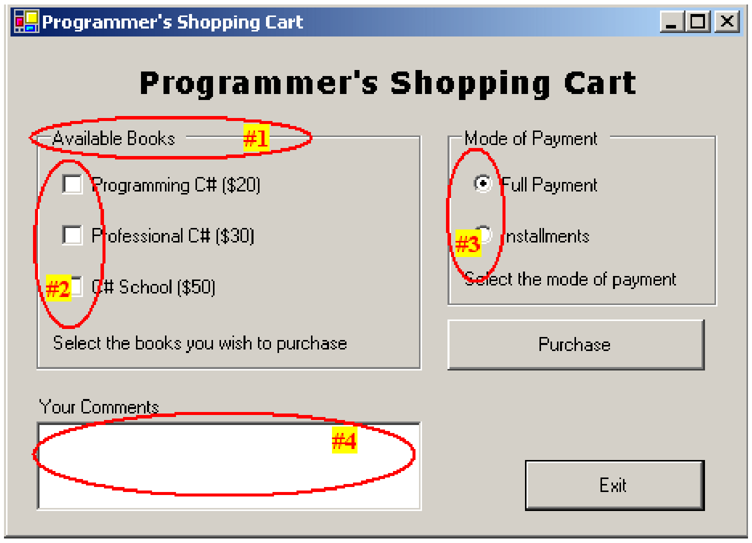
Using More Controls

Now we are getting 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 the form. Instance of System.Windows.Forms.Label. Important properties are Text, Name and Font. Usually events are not handled for the Label control.
Button	Used to display a Push Button on the form. Instance of System.Windows.Forms.Button. 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. Instance of System.Windows.Forms.TextBox. 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. Instance of System.Windows.Forms.GroupBox. 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 many available 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. 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 some push button or actions on other controls.
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, all or none of the checkboxes in a group can be selected. Instance of System.Windows.Forms.CheckBox. 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

Now 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 on both full payment and on installments. The application will finally look like this:



As you can see, we have used the GroupBox (#1), CheckBoxe (#2), RadioButtons (#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

Where StartPosition is the startup position of the form on the screen, Name is the name of form class generated by the designer and Text is the title of the Application's Window.

Now add a Label to the form as shown in the figure above. Set the Text property of the Label to 'Programmer's Shopping Cart'.

Next, 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 'Programming C# ($20)', 'Professional C# ($30)' and 'C# School ($50)'. Also set the Name property of the checkboxes to 'cbxProgCS', 'cbxProfCS' and 'cbxCSSchool'. 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, and 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 that spans 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 void btnExit_Click(object sender, System.EventArgs e)
{
    Application.Exit();
}
```

When the user has selected any book, the 'Mode of Payment' group box and the Purchase button should be enabled, and 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, either double click the checkbox in the designer or double click the 'CheckedChanged' event of checkbox in the Event property window. The CheckedChanged event of a checkbox is triggered whenever the checkbox is checked or unchecked or its Checked property is changed. Write the following code in the event handler of first checkbox:

```
private void cbxProgCS_CheckedChanged(object sender, System.EventArgs e)
{
    if(cbxProgCS.Checked == false && cbxProfCS.Checked == false && cbxCSSchool.Checked == false)
    {
        gbxPaymentMode.Enabled = false;
        btnPurchase.Enabled = false;
    }
    else
    {
        gbxPaymentMode.Enabled = true;
        btnPurchase.Enabled = true;
    }
}
```

In the code above if all the checkboxes are unchecked, we disable the 'Mode of Payment' group box and the Purchase button; otherwise, we enable them.

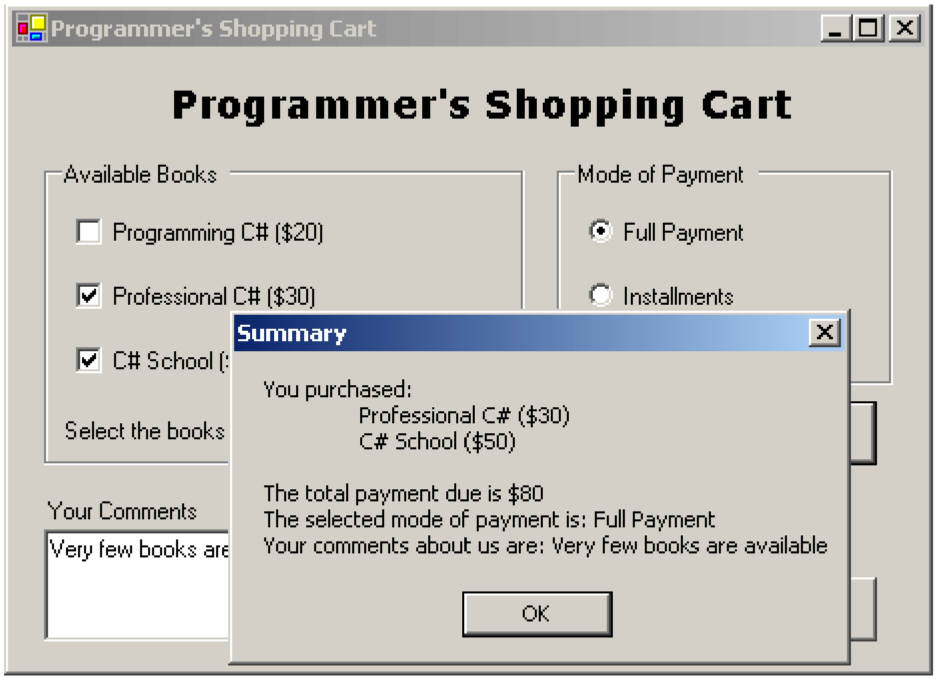
Copy and paste the same code for the CheckedChanged event of the other two checkboxes (cbxProfCS and cbxCSSchool)   Now execute the program and check/uncheck different checkboxes. You will notice if any of the checkboxes are checked, the Mode of Payment group box and Purchase button are enabled, and if none of the checkboxes are checked, they are disabled.

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 write the following code:

```
    private void btnPurchase_Click(object sender, System.EventArgs e)
    {
        string message = "You purchased:\r\n\t";
        int amount=0;
        if(cbxProgCS.Checked)
        {
            amount+=20;
            message+=cbxProgCS.Text + "\r\n\t";
        }
        if(cbxProfCS.Checked)
        {
            amount+=30;
            message+=cbxProfCS.Text + "\r\n\t";
        }
        if(cbxCSSchool.Checked)
        {
            amount+=50;
            message+=cbxCSSchool.Text + "\r\n\t";
        }

        string paymentMode="";
        if(rbnFullPayment.Checked)
        {
            paymentMode = rbnFullPayment.Text;
        }
        else
        {
            paymentMode = rbnInstallments.Text;
        }

        message+="\r\nThe total payment due is $" + amount.ToString();
        message+="\r\nThe selected mode of payment is: " + paymentMode;

        if(txtComments.Text != "")
        {
            message+="\r\nYour comments about us are: " + txtComments.Text;
        }

        MessageBox.Show(message, "Summary");
    }
```

We have used three variables in the above code. The integer variable amount is used to hold the total amount of purchase, the string variable paymentMode is used to hold the selected mode of payment and 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 build 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' static 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.

Now when we execute the program and press the purchase button after selecting some books, we see the following output:



You purchased:
Professional C# ($30)
C# School ($50)

The total payment due is $80
The selected mode of payment is: Full Payment
Your comments about us are: Very few books are available

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 button on the keyboard.
- 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 (cbxProgCS), RadioButton with rbn (rbnFullPayment) and GroupBox with gbx (gbxPaymentMode).

11. More Windows Controls & Standard Dialog Boxes

Lesson Plan

Today 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 List View controls. Later, we will learn about other common controls, including the main menu, image list, Toolbar and 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.

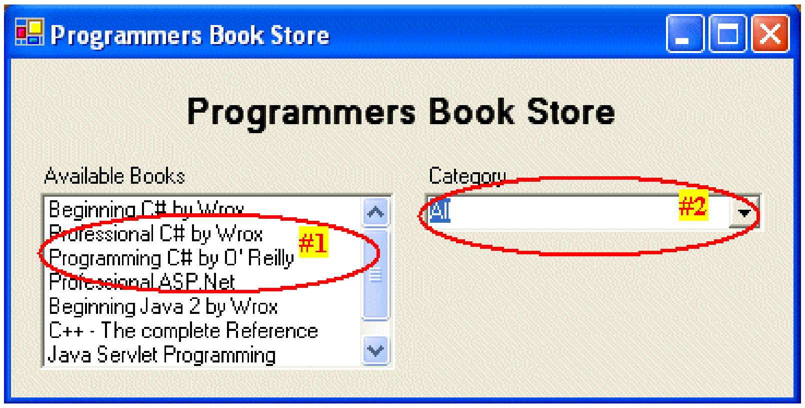
Collection Controls

Some of the windows controls contain some type of collection, like 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. 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 of 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 indexer property that allows items in the list box to be accessed programmatically (through the code).

```
    lbxBooks.Items[0] = "Program Development in Java";  // changing list box item
    string book1 = (string) lbxBooks.Items[1];           // reading list box item
```

In the above code, the first statement uses the indexer 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.

You may also get, 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

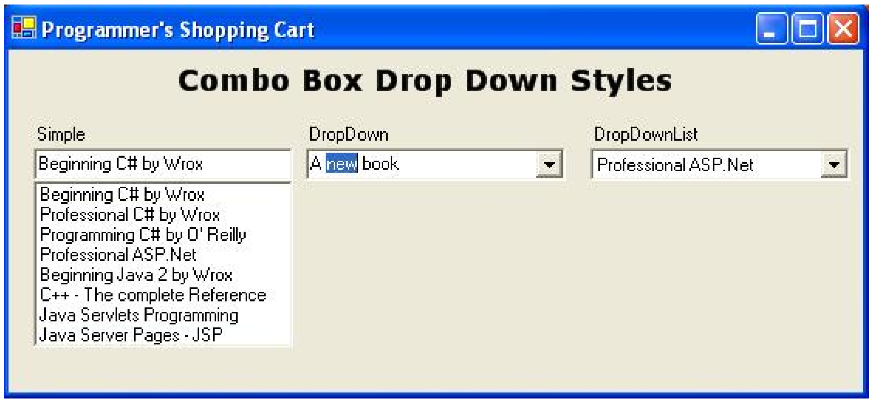
The most important and frequently used event of the list box is SelectedIndexChanged. This event is triggered when an item is selected in the list box. To add an event handler for this event, double click the list box in the form designer.

The following code will show the selected item in the message box whenever the selection in the list box is changed.

```
private void lbxBooks_SelectedIndexChanged(object sender, System.EventArgs e)
{
    MessageBox.Show("The selected item is " + lbxBooks.SelectedItem.ToString(), "Selected Item");
}
```

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.

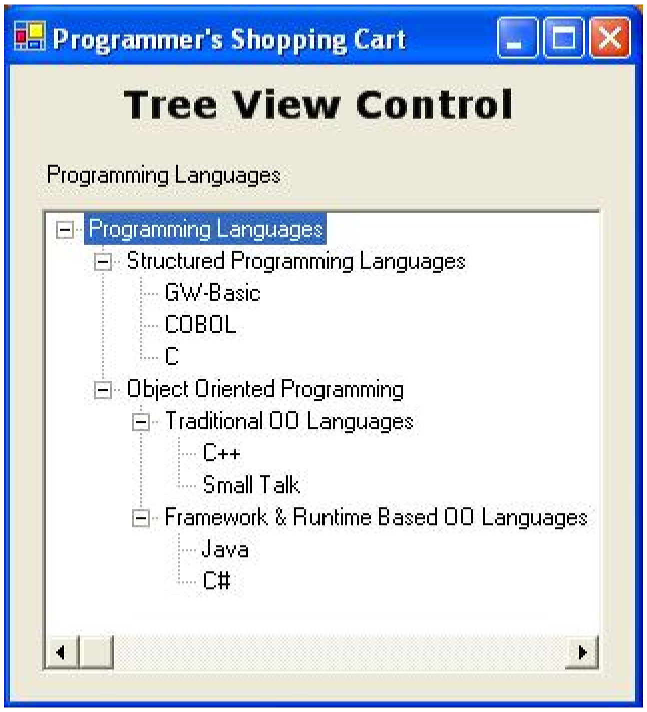


The Simple style shows the list of items and the selected item at the same time in separate areas. The simple drop down style combo box is typically used in the font dialog box (discussed later in the lesson). The DropDown style shows the items in a drop down list. The user can write a new item in its 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 provide a new item himself.

Items can be inserted, removed and accessed in just the same way they were in the list box.

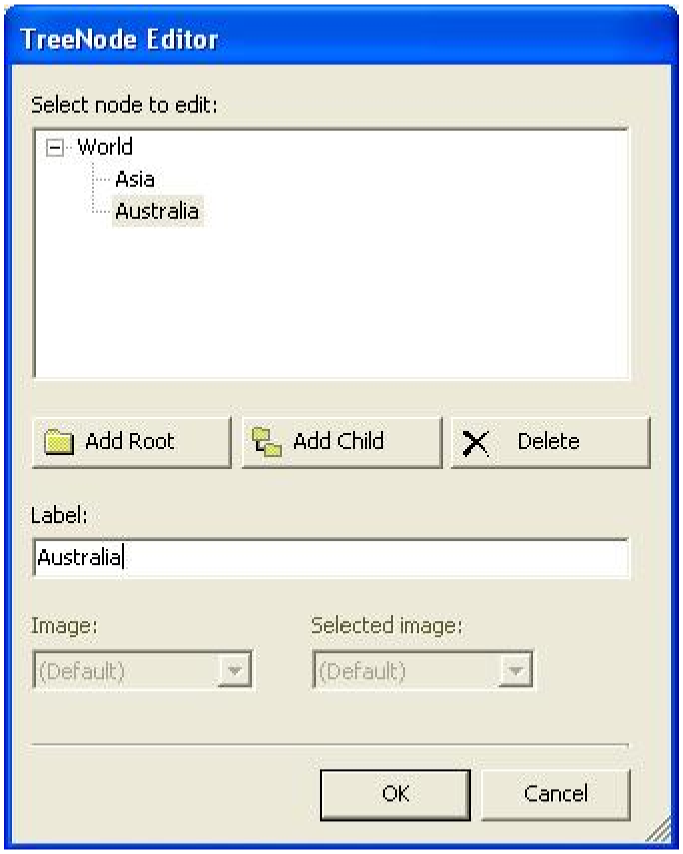
Tree View

The Tree View Control is used for hierarchical representations of items.It is represented in .Net by the ystem.Windows.Forms.TreeView class. The following screen shot gives an example of how the tree view control can look.


Programmer's Shopping Cart
Tree View Control
Programming Languages
Programming Languages
Structured Programming Languages
GW-Basic
COBOL
C
Object Oriented Programming
Traditional OO Languages
C++
Small Talk
Framework & Runtime Based OO Languages
Java
C#

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.



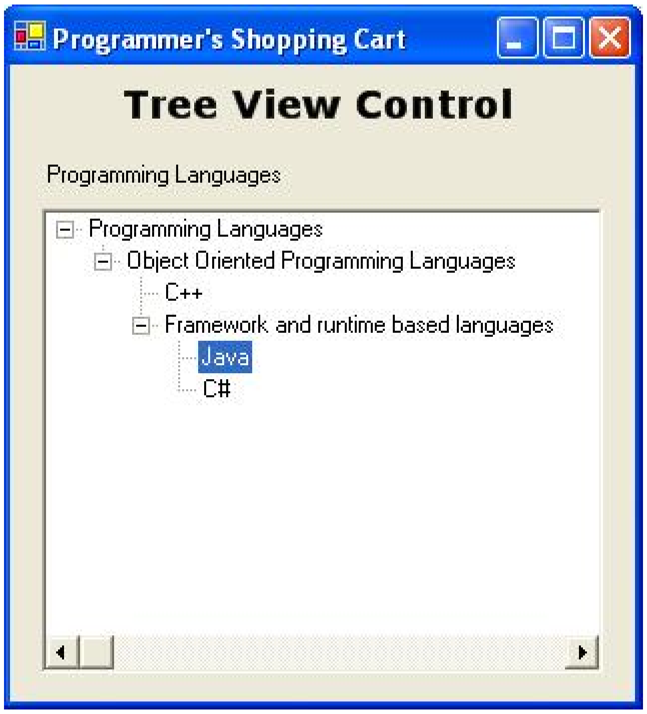
'Add Root' will add a root element (one with no parent e.g., World in the above window). 'Add Child' will add a child node to the selected element. The label of the nodes can be changed using the text box labeled 'Label'. A node can be deleted using the 'Delete' button.

Adding/Removing items at runtime

Items can be added to or removed from the tree view using the TreeNode editor at design time, but 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 void MyForm_Load(object sender, System.EventArgs e)
{
    treeView1.Nodes.Clear();
    treeView1.Nodes.Add("Programming Languages");
    TreeNode node = new TreeNode("Object Oriented Programming Languages");
    node.Nodes.Add("C++");
    TreeNode subnode = node.Nodes.Add("Framework and runtime based languages");
    subnode.Nodes.Add("Java");
    subnode.Nodes.Add("C#");
    treeView1.Nodes[0].Nodes.Add(node);
}
```

First 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.


Programmer's Shopping Cart
Tree View Control
Programming Languages
Programming Languages
Object Oriented Programming Languages
C++
Framework and runtime based languages
Java
C#

Tree View Events

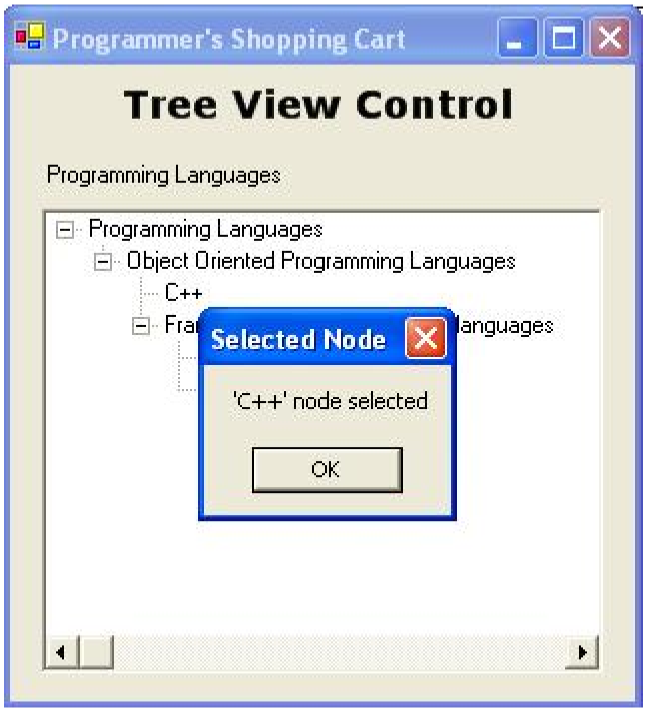
Tree View has a number of important events, some of which are listed below.

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 like ByKeyboard, ByMouse, Collapse, etc 2) 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 void treeView1_AfterSelect(object sender, System.Windows.Forms.TreeViewEventArgs e)
{
    MessageBox.Show("'" + e.Node.Text + "' node selected", "Selected Node");
}
```

The program will look like this when run:



'C++' node selected

Image List Control

The image list is an invisible control. It is merely used to store the 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 it 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.

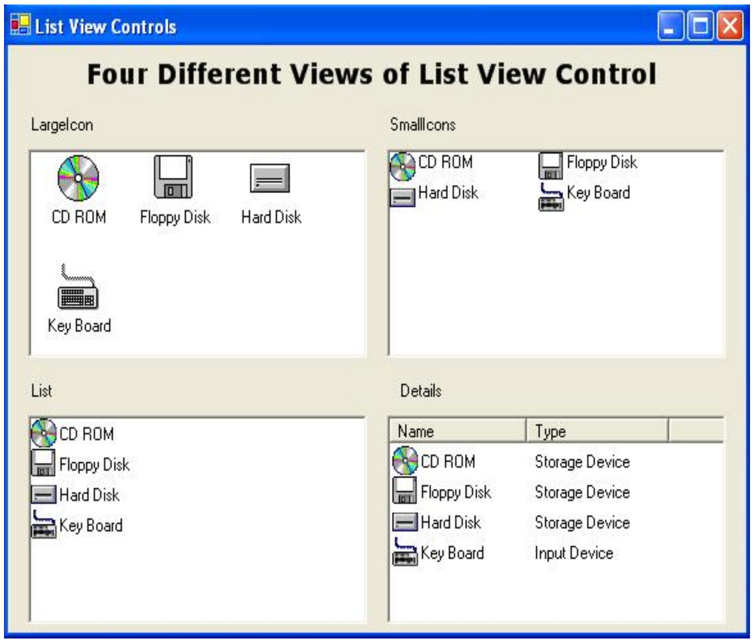
The fundamental property of the image list is its Name. An Image List has relatively few properties. The most important is the 'Images' collection, which holds the images stored in the image list. When you click the Images property of the Image List in Visual Studio's designer, it shows an image collection editor which allows you to add different .bmp, .jpg and .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.



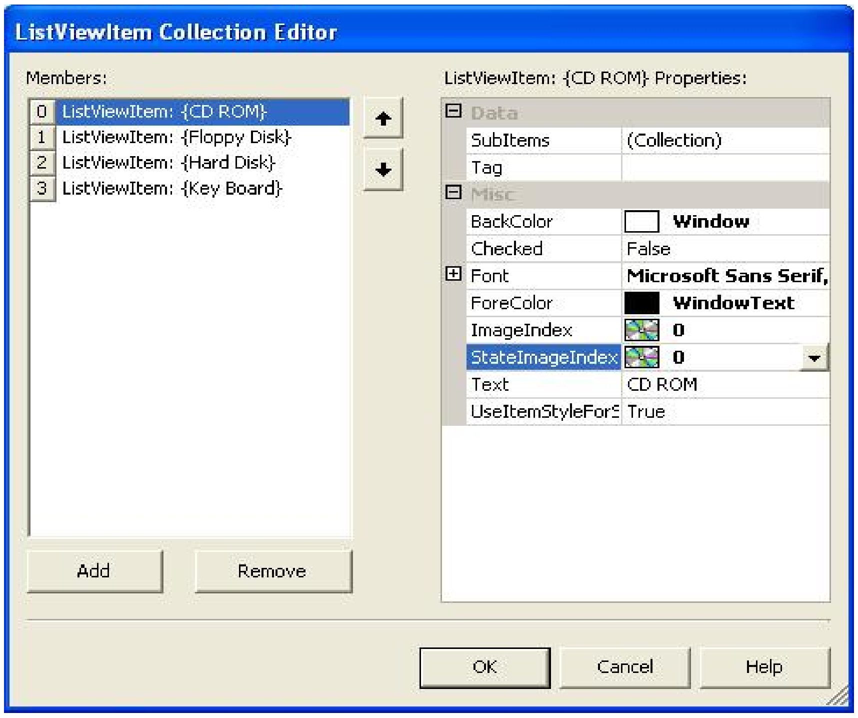
Name	Type
CD ROM	Storage Device
Floppy Disk	Storage Device
Hard Disk	Storage Device
Key Board	Input Device

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 in small icon, list and detail view. Usually the two image lists are the same with the difference only in their ImageSize property. For example in the above screen shot, the size of images in the large image list is 32, 32 while the size of images in the small image list is 20, 20 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 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.



0 ListViewItem: {CD ROM}
1 ListViewItem: {Floppy Disk}
2 ListViewItem: {Hard Disk}
3 ListViewItem: {Key Board}

ListViewItem: {CD ROM} Properties:

Data	
SubItems	(Collection)
Tag	
Misc	
BackColor	Window
Checked	False
Font	Microsoft Sans Serif,
ForeColor	WindowText
ImageIndex	0
StateImageIndex	0
Text	CD ROM
UseItemStyleForS	True

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 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 this 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 to 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
ListViewItem item = new ListViewItem("Disk", 2);// a new ListViewItem object is created
listView1.Items.Add(item);                          // and added to the list view control
```

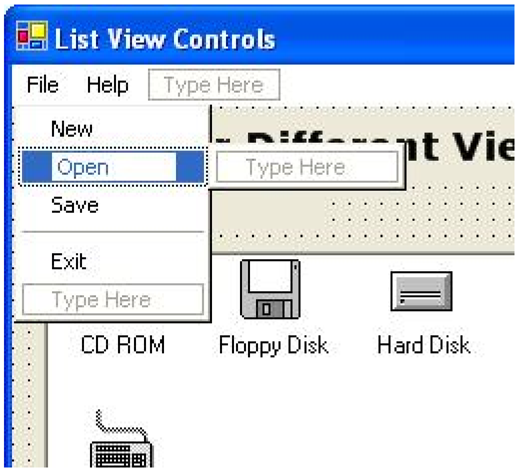
Events for List View Control

The most frequently used event for 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 void listView1_SelectedIndexChanged(object sender, System.EventArgs e)
{
    foreach(ListViewItem item in listView1.SelectedItems)
    {
        MessageBox.Show(item.Text);
    }
}
```

Main Menu

Main menu is also added to the form as a resource. 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.



Some important points about main menus are:

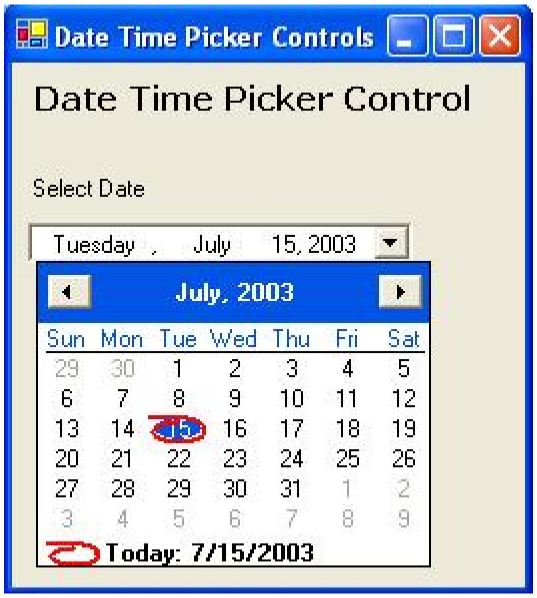
- You can change the name and text of the main menu item from the properties window while selecting it in the designer.
- You can also apply shortcut keys (like Ctrl+t or Alt+F5) for the menu items using the properties window.
- You can make a dashed line (just like the one between Save and Exit option 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

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 with 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

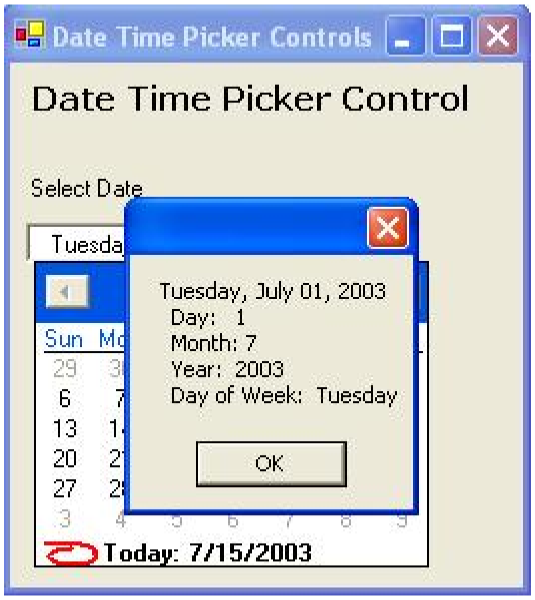
Date Time Picker control is commonly used in the windows environment to allow the user select a particular date. In .Net it is represented by the System.Windows.Forms.DateTimePicker class. It can simply be selected from the toolbox and placed on the form in the designer. Usually we only change its Name property. The following screen shot shows a form containing date time picker control.



The most frequently used event for 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 the message box

```
private void dtpSelectDate_ValueChanged(object sender, System.EventArgs e)
{
    string msg = dtpSelectDate.Text;
    msg += "\r\n  Day:   " + dtpSelectDate.Value.Day.ToString();;
    msg += "\r\n  Month: " + dtpSelectDate.Value.Month.ToString();;
    msg += "\r\n  Year:  " + dtpSelectDate.Value.Year.ToString();;
    msg += "\r\n  Day of Week:  " + dtpSelectDate.Value.DayOfWeek.ToString();;
    MessageBox.Show(msg);
}
```

In the code above, we have separately collected and printed the day, month, year and day of the week in the message box. When the program is executed following output is resulted.



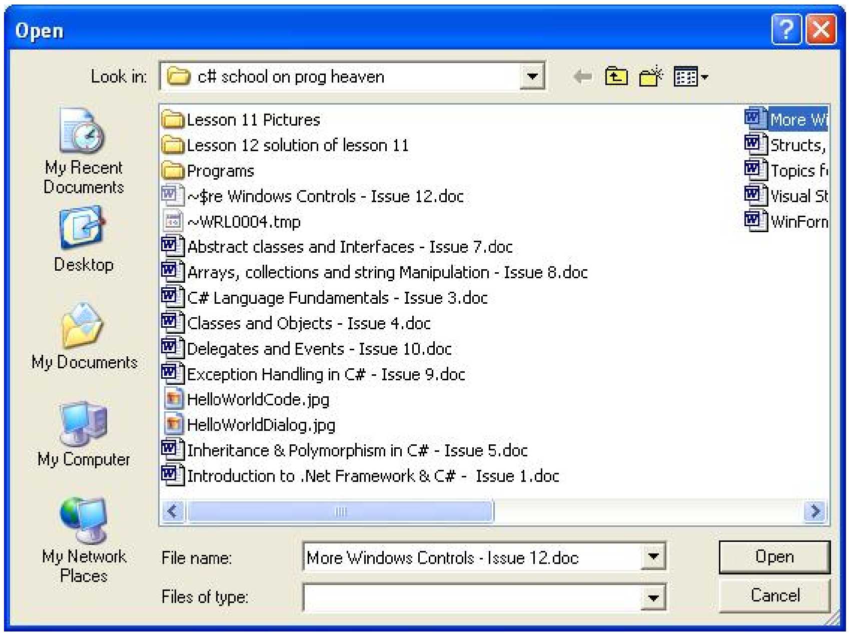
```
Tuesday, July 01, 2003
  Day:  1
  Month: 7
  Year:  2003
  Day of Week:  Tuesday
```

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 a Windows environment for opening files. It is an instance of System.Windows.Forms.OpenFileDialogBox. The following screen shot shows the common open file dialog.



Lesson 11 Pictures
Lesson 12 solution of lesson 11
Programs
~$re Windows Controls - Issue 12.doc
~WRL0004.tmp
Abstract classes and Interfaces - Issue 7.doc
Arrays, collections and string Manipulation - Issue 8.doc
C# Language Fundamentals - Issue 3.doc
Classes and Objects - Issue 4.doc
Delegates and Events - Issue 10.doc
Exception Handling in C# - Issue 9.doc
HelloWorldCode.jpg
HelloWorldDialog.jpg
Inheritance & Polymorphism in C# - Issue 5.doc
Introduction to .Net Framework & C# - Issue 1.doc
More Wi
Structs,
Topics fi
Visual St
WinForn

File name: More Windows Controls - Issue 12.doc
Files of type:

The important properties include:

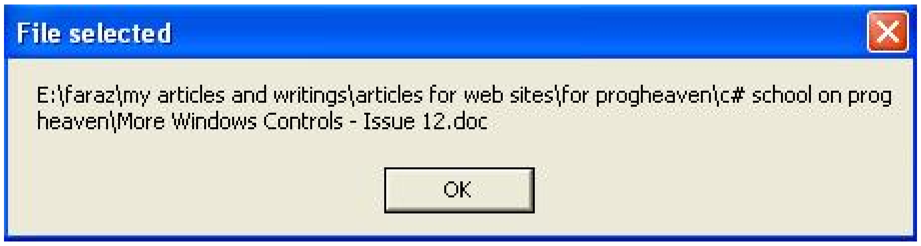
Property	Description
DefaultExt	The default extention for opening files. It uses wild card technique for filtering file names. If the value of DefaultExt is *.doc only then files with extention 'doc' will be visible in the open file dialog.
FileName	The full path and file name of the selected file.
InitialDirectory	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 file selected in a message box.

```
private void btnOpenFile_Click(object sender, System.EventArgs e)
{
    DialogResult res = openFileDialog.ShowDialog();
    if(res == DialogResult.OK)
    {
        MessageBox.Show(openFileDialog.FileName, "File selected");
    }
}
```

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 user the pressed OK button to close the dialog box. If yes then we printed the name of selected file in the message box. When the above code is executed, it shows the following result.



E:\faraz\my articles and writings\articles for web sites\for progheaven\c# school on prog heaven\More Windows Controls - Issue 12.doc

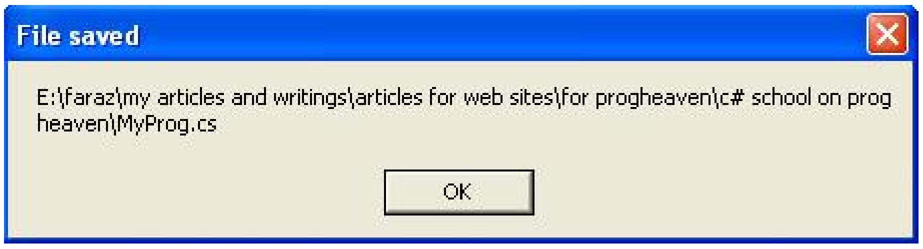
It is clear from the above message box that the FileName property returns the complete path of the selected file.

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 void btnSaveFile_Click(object sender, System.EventArgs e)
{
    DialogResult res = saveFileDialog.ShowDialog();
    if(res == DialogResult.OK)
    {
        MessageBox.Show(saveFileDialog.FileName, "File saved");
    }
}
```

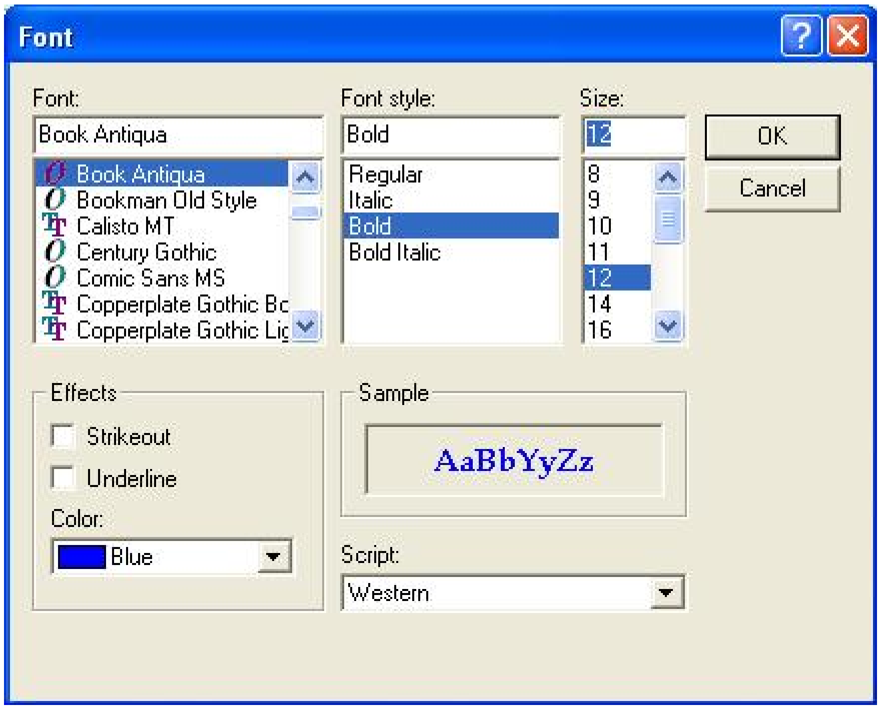
The code above will show the following message box:


File saved

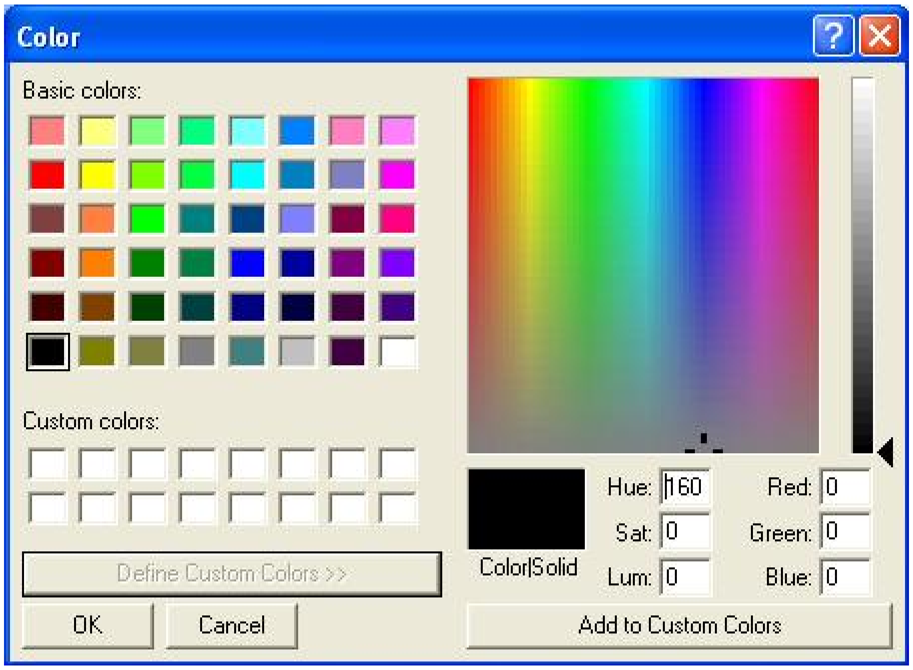
E:\faraz\my articles and writings\articles for web sites\for progheaven\c# school on prog heaven\MyProg.cs

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. It is an instance of System.Windows.Forms.ColorDialog. The font dialog box looks like this:

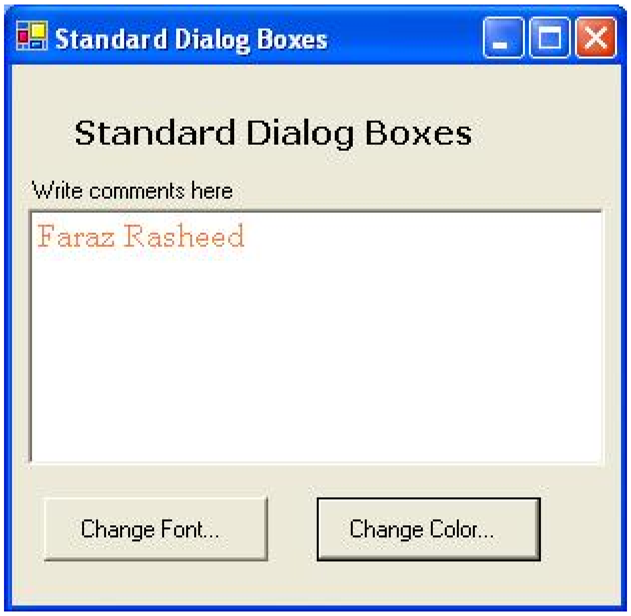


And the color dialog box looks like this:



Hue: 160
Sat: 0
Lum: 0
Red: 0
Green: 0
Blue: 0

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:



Faraz Rasheed

The form also contains two dialog boxes. One is fontDialog to change the font and the other is colorDialog to change the color. The event handler for two buttons are:

```
    private void btnChangeFont_Click(object sender, System.EventArgs e)
    {
        DialogResult res = fontDialog.ShowDialog();
        if(res == DialogResult.OK)
        {
            txtComment.Font = fontDialog.Font;
            txtComment.ForeColor = fontDialog.Color;
        }
    }

    private void btnChangeColor_Click(object sender, System.EventArgs e)
    {
        DialogResult res = colorDialog.ShowDialog();
        if(res == DialogResult.OK)
        {
            txtComment.ForeColor = colorDialog.Color;
        }
    }
```

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.

12. Data Access using ADO.Net

Lesson Plan

Today we will learn how our C# 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.

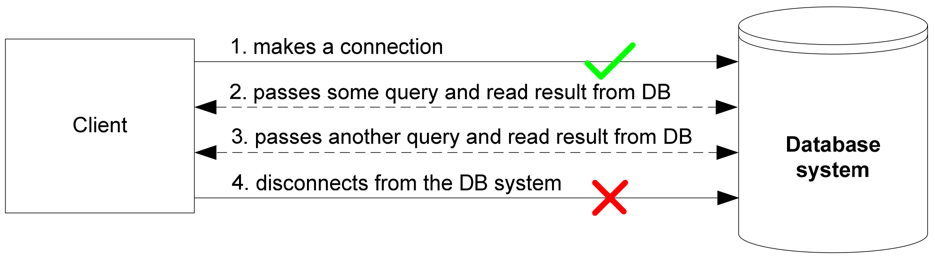
Introducing ADO.Net

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.

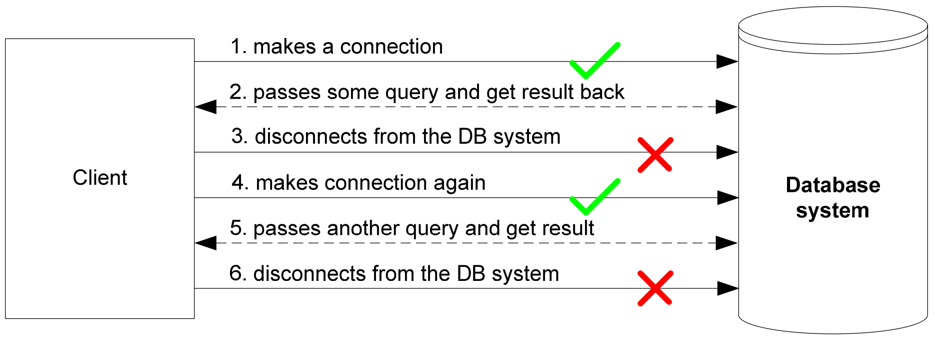
We usually interact with database systems through SQL queries or stored procedures. The best thing about ADO.Net is that it is extremely flexible and efficient. ADO.Net also introduces the concept of a disconnected data architecture. In traditional data access components, you made a connection to the 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 a 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.


Client
1. makes a connection
2. passes some query and read result from DB
3. passes another query and read result from DB
4. disconnects from the DB system
Database system

Traditional Data Access Architecture


Client
1. makes a connection
2. passes some query and get result back
3. disconnects from the DB system
4. makes connection again
5. passes another query and get result
6. disconnects from the DB system
Database system

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 relationship and their different constraints. The user can perform operations like update, insert and delete on this dataset locally, and 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 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 record sets.
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 data set..
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 name 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 query.
SqlDataAdapter, OleDbDataAdapter	A class that connects to the database system, fetches the record 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 of all the records from the table named 'emp'

```
SELECT * from emp where empno < 100
```

selects all those 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 those 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 set 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

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 the 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.

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
SqlDataAdapter dataAdapter = new
SqlDataAdapter(commandString, conn);
```

```
// For Access
OleDbDataAdapter dataAdapter = 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 article
title	String	The title of the article
topic	String	Topic or Series name of the article like 'Multithreading in Java' or 'C# School'

authorId	(Foreign Key)Integer	Unique identity of author
lines	Integer	No. 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 the 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 command is SELECT, filling the dataset object with the result of the query through the data adapter
- Reading the records from the DataTables in the datasets using the DataRow and DataColumn objects
- If the command is UPDATE, INSERT or DELETE, then updating the dataset 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:

```
using System.Data;

using System.Data.OleDb;       // for Access database

using 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
string connectionString = "server=P-III; database=programmersheaven;uid=sa; pwd=;";
```

First of all we have defined the instance name of the server, which is P-III on my system. Next we defined the name of the database, user id (uid) and password (pwd). Since my SQL server doesn't have a password for the System Administrator (sa) user, I have left it blank in the connection string. (Yes I know this is very dangerous and is really a bad practice - never, ever use a blank password on a system that is accessible over a network)
For Access, we have written the following connection string:

```
// for MS Access
string connectionString = "provider=Microsoft.Jet.OLEDB.4.0;data source = c:\\programmersheaven.mdb";
```

First we have defined the provider of the access database. Then we have defined the data source which is the address of the target database.

Author's Note: Connection string details are vendor specific. A good source for connection strings of 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 the connection is created like this:

```
// for Sql Server
SqlConnection conn = new SqlConnection(connectionString);
```

And for Access the connection is created like this:

```
// for MS Access
OleDbConnection conn = 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. Here 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:

```
string commandString = "SELECT " +
                        "artId, title, topic, " +
                        "article.authorId as authorId, " +
                        "name, lines, dateOfPublishing " +
                        "FROM " +
                        "article, author " +
                        "WHERE " +
                        "author.authorId = article.authorId";
```

Here we have passed a query to select all the articles along with the author's name. Of course you may use a simpler query, such as:

```
string commandString = "SELECT * from article";
```

Defining the Data Adapter

Next we need to define the data adapter (SqlDataAdapter or OleDbDataAdapter). The data adapter stores your command (query) and connection, and using these connects to the database when asked, fetches the result of the query and stores it in the local dataset.
For SQL Server, the data adapter is created like this:

```
// for Sql Server
SqlDataAdapter dataAdapter = new SqlDataAdapter(commandString, conn);
```

And for Access, the data adapter is created like this:

```
// for MS Access
OleDbDataAdapter dataAdapter = new OleDbDataAdapter(commandString, conn);
```

Here 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 & offline container of data. The DataSet object is created simply:

```
DataSet ds = new DataSet();
```

Now we need to fill the DataSet with the result of 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");
```

Here 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.

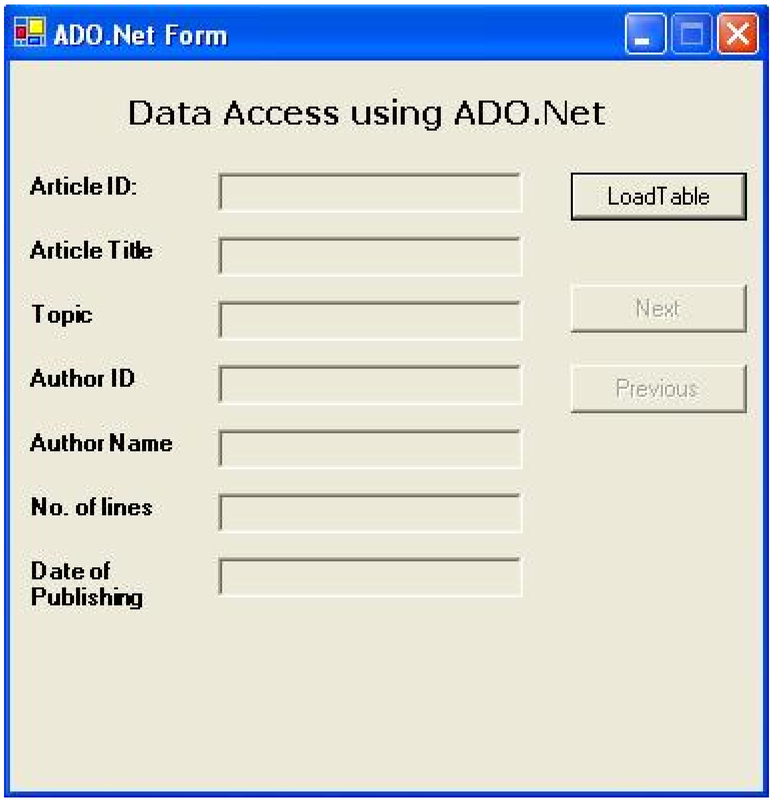
This is all we need to connect and fetch data from the database. Now the result of the query is stored in the dataset object in the prog table, which is an instance of the DataTable. We can get a reference to this table by using the indexer property of the dataset object's Tables collection.

```
DataTable dataTable = ds.Tables["prog"];
```

The indexer we have used takes the name of the table in the dataset and returns the corresponding DataTable object. Now 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 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 property to false).

We have also defined three variables in the Form class:

```
public class ADOForm : System.Windows.Forms.Form
{
    DataTable dataTable;
    int currRec=0;
    int totalRec=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 tables

For the LoadTable button, we have written the following event handler

```
private void btnLoadTable_Click(object sender, System.EventArgs e)
{
    // for Sql Server
    string connectionString = "server=P-III; database=programmersheaven;uid=sa; pwd=;";

    // for MS Access
    /*string connectionString = "provider=Microsoft.Jet.OLEDB.4.0;" +
                                         "data source = c:\\programmersheaven.mdb";*/

    // for Sql Server
    SqlConnection conn = new SqlConnection(connectionString);

    // for MS Access
    //OleDbConnection conn = new OleDbConnection(connectionString);

    string commandString = "SELECT " +
                           "artId, title, topic, " +
                           "article.authorId as authorId, " +
                           "name, lines, dateOfPublishing " +
                           "FROM " +
                           "article, author " +
                           "WHERE " +
                           "author.authorId = article.authorId";
    // for Sql Server
    SqlDataAdapter dataAdapter = new SqlDataAdapter(commandString, conn);

    // for MS Access
    //OleDbDataAdapter dataAdapter = new OleDbDataAdapter(commandString, conn);

    DataSet ds = new DataSet();
    dataAdapter.Fill(ds, "prog");

    dataTable = ds.Tables["prog"];
    currRec = 0;
    totalRec = dataTable.Rows.Count;

    FillControls();

    btnNext.Enabled = true;
    btnPrevious.Enabled = true;
}
```

First we created the connection, 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 de-comment the Access code.

Next, we have assigned the data table resulting from the query to the dataTable object which we declared at the class level, assigned zero to currRec variable and assigned the number of rows in the dataTable to

```
the totalRec variable:
dataTable = ds.Tables["prog"];
currRec = 0;
totalRec = 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 void 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();
}
```

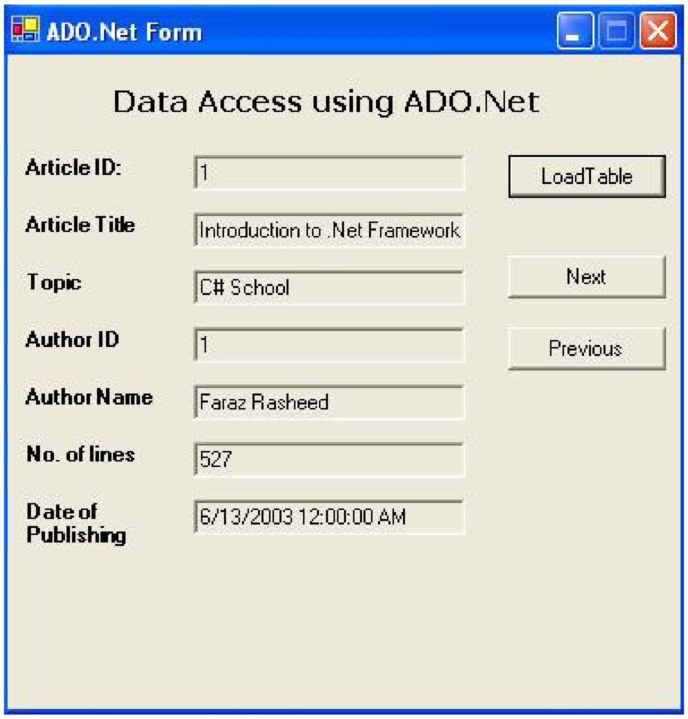
Here we have set the Text property of the text boxes to the string values of the corresponding fields of the current record. We have used the Rows collection of the dataTable and using its indexer we have got the DataRow representing the current record. We have then accessed the indexer property of this 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:-

```
DataRow row = dataTable.Rows[currRec]; // getting current row
object data = row["artId"];                  // getting data in the artId field
string strData = 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:



Article ID: 1
Article Title Introduction to .Net Framework
Topic C# School
Author ID 1
Author Name Faraz Rasheed
No. of lines 527
Date of Publishing 6/13/2003 12:00:00 AM

Navigating through the records

Navigating through the records is again very easy. For the Next button, we have written the following simple event handler

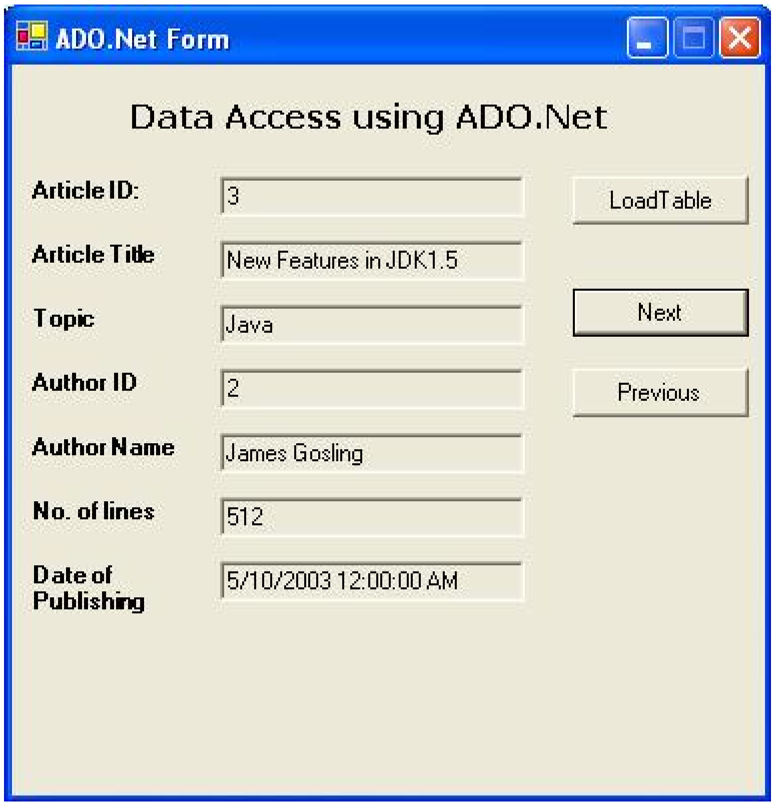
```
private void btnNext_Click(object sender, System.EventArgs e)
{
    currRec++;
    if(currRec>=totalRec)
       currRec=0;
    FillControls();
}
```

Here 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 handler for the Previous button looks like this:

```
private void btnPrevious_Click(object sender, System.EventArgs e)
{
    currRec--;
    if(currRec<0)
      currRec=totalRec-1;
    FillControls();
}
```

Here we decrement the currRec variable and check if has crossed the first record and if it has then we move it to the last record. Once again, we call the FillControls() method to display the current record.
Now you can navigate through the records using the Next and Previous buttons.



Article ID:	3
Article Title	New Features in JDK1.5
Topic	Java
Author ID	2
Author Name	James Gosling
No. of lines	512
Date of Publishing	5/10/2003 12:00:00 AM

Updating the table

So far we have only selected the data from the database and haven't changed any row, 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.

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 command. To implement this step, we will introduce a method named AddParams() in the following example.

The two steps described above are done only 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.

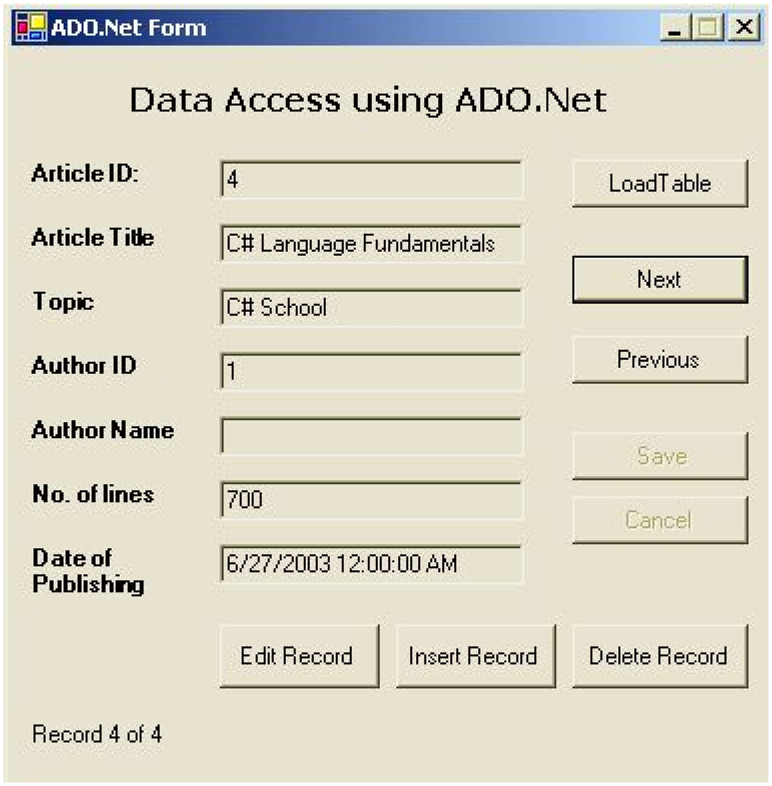
After any update we call the Update() method of the data adapter class by supplying to it, the dataset and table name as parameters. This updates our local dataset.

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 real application.

Building the Application

The application will finally look like this:



Article ID: 4
Article Title: C# Language Fundamentals
Topic: C# School
Author ID: 1
Author Name:
No. of lines: 700
Date of Publishing: 6/27/2003 12:00:00 AM

Record 4 of 4

In this application, we have defined several data access objects (like SqlDataAdapter, DataSet) as private class members so that we can access them in different methods.

```
public class ADOForm : System.Windows.Forms.Form
{
    // Private global members to be used in various methods
    private SqlConnection conn;
    private SqlDataAdapter dataAdapter;
    private DataTable dataTable;
    private DataSet ds;
    private int currRec=0;
    private int totalRec=0;
    private bool insertSelected;
    ...
```

Loading the table and displaying data in the form's controls

The event handler for the 'Load Table' button has changed a bit and now looks like this:

```
private void btnLoadTable_Click(object sender, System.EventArgs e)
{
    this.Cursor = Cursors.WaitCursor;

    string connectionString ="server=P-III; database=programmersheaven;uid=sa; pwd=;";

     conn = new SqlConnection(connectionString);
     string commandString = "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

     this.Cursor = Cursors.Default;
}
```

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 void 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");
}
```

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 any physical value. In fact, this value would 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 to it 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 void AddParams(SqlCommand cmd, params string[] cols)
{
    // Adding Hectice parameters in SQL Commands
    foreach(string col in cols)
    {
        cmd.Parameters.Add("@" + col, SqlDbType.Char, 0, col);
    }
}
```

The very first thing to note in the AddParams method is that the type of the second parameter is 'params string[]'

```
private void AddParams(SqlCommand cmd, params string[] cols)
```

The params keyword is used to tell the compiler that the method would 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? C# rules!

So let's get back to the method. What it does is simply adds the supplied column name prefixed with '@' in the Parameters collection of the SqlCommand class. The other parameters of the Add() method are the type of the parameter, 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 void btnLoadTable_Click(object sender, System.EventArgs e)
```

```
{
    this.Cursor = Cursors.WaitCursor;
    ...

    FillControls();       // show current record on the form
    InitializeCommands(); // prepare commands
    ToggleControls(true);     // enable corresponding controls

    this.Cursor = Cursors.Default;
}
```

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 void ToggleControls(bool val)
{
    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 = !val;
    btnCancel.Enabled = !val;
}
```

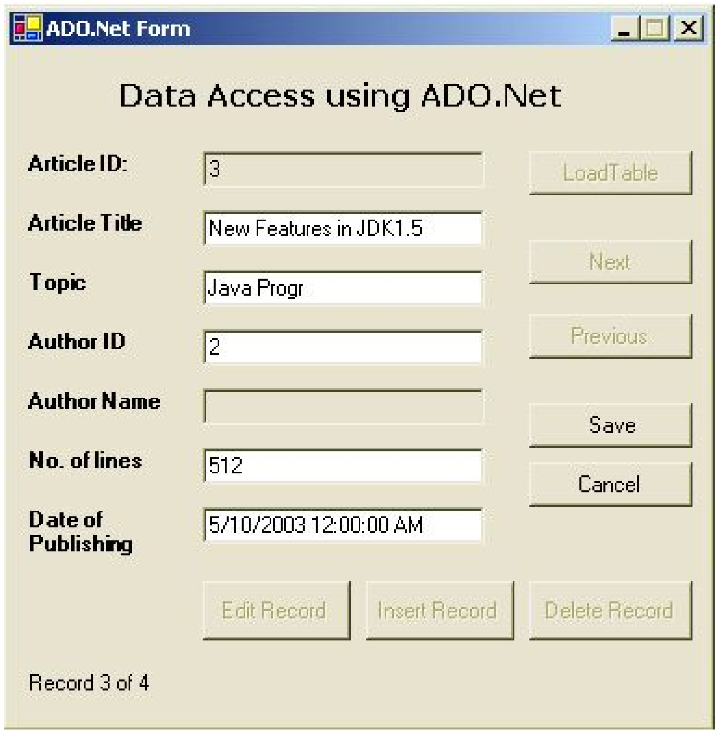
Editing (or Updating) Records

For editing the current record, we have provided an Edit Record button on the form. The event handler for this button is surprisingly very simple and is:

```
private void btnEditRecord_Click(object sender, System.EventArgs e)
```

```
{
    ToggleControls(false);
}
```

This event handler simply takes the form and the controls to 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 the changes.

Event Handler for the Save Button

The event handler for the Save button reads the values from the text boxes and stores them in the current record in the data table. The event handler for the Save Button is:

```
private void btnSave_Click(object sender, System.EventArgs e)
{
    lblLabel.Text = "Saving Changes...";
    this.Cursor = Cursors.WaitCursor;
    DataRow row = 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;
    this.Cursor = Cursors.Default;
    lblLabel.Text = "Changes Saved";
}
```

Here we first change the progress label (lblLabel) text to show the current progress and then change the cursor to wait cursor.

```
lblLabel.Text = "Saving Changes...";
    this.Cursor = Cursors.WaitCursor;
```

Then we take a reference to the current record's row and called its BeginEdit() method. The 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.

```
DataRow row = 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 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 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's event handler)

```
ToggleControls(true);
insertSelected = false;
this.Cursor = Cursors.Default;
lblLabel.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 Edit Record case, the pointer is already on 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, 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 Handler for the Cancel Button

The Cancel button's event handler is:

```
private void btnCancel_Click(object sender, System.EventArgs e)
{
     if(insertSelected)
     {
          btnDeleteRecord_Click(null, null);
          insertSelected = false;
     }

     FillControls();
     ToggleControls(true);
}
```

The form and controls can be brought into edit mode by pressing either the Edit Record or 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, in our application, a Boolean variable 'insertSelected' 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 event handler first checks whether insertSelected is true and if it is, it calls the Delete button's event handler to delete the current record and sets insertSelected back to false.

```
if(insertSelected)
{
    btnDeleteRecord_Click(null, null);
    insertSelected = false;
}
```

Then we 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 handler for the Insert Record button.

```
private void btnInsertRecord_Click(object sender, System.EventArgs e)
{
    insertSelected = true;
    DataRow row = 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);
}
```

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, added it to the Rows collection of the data table and updated the currRec and totalRec variables.

```
DataRow row = dataTable.NewRow();
dataTable.Rows.Add(row);
totalRec = dataTable.Rows.Count;
currRec = totalRec-1;
```

Now 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 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 void btnDeleteRecord_Click(object sender, System.EventArgs e)
{
    DialogResult res = MessageBox.Show(
                       "Are you sure you want to delete the current record?",
                       "Confirm Record Deletion", MessageBoxButtons.YesNo);

    if(res == DialogResult.Yes)
    {
        DataRow row = dataTable.Rows[currRec];
        row.Delete();
        dataAdapter.Update(ds, "article");
        ds.AcceptChanges();
        lblLabel.Text = "Record Deleted";
        totalRec--;
        currRec = totalRec-1;
        FillControls();
    }
}
```

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.

```
DialogResult res = 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)
{
    DataRow row = dataTable.Rows[currRec];
    row.Delete();
    dataAdapter.Update(ds, "article");
    ds.AcceptChanges();
    lblLabel.Text = "Record Deleted";
    totalRec--;
    currRec = totalRec-1;
    FillControls();
}
```

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.

Using Stored Procedures

So far 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 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 command in the previous example and we suppose you don't have any problem in understanding this even if you are not familiar with stored procedure.

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 his C# code. Real world stored procedures are much more complex and, of course, more useful than these!

Using Stored Procedures with ADO.Net in C#

Using stored procedures with ADO.Net in C# 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 void 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 void InitializeCommands()
{
    // Preparing Insert SQL Command
    SqlCommand 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;
    ...
```

Here we first created an SqlCommand object named insertCommand supplying in the constructor call the name of the target stored procedure (InsertProc in this case) and the connection to be used.

```
SqlCommand 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 void InitializeCommands()
{
    // Preparing Select Command
    SqlCommand selectCommand = new SqlCommand("SelectProc", conn);
    selectCommand.CommandType = CommandType.StoredProcedure;
    dataAdapter.SelectCommand = selectCommand;

    // Preparing Insert SQL Command
    SqlCommand insertCommand = new SqlCommand("InsertProc", conn);
    insertCommand.CommandType = CommandType.StoredProcedure;
    dataAdapter.InsertCommand = insertCommand;
    AddParams(dataAdapter.InsertCommand, "artId", "title", "topic",
              "authorId", "lines", "dateOfPublishing");

    // Preparing Update SQL Command
    SqlCommand updateCommand = new SqlCommand("UpdateProc", conn);
    updateCommand.CommandType = CommandType.StoredProcedure;
    dataAdapter.UpdateCommand = updateCommand;
    AddParams(dataAdapter.UpdateCommand, "artId", "title",
              "topic", "authorId", "lines", "dateOfPublishing");

    // Preparing Delete SQL Command
    SqlCommand deleteCommand = new SqlCommand("DeleteProc", conn);
    deleteCommand.CommandType = CommandType.StoredProcedure;
    dataAdapter.DeleteCommand= deleteCommand;
    AddParams(dataAdapter.DeleteCommand, "artId");
}
```

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 handler.

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 handler is:

```
private void btnLoadTable_Click(object sender, System.EventArgs e)
{
    this.Cursor = Cursors.WaitCursor;
    string connectionString = "server=P-III; database=programmersheaven;uid=sa; pwd=;";

    conn = new SqlConnection(connectionString);
    dataAdapter = new SqlDataAdapter();
    InitializeCommands();
    ds = new DataSet();
    dataAdapter.Fill(ds, "article");

    dataTable = ds.Tables["article"];
    currRec = 0;
    totalRec = dataTable.Rows.Count;

    FillControls();
    ToggleControls(true);

    this.Cursor = Cursors.Default;
}
```

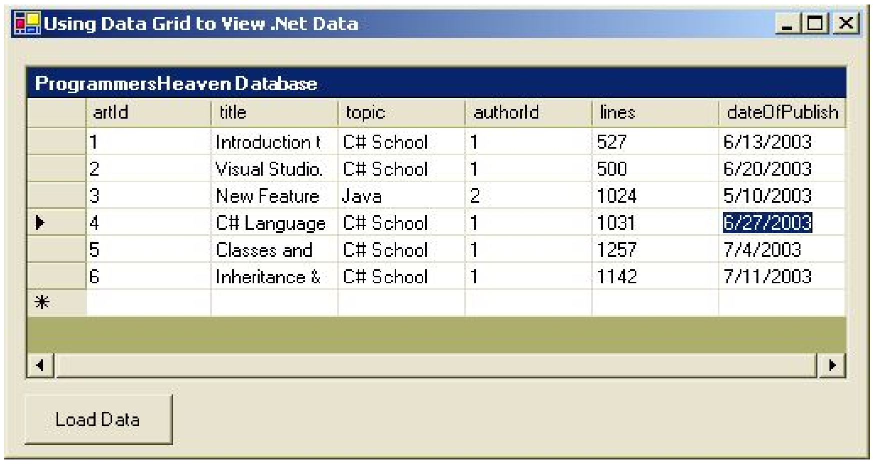
The two changes made are highlighted with bold formatting in the above code. Now we are creating the SqlDataAdapter object using the default no-argument constructor. This time we are not passing it the select command as we are using the stored procedure for it and we are not passing it the connection object as now 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 now 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 dataset and disconnected architecture, we don't update the data source (by calling DataAdapter's Update() method and 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 ofcourse 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 Data Grid Control to View .Net data

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:



	artId	title	topic	authorId	lines	dateOfPublish
	1	Introduction t	C# School	1	527	6/13/2003
	2	Visual Studio.	C# School	1	500	6/20/2003
	3	New Feature	Java	2	1024	5/10/2003
▶	4	C# Language	C# School	1	1031	6/27/2003
	5	Classes and	C# School	1	1257	7/4/2003
	6	Inheritance &	C# School	1	1142	7/11/2003
*						

A Demonstration Application for Data Grid Control

Let's create a simple application first 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 handler for the button is:

```
private void btnLoadData_Click(object sender, System.EventArgs e)
{
    string connectionString = "server=P-III; database=programmersheaven;uid=sa; pwd=;";
    SqlConnection conn = new SqlConnection(connectionString);
    string cmdString = "SELECT * FROM article";
    SqlDataAdapter dataAdapter = new SqlDataAdapter(cmdString, conn);
    DataSet ds = new DataSet();
    dataAdapter.Fill(ds, "article");

    dgDetails.SetDataBinding(ds, "article");
}
```

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 to the data grid control, which 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 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

Now we will 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:

```
DataRelation relation = new DataRelation("ArtAuth",
                                         ds.Tables["author"].Columns["authorId"],
                                         ds.Tables["article"].Columns["authorId"]);
```

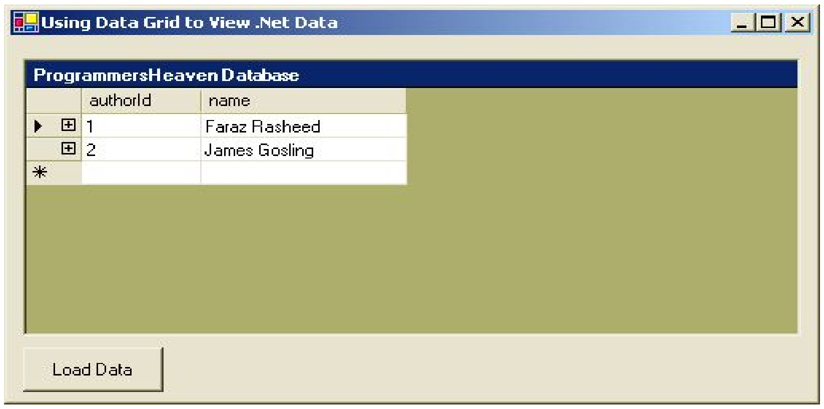
Here the first argument of the DataRelation constructor is the name for the new relation, while the second and third arguments are the columns of the tables which will be used to relate the two tables. After creating this relationship we need to add it to the Relations collection of the dataset.

```
ds.Relations.Add(relation);
```

Hence the modified code for the Load Data button is:

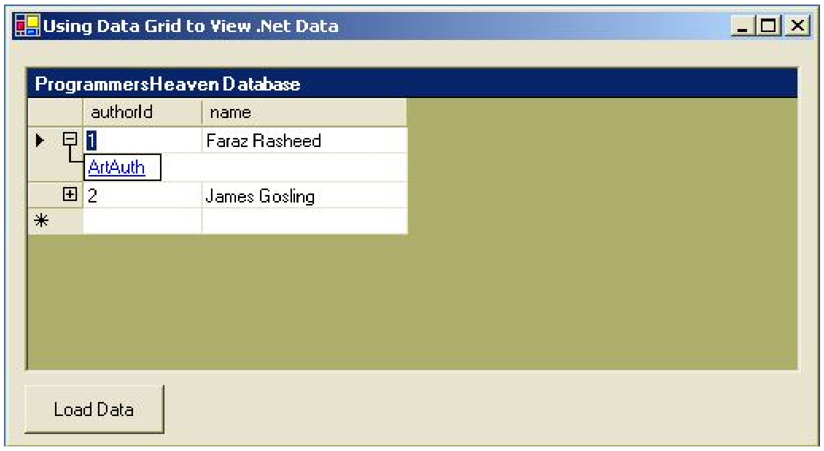
```
private void btnLoadData_Click(object sender, System.EventArgs e)
{
    string connectionString = "server=P-III; database=programmersheaven;uid=sa; pwd=;";
    SqlConnection conn = new SqlConnection(connectionString);

    string cmdString = "SELECT * FROM article";
    SqlDataAdapter dataAdapter = new SqlDataAdapter(cmdString, conn);
    DataSet ds = new DataSet();
    dataAdapter.Fill(ds, "article");
    cmdString = "SELECT * FROM author";
    dataAdapter = new SqlDataAdapter(cmdString, conn);
    dataAdapter.Fill(ds, "author");

    DataRelation relation = new DataRelation("ArtAuth",
        ds.Tables["author"].Columns["authorId"],
        ds.Tables["article"].Columns["authorId"]);
    ds.Relations.Add(relation);

    DataView dv = new DataView(ds.Tables["author"]);
    dgDetails.DataSource = dv;

}
```

In the above code we first filled the dataset with the two tables, defined the relationship between them and then added it to the dataset. In the last two lines, 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 this application, the data grid will show the records of parent table with '+' button on the left of each record:



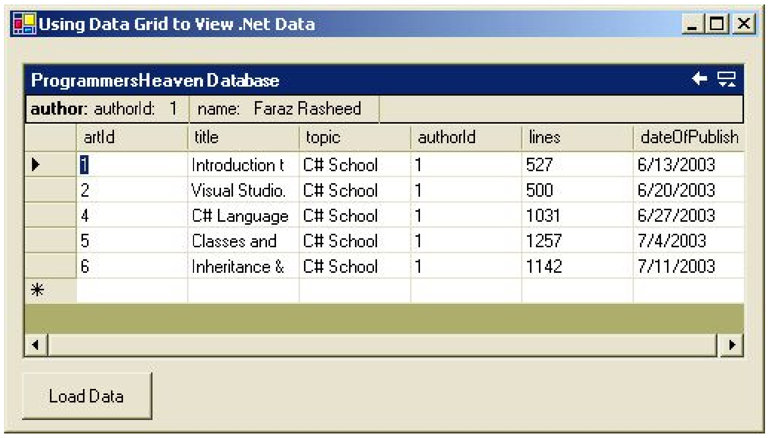
authorId	name
1	Faraz Rasheed
2	James Gosling

When you press the '+' button on the left of the record, it will expand to show the name of the relationship as a link:



	authorId	name
▶ ⊟	1	Faraz Rasheed
⊞	2	James Gosling
*		

Now when you click the relation name, the data grid will show all the related records in the child table:


Using Data Grid to View .Net Data
ProgrammersHeaven Database
author: authorId: 1 name: Faraz Rasheed
Load Data

	artId	title	topic	authorId	lines	dateOfPublish
▶	1	Introduction t	C# School	1	527	6/13/2003
	2	Visual Studio.	C# School	1	500	6/20/2003
	4	C# Language	C# School	1	1031	6/27/2003
	5	Classes and	C# School	1	1257	7/4/2003
	6	Inheritance &	C# School	1	1142	7/11/2003
*						

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.

Retrieving data using the SELECT command

For retrieving data using the SELECT command, we can write the following code:

```
private void btnLoadData_Click(object sender, System.EventArgs e)
{
    string connString = "server=siraj; database=programmersheaven; uid=sa; pwd=";
    SqlConnection conn = new SqlConnection(connString);
    string cmdString = "select * from author";
    SqlCommand cmd = new SqlCommand(cmdString, conn);

    conn.Open();
    SqlDataReader reader = cmd.ExecuteReader();
    while(reader.Read())
    {
        txtData.Text += reader["authorId"].ToString();
        txtData.Text += ", ";
        txtData.Text += reader["name"].ToString();
        txtData.Text += "\r\n";
    }
    conn.Close();
}
```

Here we have created an SqlConnection and an SqlCommand object by using the SELECT statement. We then opened 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.

```
SqlDataReader reader = 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, so we have called this method in the while statement. 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 += "\t";
    txtData.Text += reader["name"].ToString();
    txtData.Text += "\r\n";
}
```

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 Records using INSERT, UPDATE and DELETE commands

The procedure for updating records using INSERT, UPDATE and DELETE commands is very similar to the one we presented in the previous example except 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 void btnExecuteNonQuery_Click(object sender, System.EventArgs e)
{
    string connString = "server=siraj; database=programmersheaven; uid=sa; pwd=";
    SqlConnection conn = new SqlConnection(connString);

    // Un-comment any one of the following three Queries

    // INSERT Query
    string cmdString ="INSERT INTO Author " +
                      "(authorId, name) " +
                      "VALUES(3, 'Anders Hejlsberg')";

    // UPDATE Query
    /*string cmdString =  "UPDATE Author " +
                          "SET name = 'Grady Booch' " +
                          "WHERE authorId = 3";*/

    // DELETE Query
    /*string cmdString =  "DELETE FROM Author " +
                          "WHERE authorId = 3";*/

    SqlCommand cmd = new SqlCommand(cmdString, conn);

    conn.Open();
    cmd.ExecuteNonQuery();
    conn.Close();
}
```

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.

13. Multithreading

Lesson Plan

Today we will learn how to achieve multithreading in C# 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 C# application. Finally, we will learn about the different issues regarding thread synchronization and how C# 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 C# School in Microsoft Word, listening to my favorite songs in WinAmp and downloading a new song using 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, e.g. 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).

Consider the following small program:

```
namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            Fun1();
            Fun2();
            Console.WriteLine("End of Main()");
        }

        public static void Fun1()
        {
            for(int i=1; i<=5; i++)
            {
                Console.WriteLine("Fun1() writes: {0}", i);
            }
        }
        public static void Fun2()
        {
            for(int i=10; i>=6; i--)
            {
                Console.WriteLine("Fun2() writes: {0}", i);
            }
        }
    }
}
```

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 C#

The .Net Framework, and thus C#, 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 with has the void return type and which takes no arguments.

```
public delegate void ThreadStart();
```

For example, we can change our previous application to run the two methods in two different threads like this:

```
Thread firstThread = new Thread(new ThreadStart(Fun1));
Thread secondThread = new Thread(new ThreadStart(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 parameterless and has the void 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 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 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:

```
using System;
using System.Threading;

namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            Thread firstThread = new Thread(new ThreadStart(Fun1));
            Thread secondThread = new Thread(new ThreadStart(Fun2));

            firstThread.Start();
            secondThread.Start();

            Console.WriteLine("End of Main()");
        }

        public static void Fun1()
        {
            for(int i=1; i<=5; i++)
            {
                Console.WriteLine("Fun1() writes: {0}", i);
            }
        }
        public static void Fun2()
        {
            for(int i=10; i>=6; i--)
            {
                Console.WriteLine("Fun2() writes: {0}", i);
            }
        }
    }
}
```

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, but 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.

Static members of the System.Threading.Thread class

The static property CurrentThread gives a reference to the currently executing thread. Another important static 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 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.

```
static void 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");
}
```

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.

Thread Demonstration Example - Basic Operations

Now we will start to understand the implementation of threads in C#. Consider the following C# Console program:

```
using System;
using System.Threading;

namespace CSharpSchool
{
    class Test
    {
        static Thread mainThread;
        static Thread firstThread;
        static Thread secondThread;
        static void Main()
        {
            mainThread = Thread.CurrentThread;
            firstThread = new Thread(new ThreadStart(Fun1));
            secondThread = new Thread(new ThreadStart(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()");
        }

        public static void ThreadsInfo(string location)
        {
            Console.WriteLine("\r\nIn {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}\r\n",
              secondThread.Name, secondThread.ThreadState);
        }

        public static void Fun1()
        {
            for(int i=1; i<=5; i++)
            {
                Console.WriteLine("Fun1() writes: {0}", i);
                Thread.Sleep(100);
            }
            ThreadsInfo("Fun1()");
        }

        public static void Fun2()
        {
            for(int i=10; i>=6; i--)
            {
                Console.WriteLine("Fun2() writes: {0}", i);
                Thread.Sleep(125);
            }
            ThreadsInfo("Fun2()");
        }
    }
}
```

First of all we have defined three static references of type System.Threading.Thread to reference the three threads (main, first and second thread) later in the Main() method:

```
static Thread mainThread;
static Thread firstThread;
static Thread secondThread;
```

We have defined a static 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 static void Fun2()
{
    for(int i=10; i>=6; i--)
    {
        Console.WriteLine("Fun2() writes: {0}", i);
        Thread.Sleep(125);
    }
    ThreadsInfo("Fun2()");
}
```

Inside the Main() method we first instantiated the two thread instances (firstThread and secondThread) by passing to the 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.

```
static void Main()
{
    mainThread = Thread.CurrentThread;
    firstThread = new Thread(new ThreadStart(Fun1));
    secondThread = new Thread(new ThreadStart(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:

```
class Test
{
    static void Main()
    {
        Thread firstThread = new Thread(new ThreadStart(Fun1));
        Thread secondThread = new Thread(new ThreadStart(Fun2));

        firstThread.Name = "First Thread";
        secondThread.Name = "Second Thread";

        firstThread.Start();
        secondThread.Start();
    }

    public static void Fun1()
    {
        for(int i=1; i<=10; i++)
        {
            int t = new Random().Next(20);
            for(int j=0; j<t; j++)
            {
                new String(new char[] {});
            }
            Console.WriteLine(Thread.CurrentThread.Name +
                           ": created: " + t.ToString() + " empty strings");
        }
    }

    public static void Fun2()
    {
        for(int i=20; i>=11; i--)
        {
            int t = new Random().Next(20);
            for(int j=0; j<t; j++)
            {
                new String(new char[] {});
            }
            Console.WriteLine(Thread.CurrentThread.Name +
                            ": created: " + t.ToString()  + "empty strings");
        }
    }
}
```

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:

```
class Test
{
    static void Main()
    {
        Thread firstThread = new Thread(new ThreadStart(Fun1));
        Thread secondThread = new Thread(new ThreadStart(Fun2));

        firstThread.Name = "First Thread";
        secondThread.Name = "Second Thread";

        secondThread.Priority = ThreadPriority.AboveNormal;

        firstThread.Start();
        secondThread.Start();
    }

    public static void Fun1()
    {
        for(int i=1; i<=10; i++)
        {
            int t = new Random().Next(20);
            for(int j=0; j<t; j++)
            {
                new String(new char[] {});
            }
            Console.WriteLine(Thread.CurrentThread.Name +
                        ": created: " + t.ToString() + " empty strings");
        }
    }

    public static void Fun2()
    {
        for(int i=20; i>=11; i--)
        {
            int t = new Random().Next(40);
            for(int j=0; j<t; j++)
            {
                new String(new char[] {});
            }
            Console.WriteLine(Thread.CurrentThread.Name +
                          ": created: " + t.ToString() + " empty strings");
        }
    }
}
```

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:

```
static void Main()
{
    Thread firstThread = new Thread(new ThreadStart(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");
}

public static void Fun1()
{
    int i=1;
    try
    {
        for(; i<=20; i++)
        {
            int t = new Random().Next(20, 50);
            Console.WriteLine("Thread 1: slept for: " +  t.ToString() + " milliseconds");
            Thread.Sleep(t);
        }
    }
    catch(ThreadAbortException ex)
    {
        Console.WriteLine("Thread 1 aborted in iteration number: {0}", i);
    }
}
```

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:

```
static void Main()
{
    Thread firstThread = new Thread(new ThreadStart(Fun1));
    Thread secondThread = new Thread(new ThreadStart(Fun2));

    firstThread.Start();
    secondThread.Start();

    Console.WriteLine("Ending Main()");
}

public static void Fun1()
{
    for(int i=1; i<=5; i++)
    {
        Console.WriteLine("Thread 1 writes: {0}", i);
    }
}

public static void Fun2()
{
    for(int i=10; i>=5; i--)
    {
        Console.WriteLine("Thread 2 writes: {0}", i);
    }
}
```

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.

```
static void Main()
{
    Thread firstThread = new Thread(new ThreadStart(Fun1));
    Thread secondThread = new Thread(new ThreadStart(Fun2));

    firstThread.Start();
    secondThread.Start();

    firstThread.Join();
    Console.WriteLine("Ending Main()");
}
```

```
public static void Fun1()
{
    for(int i=1; i<=5; i++)
    {
        Console.WriteLine("Thread 1 writes: {0}", i);
    }
}

public static void Fun2()
{
    for(int i=15; i>=6; i--)
    {
        Console.WriteLine("Thread 2 writes: {0}", i);
    }
}
```

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 using only 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 with a recording of the 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 could not 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 C# Locking Mechanism

Consider the following code:

```
class Test
{
    static StringBuilder text = new StringBuilder();
    static void Main()
    {
        Thread firstThread = new Thread(new ThreadStart(Fun1));
        Thread secondThread = new Thread(new ThreadStart(Fun2));

        firstThread.Start();
        secondThread.Start();

        firstThread.Join();
        secondThread.Join();

        Console.WriteLine("Text is:\r\n{0}", text.ToString());
    }

    public static void Fun1()
    {
        for(int i=1; i<=20; i++)
        {
            Thread.Sleep(10);
            text.Append(i.ToString() + " ");
        }
    }

    public static void Fun2()
    {
        for(int i=21; i<=40; i++)
        {
            Thread.Sleep(2);
            text.Append(i.ToString() + " ");
        }
    }
}
```

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 C# lock keyword does. We can change the above program to provide the locking functionality:

```
class Test
{
     static StringBuilder text = new StringBuilder();
     static void Main()
     {
         Thread firstThread = new Thread(new ThreadStart(Fun1));
         Thread secondThread = new Thread(new ThreadStart(Fun2));

         firstThread.Start();
         secondThread.Start();

         firstThread.Join();
         secondThread.Join();

         Console.WriteLine("Text is:\r\n{0}", text.ToString());
     }

     public static void Fun1()
     {
         lock(text)
         {
             for(int i=1; i<=20; i++)
             {
                 Thread.Sleep(10);
                 text.Append(i.ToString() + " ");
             }
         }
     }

     public static void Fun2()
     {
         lock(text)
         {
             for(int i=21; i<=40; i++)
             {
                 Thread.Sleep(2);
                 text.Append(i.ToString() + " ");
             }
         }
     }
}
```

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 1
lock(A)
{
    lock(B)
    {
        // do some work
    }
}
```

While the second thread locks B and then A

```
// code for Thread 2
lock(B)
{
    lock(A)
    {
        // do some work
    }
}
```

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.

```
using System;
using System.Threading;
using System.Text;

namespace CSharpSchool
{
    class Test
    {
        static StringBuilder text = new StringBuilder();
        static StringBuilder doc = new StringBuilder();
        static void Main()
        {
            Thread firstThread = new Thread(new ThreadStart(Fun1));
            Thread secondThread = new Thread(new ThreadStart(Fun2));

            firstThread.Start();
            secondThread.Start();
        }

        public static void Fun1()
        {
            lock(text)
            {
                Thread.Sleep(10);
                for(int i=1; i<=20; i++)
                {
                    text.Append(i.ToString() + " ");
                }
                lock(doc)
                {
                    doc.Append(text.ToString());
                    for(int i=1; i<=20; i++)
                    {
                        doc.Append(i.ToString() + " ");
                    }
                }
            }
        }
        public static void Fun2()
        {
            lock(doc)
            {
                Thread.Sleep(10);
                for(int i=21; i<=40; i++)
                {
                    doc.Append(i.ToString() + " ");
                }
                lock(text)
                {
                    text.Append(doc.ToString());
                    for(int i=21; i<=40; i++)
                    {
                        text.Append(i.ToString() + " ");
                    }
                }
            }
        }
    }
}
```

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 any 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.

14. The File System & Streams

Lesson Plan

Today we will learn how we can manipulate the Windows file system and different types of streams using C# 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 C#. 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:

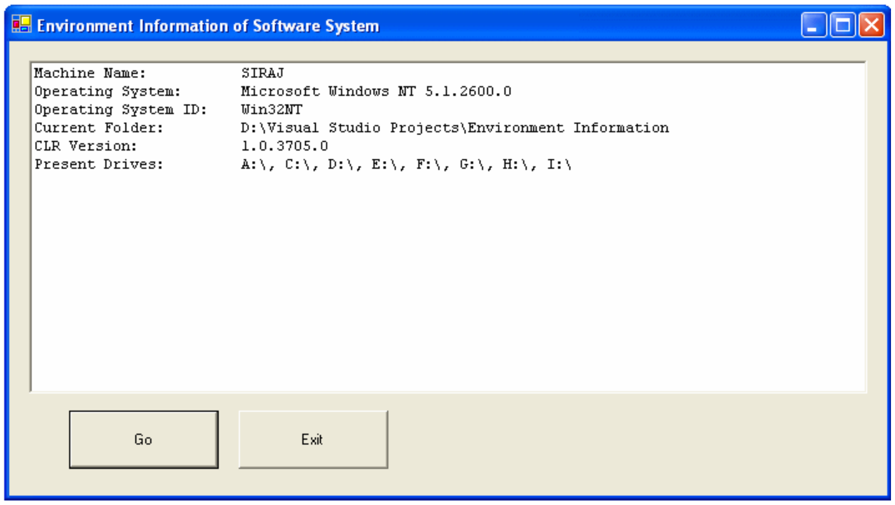
```
private void btnGo_Click(object sender, System.EventArgs e)
{
    OperatingSystem os = Environment.OSVersion;
    PlatformID OSid = os.Platform;

    string[] drives = Environment.GetLogicalDrives();
    string drivesString = "";
    foreach(string drive in drives)
    {
           drivesString += drive + ", ";
    }
    drivesString = drivesString.TrimEnd(' ', ',');

    lbx.Items.Add("Machine Name:         \t" + Environment.MachineName);
    lbx.Items.Add("Operating System:   \t" + Environment.OSVersion);
    lbx.Items.Add("Operating System ID:\t" + OSid);
    lbx.Items.Add("Current Folder:      \t" + Environment.CurrentDirectory);
    lbx.Items.Add("CLR Version:          \t" + Environment.Version);
    lbx.Items.Add("Present Drives:      \t" + drivesString);
}
```

Here we have simply retrieved the environment information from 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 exit button simply calls the Exit() method of the Environment class to terminate the current process.

```
private void btnExit_Click(object sender, System.EventArgs e)
{
  Environment.Exit(0);
}
```

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:



```
Machine Name:           SIRAJ
Operating System:       Microsoft Windows NT 5.1.2600.0
Operating System ID:    Win32NT
Current Folder:         D:\Visual Studio Projects\Environment Information
CLR Version:            1.0.3705.0
Present Drives:         A:\, C:\, D:\, E:\, F:\, G:\, H:\, I:\
```

Obtaining the paths of various Windows Standard folders – Environment.GetFolderPath()

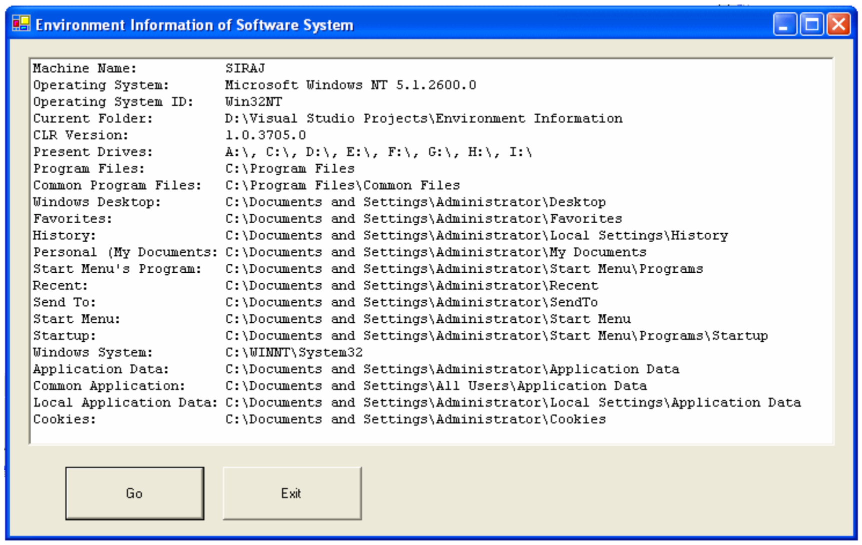
The method Environment.GetFolderPath() can be used to get the complete paths of various windows standard folders on the current machine. The only argument passed to the method is a value from the System.Environment.SpecialFolder enumeration. Some of the more common members of this enumeration are:

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 setting

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 void btnGo_Click(object sender, System.EventArgs e)
{
    OperatingSystem os = Environment.OSVersion;
    PlatformID OSid = os.Platform;
    string[] drives = Environment.GetLogicalDrives();
    string drivesString = "";
    foreach(string drive in drives)
    {
          drivesString += drive + ", ";
    }
    drivesString = drivesString.TrimEnd(' ', ',');

    lbx.Items.Add("Machine Name:       \t" + Environment.MachineName);
    lbx.Items.Add("Operating System:   \t" + Environment.OSVersion);
    lbx.Items.Add("Operating System ID:\t" + OSid);
    lbx.Items.Add("Current Folder:     \t" + Environment.CurrentDirectory);
    lbx.Items.Add("CLR Version:        \t" + Environment.Version);
    lbx.Items.Add("Present Drives:     \t" + drivesString);

    lbx.Items.Add("Program Files:     \t" +
      Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles));
    lbx.Items.Add("Common Program Files:\t" +
      Environment.GetFolderPath(Environment.SpecialFolder.CommonProgramFiles));
    lbx.Items.Add("Windows Desktop:   \t" +
        Environment.GetFolderPath(Environment.SpecialFolder.DesktopDirectory));
    lbx.Items.Add("Favorites:         \t" +
        Environment.GetFolderPath(Environment.SpecialFolder.Favorites));
    lbx.Items.Add("History:           \t" +
        Environment.GetFolderPath(Environment.SpecialFolder.History));
    lbx.Items.Add("Personal (My Documents:\t" +
        Environment.GetFolderPath(Environment.SpecialFolder.Personal));
    lbx.Items.Add("Start Menu's Program:\t" +
        Environment.GetFolderPath(Environment.SpecialFolder.Programs));
    lbx.Items.Add("Recent:            \t" +
        Environment.GetFolderPath(Environment.SpecialFolder.Recent));
    lbx.Items.Add("Send To:           \t" +
        Environment.GetFolderPath(Environment.SpecialFolder.SendTo));
    lbx.Items.Add("Start Menu:        \t" +
        Environment.GetFolderPath(Environment.SpecialFolder.StartMenu));
    lbx.Items.Add("Startup:           \t" +
        Environment.GetFolderPath(Environment.SpecialFolder.Startup));
    lbx.Items.Add("Windows System:    \t" +
        Environment.GetFolderPath(Environment.SpecialFolder.System));
    lbx.Items.Add("Application Data:  \t" +
        Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData));
    lbx.Items.Add("Common Application:\t" +
        Environment.GetFolderPath(Environment.SpecialFolder.CommonApplicationData));
    lbx.Items.Add("Local Application Data:\t" +
        Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData));
    lbx.Items.Add("Cookies:            \t" +
        Environment.GetFolderPath(Environment.SpecialFolder.Cookies));
}
```

When I executed the program with the above changes on my system, I got the following result:



```
Machine Name:          SIRAJ
Operating System:      Microsoft Windows NT 5.1.2600.0
Operating System ID:   Win32NT
Current Folder:        D:\Visual Studio Projects\Environment Information
CLR Version:           1.0.3705.0
Present Drives:        A:\, C:\, D:\, E:\, F:\, G:\, H:\, I:\
Program Files:         C:\Program Files
Common Program Files:  C:\Program Files\Common Files
Windows Desktop:       C:\Documents and Settings\Administrator\Desktop
Favorites:             C:\Documents and Settings\Administrator\Favorites
History:               C:\Documents and Settings\Administrator\Local Settings\History
Personal (My Documents: C:\Documents and Settings\Administrator\My Documents
Start Menu's Program:  C:\Documents and Settings\Administrator\Start Menu\Programs
Recent:                C:\Documents and Settings\Administrator\Recent
Send To:               C:\Documents and Settings\Administrator\SendTo
Start Menu:            C:\Documents and Settings\Administrator\Start Menu
Startup:               C:\Documents and Settings\Administrator\Start Menu\Programs\Startup
Windows System:        C:\WINNT\System32
Application Data:      C:\Documents and Settings\Administrator\Application Data
Common Application:    C:\Documents and Settings\All Users\Application Data
Local Application Data: C:\Documents and Settings\Administrator\Local Settings\Application Data
Cookies:               C:\Documents and Settings\Administrator\Cookies
```

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 static 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 Create() method

Suppose we wish to create a file named "c-sharp.txt" on the root folder of C drive. We can write the following statement to do this:

```
File.Create("C:\\c-sharp.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

```
using System.IO;
```

Copying and Moving a file using Copy() and Move() methods

Now if we want to copy this file to C:\my programs folder, we can use the following statement:

```
File.Copy("C:\\c-sharp.txt", "c:\\my programs\\c-sharp.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:\\c-sharp.txt", "c:\\my programs\\c-sharp.txt", true);
```

Checking the existence of the file using Exists() method

This method can be used to check the existence of the file

```
if(!File.Exists("C:\\c-sharp.txt"))
{
    File.Create("C:\\c-sharp.txt");
}
```

Getting Attributes of a file using GetAttributes() method

We can check the attributes of a file using the GetAttributes() method

```
FileAttributes attrs = File.GetAttributes("c:\\c-sharp.txt");
lbx.Items.Add("File 'c:\\c-sharp.txt'");
lbx.Items.Add(attrs.ToString());
```

When I executed the program, I got the information 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 of 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.
Extension	Returns the extension (type) of this file (e.g., .exe, .cs, .aspx).
FullName	Returns the full path and name of the file (e.g., C:\C-Sharp.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., C-Sharp.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

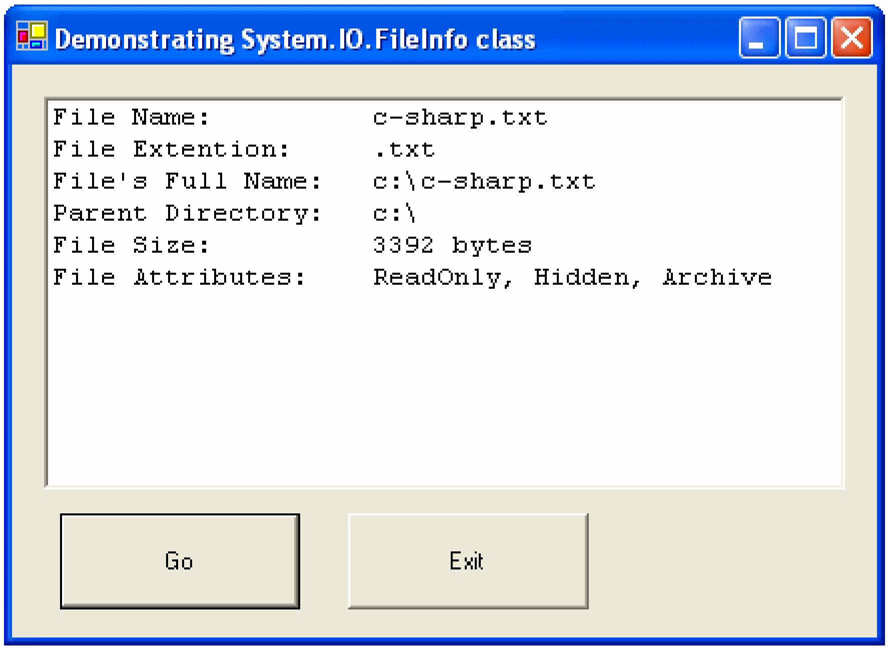
Although almost all the above properties and methods are understandable just by reading their name; we still need to create a simple example to demonstrate the functionality of the FileInfo class. In the following example, we will simply perform different operations on a file and display the result in 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, 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 void btnGo_Click(object sender, System.EventArgs e)
{
    FileInfo file = new FileInfo("c:\\c-sharp.txt");
    lbx.Items.Add("File Name:           " + file.Name);
    lbx.Items.Add("File Extention:      " + file.Extension);
    lbx.Items.Add("File's Full Name:    " + file.FullName);
    lbx.Items.Add("Parent Directory:    " + file.DirectoryName);
    lbx.Items.Add("File Size:           " + file.Length.ToString() + " bytes");
    lbx.Items.Add("File Attributes:     " + file.Attributes.ToString());
}
```

Here 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:



```
File Name:           c-sharp.txt
File Extention:      .txt
File's Full Name:    c:\c-sharp.txt
Parent Directory:    c:\
File Size:           3392 bytes
File Attributes:     ReadOnly, Hidden, Archive
```

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 static 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 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 void btnGo_Click(object sender, System.EventArgs e)
{
    lbx.Items.Add("Directory 'C:\\Faraz' exists:       " +  Directory.Exists("C:\\Faraz"));
    lbx.Items.Add("Creating Directory 'C:\\Faraz':     " +  Directory.CreateDirectory("C:\\Faraz"));
    lbx.Items.Add("Directory 'C:\\Faraz' exists:       " + Directory.Exists("C:\\Faraz"));
    lbx.Items.Add("Parent Directory of 'Faraz' is:    " + Directory.GetParent("C:\\Faraz"));
    lbx.Items.Add("Deleting Directory 'C:\\Faraz'...  ");
    Directory.Delete("C:\\Faraz", true);
    lbx.Items.Add("Directory 'C:\\Faraz' exists:       " + Directory.Exists("C:\\Faraz"));
}
```

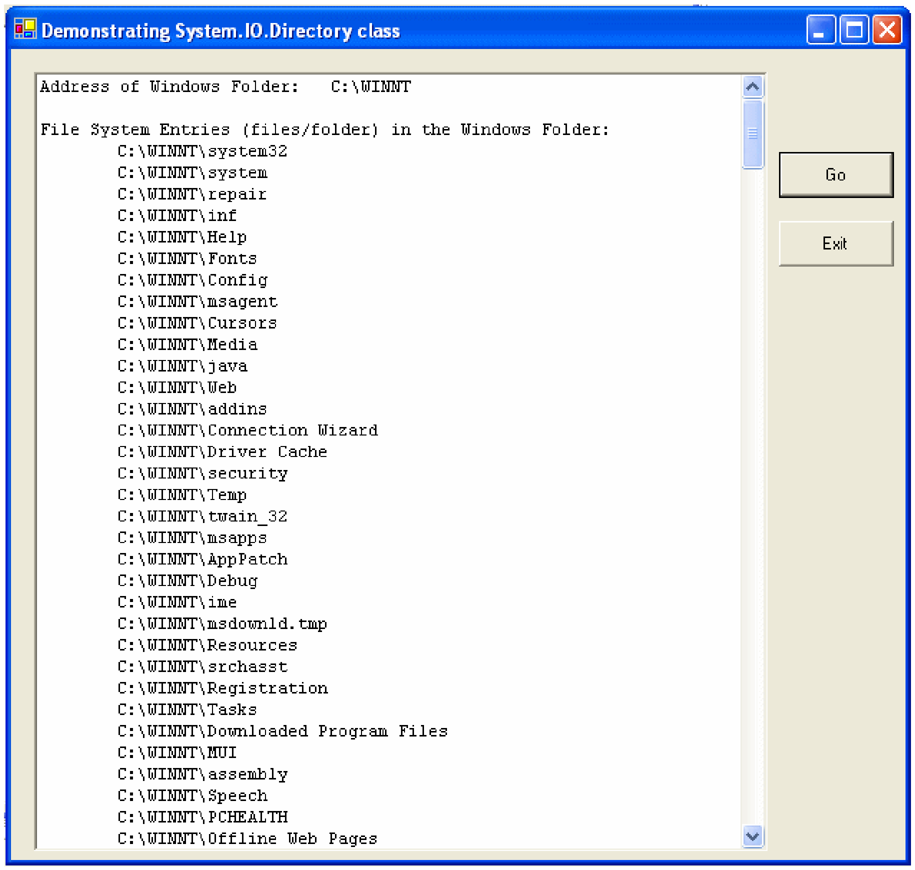
Again, the code simply calls the various static methods of the Directory class to perform these operations. One thing to note here is that we have passed the path-string and a true value to the Directory.Delete() method. Passing the true value as the second parameter tells the runtime environment (CLR) 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 void btnGo_Click(object sender, System.EventArgs e)
{
    // get the path of Windows Folder's System Folder
    string winFolder = Environment.GetFolderPath(Environment.SpecialFolder.System);
    // Separate the Windows Folder
    winFolder = winFolder.Substring(0, winFolder.LastIndexOf('\\'));

    string[] fileSystemEntries = Directory.GetFileSystemEntries(winFolder);
    string[] files = Directory.GetFiles(winFolder);
    string[] directories = 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: ");
    foreach(string fileSystemEntry in fileSystemEntries)
    {
        lbx.Items.Add("\t" + fileSystemEntry);
    }

    // show files in windows folder
    lbx.Items.Add("");
    lbx.Items.Add("Files in the Windows Folder: ");
    foreach(string file in files)
    {
        lbx.Items.Add("\t" + file);
    }

    // show folder in windows folder
    lbx.Items.Add("");
    lbx.Items.Add("Directories in the Windows Folder: ");
    foreach(string directory in directories)
    {
        lbx.Items.Add("\t" + directory);
    }
}
```

And when I executed the above program on my system, I got the following result:



```
Address of Windows Folder:   C:\WINNT

File System Entries (files/folder) in the Windows Folder:
        C:\WINNT\system32
        C:\WINNT\system
        C:\WINNT\repair
        C:\WINNT\inf
        C:\WINNT\Help
        C:\WINNT\Fonts
        C:\WINNT\Config
        C:\WINNT\msagent
        C:\WINNT\Cursors
        C:\WINNT\Media
        C:\WINNT\java
        C:\WINNT\Web
        C:\WINNT\addins
        C:\WINNT\Connection Wizard
        C:\WINNT\Driver Cache
        C:\WINNT\security
        C:\WINNT\Temp
        C:\WINNT\twain_32
        C:\WINNT\msapps
        C:\WINNT\AppPatch
        C:\WINNT\Debug
        C:\WINNT\ime
        C:\WINNT\msdownld.tmp
        C:\WINNT\Resources
        C:\WINNT\srchasst
        C:\WINNT\Registration
        C:\WINNT\Tasks
        C:\WINNT\Downloaded Program Files
        C:\WINNT\MUI
        C:\WINNT\assembly
        C:\WINNT\Speech
        C:\WINNT\PCHEALTH
        C:\WINNT\Offline Web Pages
```

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:

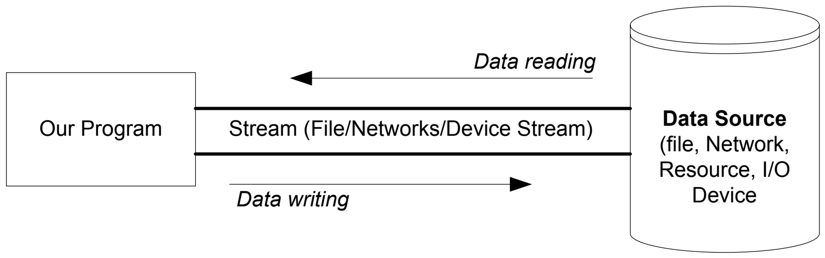
```
private void btnGo_Click(object sender, System.EventArgs e)
{
    // get the path of Windows Folder's System Folder
    string winFolder = Environment.GetFolderPath(Environment.SpecialFolder.System);
    // Separate the Windows Folder
    winFolder = winFolder.Substring(0, winFolder.LastIndexOf('\\'));

    DirectoryInfo winFolderObj = new DirectoryInfo(winFolder);
    FileSystemInfo[] fileSystemInfos = winFolderObj.GetFileSystemInfos();
    FileInfo[] fileInfos = winFolderObj.GetFiles();
    DirectoryInfo[] directoryInfos = 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: ");
    foreach(FileSystemInfo fileSystemInfo in fileSystemInfos)
    {
        lbx.Items.Add("\t" + fileSystemInfo.FullName);
    }

    // show files in windows folder
    lbx.Items.Add("");
    lbx.Items.Add("Files in the Windows Folder: ");
    foreach(FileInfo fileInfo in fileInfos)
    {
        lbx.Items.Add("\t" + fileInfo.FullName);
    }

    // show folder in windows folder
    lbx.Items.Add("");
    lbx.Items.Add("Directories in the Windows Folder: ");
    foreach(DirectoryInfo directoryInfo in directoryInfos)
    {
        lbx.Items.Add("\t" + directoryInfo.FullName);
    }
}
```

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 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 exception that might be thrown because of the absence of specified files and the privileges of different files. It is always better practice to write the file access code in try...catch blocks.

Streams

In a programming context, various definitions of stream exist. Some say 'A stream is an object used to read and write data to and from some 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 the data source while the other is attached to the reader or writer of the 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 the data source.


Data reading
Our Program
Stream (File/Networks/Device Stream)
Data Source
(file, Network, Resource, I/O Device
Data writing

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 device. 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. Dot Net (.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, and 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(byte[] buffer, int offset, int count) (Return type: int)	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(byte[] buffer, int offset, int count) (Return type: void)	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 use the System.IO namespace in our project

```
using System;
using 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.

```
FileStream fs = File.Open("C:\\C-sharp.txt", FileMode.Open);
```

You may also use the FileInfo class' Open(), OpenRead() and OpenWrite() methods.

```
FileInfo objFileInfo = new FileInfo("C:\\C-sharp.txt");
FileStream fs = 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

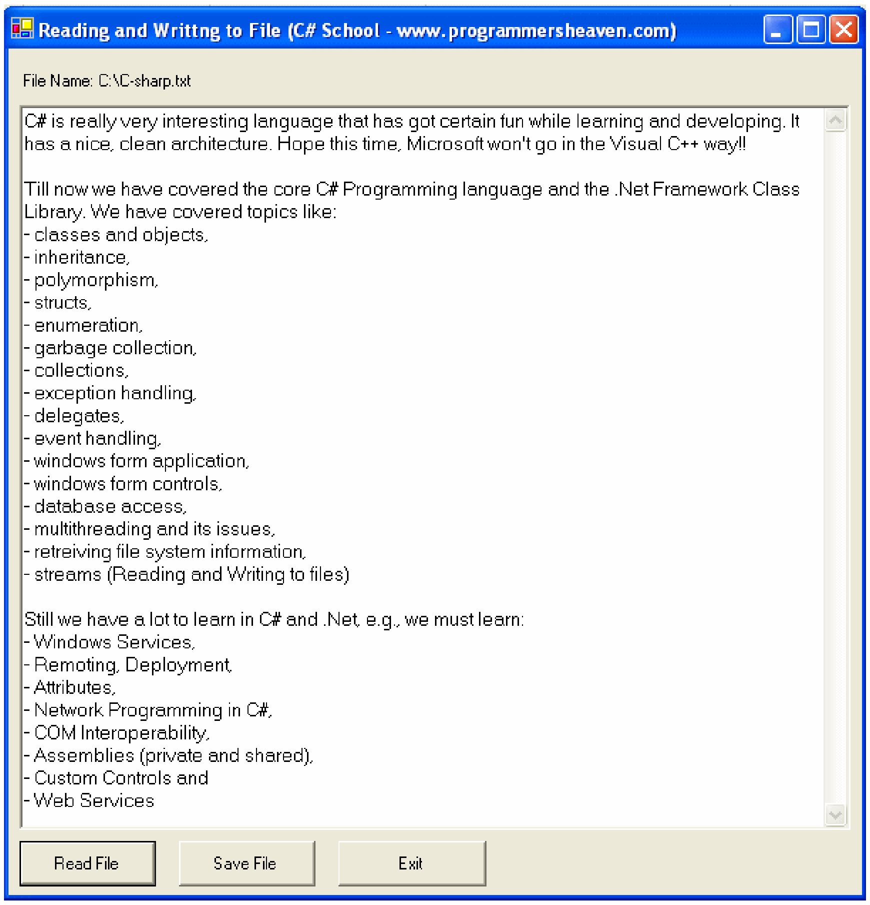
A FileMode enumeration instance that defines how the operating system should open the file. The possible values include Open, OpenOrCreate, Append, Create and others. If FileMode is Open, it will attempt to open the existing file. If FileMode is OpenCreate it will attempt to open the existing file; if 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 C#. 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:


Reading and Writtng to File (C# School - www.programmersheaven.com)
File Name: C:\C-sharp.txt
Read File
Save File
Exit

```
C# is really very interesting language that has got certain fun while learning and developing. It
has a nice, clean architecture. Hope this time, Microsoft won't go in the Visual C++ way!!

Till now we have covered the core C# Programming language and the .Net Framework Class
Library. We have covered topics like:
- classes and objects,
- inheritance,
- polymorphism,
- structs,
- enumeration,
- garbage collection,
- collections,
- exception handling,
- delegates,
- event handling,
- windows form application,
- windows form controls,
- database access,
- multithreading and its issues,
- retreiving file system information,
- streams (Reading and Writing to files)

Still we have a lot to learn in C# and .Net, e.g., we must learn:
- Windows Services,
- Remoting, Deployment,
- Attributes,
- Network Programming in C#,
- COM Interoperability,
- Assemblies (private and shared),
- Custom Controls and
- Web Services
```

The code behind it is very simple. The event handler for the Read File button (btnGo) is:

```
private void btnGo_Click(object sender, System.EventArgs e)
{
    string fileName = "C:\\C-sharp.txt";
    lblFileName.Text = "File Name: " + fileName;
    // Open existing file in Read only mode without allowing any sharing
    FileStream fs = 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.
    byte[] byteText = new byte[fs.Length];
    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();
}
```

Here, 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 the array 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 void btnSaveFile_Click(object sender, System.EventArgs e)
{
    string fileName = "C:\\C-sharp.txt";

    // Open existing file in Read only mode without allowing any sharing
    FileStream fs = 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
    byte[] byteText = 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();
}
```

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);
```

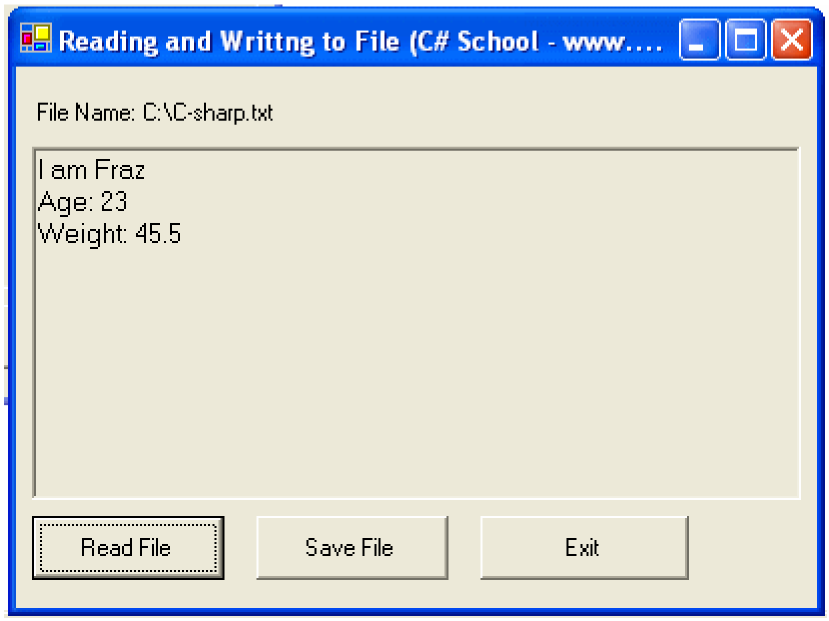
In the above line we have told the FileStream class to write the bytes in the byteText array to the associated file. We have asked it to start writing from the first byte of the array and write the complete array to the file.

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 the 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 Dot Net (.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:


Reading and Writtng to File (C# School - www....
File Name: C:\C-sharp.txt
Read File
Save File
Exit

```
I am Fraz
Age: 23
Weight: 45.5
```

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 void btnSaveFile_Click(object sender, System.EventArgs e)
{
    string fileName = "C:\\C-sharp.txt";

    // Open existing file in Read only mode without allowing any sharing
    FileStream fs = new FileStream(fileName, FileMode.Open,   FileAccess.Write, FileShare.None);

    // Open the Writer over this file stream
    BinaryWriter writer = new BinaryWriter(fs);

    // write different types of primitives to the file
    writer.Write("I am Fraz\r\n");
    writer.Write("Age: ");
    writer.Write(23);
    writer.Write("\r\nWeight: ");
    writer.Write(45.5);

    // close the file stream so that other streams may use it
    writer.Close();
    fs.Close();

    btnLoadFile.Enabled = true;
}
```

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 type of primitives. Finally we have closed the two streams and enabled the Load File button.
The event handler for the Load File button is:

```
private void btnGo_Click(object sender, System.EventArgs e)
{
    string fileName = "C:\\C-sharp.txt";
    lblFileName.Text = "File Name: " + fileName;

    // Open existing file in Read only mode without allowing any sharing
    FileStream fs = new FileStream(fileName, FileMode.Open,  FileAccess.Read, FileShare.None);
    // Open the Reader over this file stream
    BinaryReader reader = new BinaryReader(fs);

    // read different types of primitives to the file
    string name = reader.ReadString();
    string ageString = reader.ReadString();
    int age = reader.ReadInt32();
    string wtString = reader.ReadString();
    double weight = 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();
}
```

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.

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.

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. 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:

- 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.
- 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 Dot (.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 C# School up till this issue. Attributes are a fantastic feature of C# that allow 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:

```
using System;
using System.IO;                    // for FileStream
using System.Runtime.Serialization.Formatters.Binary;    // for BinaryFormatter

namespace Compiler
{
    class Test
    {
        static void Main()
        {
            Addition addition = new Addition(3, 4);

            FileStream fs = new FileStream(@"C:\C-Sharp.txt", FileMode.Create);
            BinaryFormatter binForm = 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....");
            Addition sum = (Addition) binForm.Deserialize(fs);

            int res = sum.Add();
            Console.WriteLine("The sum of 3 and 4 is: {0}", res);
        }
    }

    [Serializable]
    class Addition
    {
        private int num1;
        private int num2;
        private int result;

        public Addition()
        {
        }

        public Addition(int num1, int num2)
        {
            this.num1 = num1;
            this.num2 = num2;
        }


        public int Add()
        {
            result = num1 + num2;
            return result;
        }

        public int Result
        {
            get { return result; }
        }
    }
}
```

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 appropriate namespaces.

```
using System;
Uusing System.IO;                  // for FileStream
using 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).

```
FileStream fs = new FileStream(@"C:\C-Sharp.txt", FileMode.Create);
BinaryFormatter binForm = 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....");
Addition sum = (Addition) binForm.Deserialize(fs);
```

The BinaryFormatter class' Deserialize() method takes the file stream as its parameter, 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

```
int res = sum.Add();
Console.WriteLine("The sum of 3 and 4 is: {0}", res);
```

When you compile and execute the program, 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 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:

```
using System;
using System.IO;                   // for FileStream
using System.Runtime.Serialization.Formatters.Binary; // for BinaryFormatter

namespace Compiler
{
    class Test
    {
        static void Main()
        {
            Addition addition = new Addition(3, 4);
            addition.Add();
            Console.WriteLine("The value of result is: {0}", addition.Result);

            FileStream fs = new FileStream(@"C:\C-Sharp.txt", FileMode.Create);
            BinaryFormatter binForm = 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....");
            Addition sum = (Addition) binForm.Deserialize(fs);

            Console.WriteLine("The value of result is: {0}", sum.Result);

            Console.WriteLine("The sum of addition is: {0}", sum.Add());
        }
    }

    [Serializable]
    class Addition
    {
        private int num1;
        private int num2;
        [NonSerialized]
        private int result;
        public Addition()
        {
        }

        public Addition(int num1, int num2)
        {
            this.num1 = num1;
            this.num2 = num2;
        }


        public int Add()
        {
            result = num1 + num2;
            return result;
        }

        public int Result
        {
            get { return result; }
        }
    }
}
```

Note the [NonSerialized] attribute over the result field. Now consider the Main() method of the program. We first create an instance of the Addition class with two numbers, call its Add() method to compute the result and print it.

```
Addition addition = 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:

```
FileStream fs = new FileStream(@"C:\C-Sharp.txt", FileMode.Create);
BinaryFormatter binForm = 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....");
```

```
Addition sum = (Addition) binForm.Deserialize(fs);
```

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 like to perform some work on the object when deserializing so that we may prepare non-serialized fields. For example, we may like 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

```
void OnDeserialization(object sender);
```

This method is always called in the implementing class when the object is deserialized. Let's change the previous application so that result variable retains its value even if it is not serialized. The modified source code is:

```
using System;
using System.IO;                  // for FileStream
using System.Runtime.Serialization;           // for IDeserializationCallBack
using System.Runtime.Serialization.Formatters.Binary; // for BinaryFormatter
namespace Compiler
```

```
{
    class Test
    {
        static void Main()
        {
            Addition addition = new Addition(3, 4);
            addition.Add();
            Console.WriteLine("The value of result is: {0}", addition.Result);

            FileStream fs = new FileStream(@"C:\C-Sharp.txt", FileMode.Create);
            BinaryFormatter binForm = 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....");
            Addition sum = (Addition) binForm.Deserialize(fs);

            Console.WriteLine("The value of result is: {0}", sum.Result);
        }
    }

    [Serializable]
    class Addition : IDeserializationCallback
    {
        private int num1;
        private int num2;
        [NonSerialized]
        private int result;

        public Addition()
        {
        }

        public Addition(int num1, int num2)
        {
            this.num1 = num1;
            this.num2 = num2;
        }
        public int Add()
        {
            result = num1 + num2;
            return result;
        }

        public int Result
        {
            get { return result; }
        }
        public void OnDeserialization(object sender)
        {
            result = num1 + num2;
        }
    }
}
```

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 void OnDeserialization(object sender)
{
    result = num1 + num2;
}
```

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.

Asynchronous Reading and Writing with Streams

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.

```
byte[] byteText = new byte[fs.Length];
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 the 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 C# 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 void AsyncCallback(IAsyncResult ar)
```

This means that the delegate AsyncCallback can reference any method that has a void 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 void OnWriteCompletion(IAsyncResult ar)
{
    Console.WriteLine("Write Operation Completed");
}
```

A demonstration application

Let’s now create a simple console application that demonstrates the use of Asynchronous read/write to streams. The read/write operations in this application will be asynchronous. The source code of the program is:

```
using System;
using System.IO;
using System.Threading;

namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            FileStream fs = new FileStream(@"C:\C-Sharp.txt", FileMode.Open);
            byte[] fileData = new byte[fs.Length];

            Console.WriteLine("Reading file...");
            fs.Read(fileData, 0, fileData.Length);

            fs.Position = 0;
            AsyncCallback callbackMethod = new AsyncCallback(OnWriteCompletion);
            fs.BeginWrite(fileData, 0, fileData.Length, callbackMethod, null);

            Console.WriteLine("Write command issued");
            for(int i=0; i<10; i++)
            {
                Console.WriteLine("Count Reaches: {0}", i);
                Thread.Sleep(10);
            }
            fs.Close();
        }
        static void OnWriteCompletion(IAsyncResult ar)
        {
            Console.WriteLine("Write Operation Completed...");
        }
    }
}
```

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, read 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()’.

```
AsyncCallback callbackMethod = new AsyncCallback(OnWriteCompletion);
fs.BeginWrite(fileData, 0, fileData.Length, callbackMethod, null);
```

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 till the writing is complete, it should continue and print the numbers in the loop. When 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 ‘C-Sharp.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:

- 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() 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 a 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 would be closed.

15. New Features In C# 2.0

C# evolves

Everything presented so far in this book should work with both the first and second versions of the C# language and the underlying .Net framework. This chapter looks at the major additions that were made to the second version of C#, known as C# 2.0.

Much care is needed when adding new features to a language, particularly with regard to what effect they will have on existing programs. Usually it is desirable to maintain backward compatibility with programs written in previous versions of the language. C# 2.0 is a superset of C# 1.0, meaning that programs written in C# 1.0 should still compile and run as expected with C# 2.0 and version 2.0 of the .Net runtime. The converse, however, is not true. Clearly C# 2.0 programs cannot be expected to compile with a C# 1.0 compiler, but you might expect that the compiled .Net bytecode might run on the first version of the CLR. This is not the case – additions have also been made at the runtime level, for good reasons that will be explored later. This means that if you write, compile and distribute C# 2.0 programs, your users will need to have the second version of the .Net runtime to run them.

The need for generics

The addition of generics is the most significant change in version 2.0 of the C# language. The concept behind generics is known as parametric polymorphism, which provides us with more of a clue as to what generics is about. We have already seen polymorphism in C#, way back in chapter 4. Polymorphism is about types, and the form of polymorphism we have seen so far in C# has been subtyping, achieved through inheritance. Here more specialized classes can be used anywhere that the parent class they inherit from can, and the compiler knows that this is safe since the child classes have the same behavior and representation as the parent classes.

Parametric polymorphism is just another way of introducing abstractions into your program. It aims to deal with a different problem to subtyping; the two complement each other rather than compete. Consider the following snippet of C# code.

```
// Create a list that we'll put values of factorial in.
ArrayList factorial = new ArrayList();


// Compute first 10 factorials.
int fact = 1;
for (int i = 1; i <= 10; i++)
{
    fact *= i;
    factorial.Add(fact);
}
```

```
// Print them.
for (int i = 0; i < 10; i++)
{
    int value = (int) factorial[i];
    Console.WriteLine(value);
}
```

Here we are using an ArrayList collection to hold a list of integers – the first 10 values of the factorial function in this case. This code works as expected, so what's the problem? The first issue is that we have to use a cast (to type int) when retrieving values from the collection. A cast implies a runtime check – this means that every time we retrieve a value we have to check it really is an integer. Aside from this runtime cost, we can see from looking at the program that the value will always be an integer and it would be preferable not to have to write the cast. The second issue is that, since a collection needs to be able to hold objects of any type, there is no way to optimize the cases where an integer (or any other non-reference type) is being stored. This means that boxing and unboxing must be performed when adding items to and fetching items from the collection. A third and more general issue not directly exhibited in this program is that it is possible to insert an object that isn't an integer into this collection. This will result in the program failing at runtime.

All of these problems have one common cause – there is no way to state that the collection will only hold integers. If there was a way to state this, the need for the cast would disappear, it would become possible to avoid boxing and unboxing and the compiler can check that only integers are added to the collection and fail at compile time if any other object is added. Enter generics.

Generic collections

The snippet of code below is a re-write of the previous one, but this time generics have been used.

```
// Create a list that we'll put values of factorial in.
List<int> factorial = new List<int>();

// Compute first 10 factorials.
int fact = 1;
for (int i = 1; i <= 10; i++)
{
    fact *= i;
    factorial.Add(fact);
}

// Print them.
for (int i = 0; i < 10; i++)
{
    int value = factorial[i];
    Console.WriteLine(value);
}
```

Only three changes have been made to the program. Two of them are on the second line.

```
List<int> factorial = new List<int>();
```

The first thing to note is that the type List rather than ListArray is being used. List is a new type in C# 2.0 and is located in the new System.Collections.Generic namespace. The bigger difference is that the new generics syntax is used here. After the type List, the type int has also been mentioned, placed in angle brackets. This is called a type parameter (thus the name parametric polymorphism), and it specifies the type of value that will be stored inside the collection.

The third change to the program is that the cast has now disappeared. The compiler now knows that the list can only hold integers, and therefore there is no need to do a check at runtime. Further to this, it is now possible for the runtime to optimize the collection, removing the need for boxing and unboxing, so there is a performance gain to be had here too.

Finally, consider what happens if they following line is added after the first loop.

```
factorial[2] = "Monkey!";
```

With the original program, this would compile. However, the program would crash at runtime.

```
1
2
System.InvalidCastException was unhandled
       at CSharp2Examples.Program.Main(String[] args)
```

Using the generic List type means the compiler now has enough information to know that this line is invalid – only integers should be added to the collection. Trying to compile the program now results in an error message.

```
Cannot implicitly convert type 'string' to 'int'
```

Another example of a generic collection is the new Dictionary type. Imagine that you wanted to store a list of names of people (of type string) with their associated ages (of type int). A common solution would be to use a HashTable collection, with names as the keys and ages as the values.

```
HashTable ages = new HashTable();
```

The Dictionary class is a drop-in replacement for the HashTable, but with the types of the keys and values parameterized. It is instantiated as follows.

```
Dictionary<string, int> ages = new Dictionary<string, int>();
```

This generic type takes two type parameters – one for the keys and the other for the values. When more than one type parameter is provided, they are separated by commas, just as parameters being passed to a method would be. In fact, as we'll go on to see, the analogy to method parameters runs deeper when you implement generic types of your own.

Finally, it's worth making clear that it is possible to have a list of lists of integers or similar. This is written as you might expect:

```
List<List<int>> nested = new List<List<int>>();
```

Author's Note: I'm guilty of mixing terminology somewhat: generics with parametric polymorphism and generic type with parametric type. You're now either familiar with both sets of terminology or horribly confused. You'll likely find most users of C# prefer to talk about generics. Parametric polymorphism is the more academic term, but I think it captures the concept more clearly, which is why I've used it here.

Creating generic types

It is possible to create your own generic types in C# as well as using the supplied ones. This is useful when you have a class and that provides the same functionality for more than one type or group of types. For example, imagine I want a class with one property that takes some default value or object in the constructor. If another value has been stored through property it will then hand that back. If not, it will hand back the default value. I want to use this for integers, floats and some object types.

One option I have is to implement it as follows, using the object type so that anything can be stored in the class.

```
class DefaultProxy
{
    private object defaultObject;
    private object stored;

    // Set the default object we'll return if nothing has been
    // set.
    public DefaultProxy(object o)
    {
        defaultObject = o;
    }

    // Property for object o.
    public object target
    {
        get
        {
            return stored == null ? defaultObject : stored;
        }
        set
        {
            stored = value;
        }
    }
}
```

This will work, but has the same issue as the non-parameterized collections we saw earlier. Another option would be to write a specialized copy of the class for each type, for example IntegerDefaultProxy, FloatDefaultProxy etc. This is also a bad idea, since if a bug is found then every class needs to be updated.

Generics provide the solution. Turning the class above into a generic type involves two steps. The first is to parameterize the type – that is, state that the class takes a type parameter. This is done using the angle bracket syntax, and is analogous to writing the parameter list when defining a method.

```
class DefaultProxy<T>
```

We call "T" a type variable. Just like we can use a variable anywhere that we would use a value, we can use a type variable anywhere that we would use a type. Therefore, we can replace the use of the "object" type in the above code with the type variable, as shown below.

```
class DefaultProxy<T>
{
    private T defaultObject;
    private T stored;

    // Set the default object we'll return if nothing has been
    // set.
    public DefaultProxy(T o)
    {
        defaultObject = o;
    }

    // Property for object o.
    public T target
    {
        get
        {
            return stored == null ? defaultObject : stored;
        }
        set
        {
            stored = value;
        }
    }
}
```

And that's it – we've just created a generic type. It is instantiated by supplying a type to use for type variable "T"; in the examples below we create two instances of this generic type, one parameterized with int and the other with string.

```
DefaultProxy<int> proxy1 = new DefaultProxy<int>();
DefaultProxy<string> proxy2 = new DefaultProxy<string>();
```

It may help you to think of the word "int" being textually substituted wherever a "T" is found in the DefaultProxy class, though be aware that under the hood this is not what really happens.

Constraining type parameters

The generic types that we've seen so far allow themselves to be parameterized with any type. However, this constrains us to performing operations that only apply to all types. Consider the following program.

```
class MyGenericType<T>
{
    private T item;

    public void PerformSomeOperation()
    {
        foreach(object o in item)
        {
            // do something
        }
    }
}
```

Since not every type is enumerable (for example, an int is not), attempting to compile this class will result in an error:

```
foreach statement cannot operate on variables of type 'T' because 'T' does not
contain a public definition for 'GetEnumerator'
```

To get around this, we constrain the type variable “T” such that it must always implement the IEnumerable interface. This is done using the new “where” keyword along with the “:” syntax, as used for specifying which class to inherit from and what interfaces to implement.

```
class MyGenericType<T> where T : IEnumerable
```

The class will now compile, and parameterizing it with any type that implements the IEnumerable interface will succeed.

```
MyGenericType<List<int>> chimp = new MyGenericType<List<int>>();
```

However, supplying a type for the type parameter that does not implement IEnumerable, such as “int”, will give a compile time error.

```
The type 'int' must be convertible to 'System.Collections.IEnumerable' in order
to use it as parameter 'T' in the generic type or method
'CSharp2Examples.MyGenericType<T>'
```

Note that it is possible to constrain a type variable such that it must implement more than one interface or inherit from a certain parent class and implement one or more interfaces. To do this, just separate them by commas.

```
class MyGenericType<T> where T : IEnumerable, IComparable
```

To supply constraints for a second type variable, the “where” keyword must be placed again at the start of that constraint. Laying them out as I have here is not required, but perhaps makes the constraints clearer to read.

```
class MyGenericType<T, U> where T : IEnumerable, IComparable
                                       where U : ICollection
```

It is also possible to constrain a parameter to only accept reference or value types using the class and struct keywords. Note that these constraints must appear before any interface or inheritance constraints for a given type variable.

```
class MyGenericType<T, U> where T : struct
                                      where U : class
```

If when implementing a generic type you wish to instantiate a class whose type is given by a type variable, then you must be certain that it has a constructor. The new() constraint can be used to specify that any type that the generic type is parameterized with must provide the default parameterless constructor. Note that for a given type variable this constraint must come after any others, and that it is invalid to use it with the struct constraint.

```
class MyGenericType<T> where T : new()
```

Final thoughts on generics

If you're a C++ programmer then you've probably read through this thinking "I've seen this all before, it's just templates again". C++ templates also work to facilitate parametric polymorphism, however they are compiled away. That is, templates do not exist at runtime – the compiler uses the template to produce classes as needed, one for each type the template is parameterized with. It's somewhat like an extension of C's macros – it really is textual substitution.

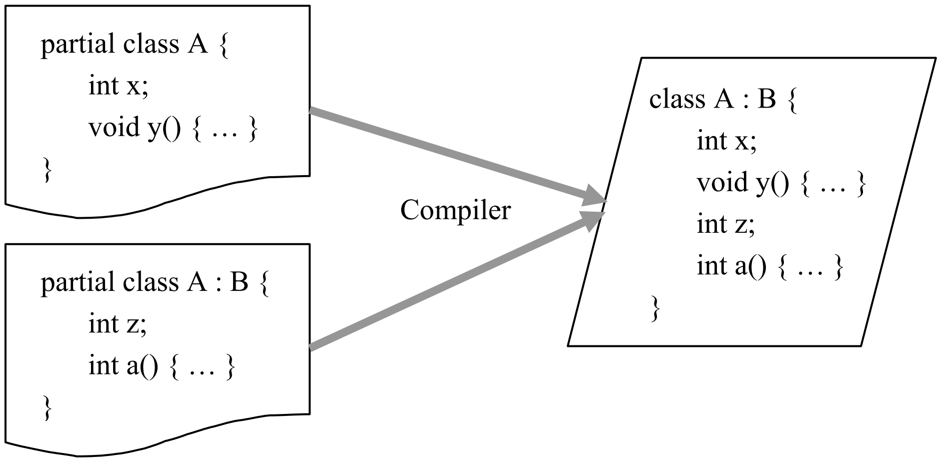
Java has also gained generics, with syntax somewhat similar to that seen in C#, but as with C++ templates generics are compiled away. This was done so that the runtime (the JVM) did not have to be modified to support generics. However, this introduces two problems. The first is that reflection (looking at types at runtime) does not recognize type parameters. The second is that you lose the performance benefit, since the generics implementation in the compiler actually just replaces every type variable with Object and then inserts casts as required. Therefore parameterizing on a primitive (non-reference) type leads to boxing and unboxing.

In .Net, the runtime has been modified to be aware of generics. While this breaks backward compatibility with previous versions of the runtime, it means that generics exist when you do reflection and that the optimizations you'd hope for with regard to eliminating boxing and unboxing are realized.

Author's Note: I was fortunate enough to be at a talk by Andrew Kennedy (from Microsoft Research) who worked on the .Net generics implementation. If you're interested in how generics are implemented in the .Net runtime, I greatly recommend reading his slides and/or papers. You can find his website at http://research.microsoft.com/~akenn/.

Partial types

When I first heard the term partial type I thought it was going to be something exciting. After all, parametric types are pretty cool. It turns out that they are actually quite boring by comparison, but thankfully trivial to understand. The use of the word "type" here is referring to types that you define yourself – that is, classes and structs (but not enumerations). The "partial" bit simply means that you can choose to define only part of the class at a time. Essentially, you're saying to the compiler "here is part of the definition of this class – some of the fields and some of the methods – but there may be other parts elsewhere".



The win from partial types is that a class definition can now be split over multiple files. The most common use of this is in Visual Studio or other GUI designers, where part of the class for a window is generated and maintained automatically. With partial types the human generated and machine generated code can go in separate files. Another use case is where two programmers want to work on different methods of the same class at the same time. If two aspects of what the class does can be cleanly pulled apart, then each aspect can be put into a partial class across separate files. It may also be useful for when a class gets large and there is a sensible way to split it up, but you should probably be asking whether one class is doing too much if you're reaching that stage.

The syntax is simple – to specify that other parts of a class may be defined elsewhere just add the new "partial" modifier. For example, imagine we had a class that defined an attendee at a party. We may wish to separate out the different types of things a partygoer can do into separate files. For example, in the file "Drinking.cs" we might have:

```
partial class PartyGoer
{
    private int drinksHad;
    private bool drunk;

    public void HaveADrink(string drinkName)
    {
        // ...
    }
}
```

Whereas the file "Relationship.cs" might perhaps contain:

```
partial class PartyGoer
{
    public string[] FindAttractivePeople()
    {
        // ...
    }

    public void ChatUp(string name)
    {
        // ...
    }
}
```

Note that the private fields defined in one fragment of the partial class are visible in other parts too, since really they are the same class. Therefore in “Relationship.cs” it is perfectly valid to write a method that accesses a private field from the fragment of the class defined in “Drinking.cs”.

```
public void AskToMarry(string name)
{
    if (drunk)
    {
        // Ask!
    }
    else
    {
        throw new ThatIsAReallyStupidIdeaException();
    }
}
```

Partial classes are compiled away – they do not exist at runtime. The compiler collects all of the fragments of the class together and compiles it as if it had all been specified together. For this reason, all fragments of a class must be in the same assembly. Note that it is not required to specify all interfaces that are implemented by the class every time that a part of it is defined – the compiler will take the union of all those mentioned across the partial classes. Therefore, if you try to inherit from different classes in different parts of the class, you’ll get a compile time error. Always remember that it really is just one class with its definition split up.

Partial classes have the potential to make a program more understandable by more clearly splitting up functionality. However, there is a risk that they will actually make debugging and understanding programs harder because you can no longer see right away from the code what a class inherits from and what interfaces it implements – you have to find all of the fragments of the class to be sure. All of the fields and methods are not in one place either, which means more searching through multiple files.

I’m certainly not suggesting that partial classes should not be used, but I would encourage carefully considering whether their use is going to make life easier or harder for anyone who has to maintain the code after you.

Author's Note: It occurs to me, and perhaps to you too, that while partial classes seem to do a good job of splitting up functionality, they do little to help with re-use. Object orientation achieves both – by splitting functionality between classes, we gain re-usability since a class can then be use elsewhere. You might be able to imagine something like a partial class, but that stands alone from any particular class and can be "merged into" other classes as needed. This isn't a new idea – it has been done in other languages already under names such as roles and mixins. I don't believe it is on the horizon in C#, though.

Nullable types

When dealing with reference types it is possible to set the reference to null, indicating that there is no object being referenced. This can be used to indicate that, for example, no useful data was available or accessible. You can imagine this situation when fetching data from a database – if the query fails to find a row, a null reference can be returned to indicate this; otherwise, a reference to object corresponding to the row that was found can be returned.

Now imagine a situation where we're doing a query but just want one value back, and so our database fetch method will just return, for example, an integer. A problem arises here, for with value types there is no null value. One solution, if it is an integer that is being fetched, is to make a value of zero mean "nothing was found". This is not always possible, however – sometimes any integer is valid and we need another way to signal that there is no value.

Nullable types address this problem by allowing value types to take a null value. To specify that a value type should be nullable, simply place a question mark after the name of the type:

```
int? x = null;
double? y = 5.2;
```

Notice that now as well as being able to assign a value, we can also assign null. To test whether a nullable variable has been given a value or if it is null, either test it for equality to null as you would with a reference or test its boolean HasValue property, which will be set to false if the variable is null.

```
if (x != null)    // Alternatively, if (x.HasValue)
{
    // It's not null.
}
else
{
    // It is null.
}
```

You can perform arithmetic and logical operations on nullable value types just as you would on their non-nullable variants. This behaves exactly the same when the variables are not null.

```
int? x = 6;
```

```
int? y = 7;
int? z = x * y; // z is now 42
```

However, if a null variable gets evaluated while evaluating some arithmetic or logical expression, the result of that whole expression becomes null.

```
int? x = 6;
int? y = null;
int? z = x * y; // z is now null
```

Finally, there is some extra syntax for assigning a default value if a nullable variable is null. This is the new null coalescing operator, which is spelt "??". It is useful when assigning a nullable type to a non-nullable type.

```
int? x = 5;
int? y = null;
int a = x ?? 0;  // a is 5
int b = y ?? 0;  // b is 0
```

Directly assigning a nullable variable to a variable of the equivalent non-nullable type is a compile time error.

```
int? x = 5;
int a = x;  // Ooh, naughty.
```

You can insert a cast, which stops the compiler complaining:

```
int? x = 5;
int a = (int) x;  // OK
```

However, if x was null:

```
int? x = null;
int a = (int) x;
```

Then it will compile, but an exception will be thrown at runtime.

```
System.InvalidOperationException was unhandled
  Message="Nullable object must have a value."
```

Essentially casts convert nulls to exceptions, which may be useful from time to time, though you might be able to throw a more helpful exception explicitly.

The question mark syntax and null coalescing operator are really just syntactic sugar. Under the hood this compiles down to using the Nullable type, which is a generic value type in the standard .Net class library. Therefore other .Net languages without special support for nullable types can still work with them. Here is a short example of using the Nullable type directly, with the prettier C# commented in to the right.

```
Nullable<int> x = null;                   // Same as int? x = null;
int a = x.HasValue ? x.Value : 0;   // Same as int a = x ?? 0;
```

Nullable types are another example of how generics can be put to good use.

Anonymous methods in event handling

Delegates in C# enable us to call methods indirectly. That is, instead of knowing the name of the method that is being called, we have a reference to it and make the call through the reference. Delegates form the basis of the .Net event system, and a common usage pattern is to write some method to handle an event…

```
public void MyButton_OnClick(object sender, EventArgs e)
{
    // Handle the event…
}
```

…and then add it to the multicast delegate for that event.

```
myButton.Click += new EventHandler(MyButton_OnClick);
```

Usually the method that handles the event (MyButton_OnClick in this case) is only ever called when the event occurs - that is, it is only ever called using the delegate and not directly using its name. Furthermore, many event handling methods are only a few lines of code long. Also, it would be good to be able to somehow draw the handler method and its subscription to the event closer together. Anonymous methods address all of these observations and more by enabling us to create a delegate type and supply the implementation for the method that the delegate will reference immediately afterwards. The syntax for an anonymous method is as follows.

```
myButton.Click += delegate(object sender, EventArgs e)
{
    // Handle the event...
};
```

Notice that the first line looks somewhat similar to the subscription to the event handler, but instead of specifying which method to subscribe to the handler, a new delegate type is being created using the keyword "delegate". Immediately following the creation of the delegate type is a method body, terminated by a semicolon after the closing curly bracket, which is easy to forget. The parameters for the methods will be accessible through the names specified in the delegate type definition, as demonstrated below.

```
myButton.Click += delegate(object sender, System.EventArgs e)
{
    MessageBox.Show(e.ToString());
};
```

Adventures with anonymous methods

So far anonymous methods may appear to be little more than a neat trick to reduce the amount of code that needs to be written for event handlers. One of the things makes them somewhat more powerful than this is that the local variables of the enclosing method are visible inside them. Consider the following program.

```
// Create a list containing the numbers 1 to 10.
List<int> numbers = new List<int>();
for (int i = 1; i <= 10; i++)
    numbers.Add(i);

// Add them all together.
int sum = 0;
numbers.ForEach(delegate(int x) { sum += x; });
Console.WriteLine(sum);  // Prints 55
```

The second line from the bottom contains the interesting use of an anonymous method. The ForEach method takes a delegate reference and for each value in the collection makes a call through that delegate. Note that the delegate reference must be of a delegate type that takes one parameter (of type int in this case). Here an anonymous method is defined by following the delegate definition with a chunk of code in curly braces. What is perhaps somewhat surprising is that we can use the variable “sum” inside the anonymous method. In the above example we increment it by the value that was passed. Essentially this is just a complicated way to write a foreach loop, but the technique is far more general and can be used in a wide variety of situations.

You might at this point be wondering what happens if you return a delegate that references an anonymous method that uses one of the local variables in its enclosing method. Surely local variables only live as long as the method that is calling them is executing, and thus the variable referred to by the anonymous method will no longer exist? In fact, this is not the case. Consider what the following program will do when you run it.

```
using System;
using System.Collections.Generic;

namespace CSharp2Examples
{
    // A new delegate type that takes no parameters and returns
    // an integer.
    delegate int Counter();

    class Program
    {
        static Counter GetCounter(int start)
        {
            // Counter variable outside of the anonymous method.
            int count = start;

            // Return our counter method.
            return delegate()
            {
                count++;
                return count;
            };
        }

        static void Main(string[] args)
        {
            // Create a counter starting at 15.
            Counter c = GetCounter(15);

            // Loop 5 times calling the anonymous method.
            for (int i = 0; i < 5; i++)
            {
                int value = c();
                Console.WriteLine(value);
            }
        }
    }
}
```

The program will give the following output:

```
16
17
18
19
20
```

The local variable “count” is somehow being kept around. Under the hood the compiler is actually doing a fairly elaborate piece of analysis and transformation, creating an anonymous nested class and placing any locals that could “escape” into that, so when it comes to runtime they are not really local variables anymore. Thinking of the anonymous method as if it were capturing the locals that it uses in its enclosing method probably provides a more helpful way of thinking about what is going on here, though. This process is known as taking a closure.

Author's Note: If this section on more advanced uses of anonymous methods has made you scratch your head somewhat, you’re probably not alone. Most programmers are used to the idea of passing data around, but fewer are as used to or comfortable with the idea of passing references to chunks of code around. This is sometimes referred to as higher order programming, and is commonly done in the functional programming paradigm. Concepts such as closures and parametric polymorphism have also been popular in functional programming for some time – it’s interesting to see them continue to break into more mainstream languages.

Final thoughts on C# 2.0

This chapter hasn’t covered every new feature in C# 2.0, but it has explored four of the most major additions that will enable you to develop more robust and efficient solutions, usually with less effort. I hope this chapter has not just explained the features, but also given you a feel for where they can help and what kinds of problems they are applicable to. Remember that the newest toy in the box isn’t always the best one to help solve your problem – but don’t be afraid to try out the new toys either!

16. The Road Ahead

Learning More

To fully cover the .Net framework and the C# language would need many more pages than this book has. However, we hope that it has equipped you with a sufficient understanding of the C# language and the .Net platform that, along with additional documentation, you will be ready to take on real world programming tasks.

The primary source of documentation for the .Net class library is the MSDN (Microsoft Developer Network). This documents every class in the library, detailing each of its members and often providing usage examples. This can be found at http://msdn2.microsoft.com/en-us/library/ms229335.aspx.

Programmer's Heaven has an area of the site dedicated to C#. This contains hundreds of articles on a wide range of topics, from attributes to XML, as well as a listing of tools and source code that can be freely downloaded. There are many more C# resources available online too – just search for them!

Getting Help

Ran into a problem? Got stuck? Don't worry – help is available. There are a number of places that you can discuss C# and the .Net framework with others, including messages boards and IRC channels. Programmer's Heaven has a C# message board, located at http://www.programmersheaven.com/c/MsgBoard/wwwboard.asp?Board=37.

Book.revision++

This is the first version of the C# school e-book, but we hope that it will not be the last. Future updates may include details of new language features in the forthcoming version 3 of the C# language as well as additional chapters on real world usage of C# and .Net, perhaps including ASP.Net, web services and XML.

What really shapes the future of this book is you – the reader – so please do get in contact with us and let us know what you would like to see improved or added. Programmer's Heaven can be reached by email at info@programmersheaven.com and this e-book has a page with the latest news about it located at http://www.programmersheaven.com/2/CSharpBook

Good Luck!

The author and the editors of this book wish you the best of luck with learning and developing applications using C# and the .Net platform. Happy coding!