

Programming in C Sharp Using Net Framework

Index

Preface

Block 1

Unit 1: Introduction of C#

Introduction

C# Basics

Microsoft.Net

Some common Difference between C# & others OOPs languages

Scope of Variables

Unit 2: Operators & Expressions in C#

Introduction

Operator and Operands

Types of Operator

Arithmetic Operator

Logical Operator

Relational Operator

Increment & Decrement Operator

Assignment Operator

Special Operator

Conditional Operator

Bitwise Operator

Precedence of Operator

Type Conversion

Unit 3: Control Statement in C#

Introduction

Statements

Types of Control Statements

Branching Control Statements

Jumping Statement

Looping/Iteration Control Statement

Summary

Exercise

Unit 4: Structures and Array

Introduction

Structures

Nested Structures
Difference between Classes & Structures
Enumerations
Enum Type Conversion
Arrays
Creating an Array
Types of Array
1-D Array
2-D Array
Dynamic Array

Block 2

Unit 6: Inheritance and Polymorphism

Introduction
Inheritance in C#
Types of Inheritance
Single Inheritance
Multilevel Inheritance
Multiple Inheritances (Interface)
Polymorphism in C#
Abstract Base Classes & Methods

Unit 7: Overloading and Overriding

Introduction
Operator Overloading
Unary Operator Overloading
Binary Operator Overloading
Method Overloading
Method Overriding

Unit 8: Event & Delegates

Introduction
Delegates
Declaration of Delegates
Instating a Delegates
Delegates
Events

Unit 9: Properties & Indexer

Introduction
Properties
Types of Properties
Indexers

Block 3

Unit 10: Assemblies & Attributes

Introduction
Assemblies
Creating Assemblies
Types of Assemblies
Assemblies and the Internal Access Modifiers
Customizing an Attributes
In-built Attributes
Using Win32 API
Creating Custom Attributes
Versioning
Reflection

Unit 11: Directive and Debugging

Introduction
Error
Types of Errors
Finding Errors
Preprocessor Directive
Using Debuggers

Unit 12: Exception Handling

Introduction
Exceptions
Types of Exceptions
Raising Exceptions using Throw
The try statement
The try-catch statement
The finally statement

Unit 13: Threading in C#

Introduction
Definition of Threading

Thread Creation
Synchronizing of Thread
Multithreading

Block 4

Unit 14: Namespace

Introduction
Namespaces
Qualified Naming
Using Namespace Directives
Using Alias Directives
The .NET Base Class Library

Unit 15: Graphics

Introduction
Graphics Object
Brushes
Pens
Images
Text and Drawing

Unit 16: Window Based Application in C#

Introduction
Component of a Form
Labels
Buttons
Text Boxes
List Boxes
Check Boxes and Radio Buttons
C# Control
Method
Properties
Event

Unit 17: Web Based Application in C#

Introduction
ASP.NET Web Application
Creating a Web Based Project with C#

Block 5

Unit 18: Files and Database Programming

Introduction

Files and Directories

Creating a File

Deleting a File

Database

MS-Access

Inserting Data in MS-Access

Deleting Data in MS-Access

Updating Data in MS-Access

Retrieving Data in MS-Access

SQL

Inserting Data in SQL

Deleting Data in SQL

Updating Data in SQL

Retrieving Data in SQL

XML

Inserting Data in XML

Deleting Data in XML

Updating Data in XML

Retrieving Data in XML

Converting SQL Data in XML format

Unit 19: COM form C# Application

Introduction

Using COM Component from C#

Unit 20: C# and .NET Security

Introduction

Security Role

Code Security

Code Security Policy

Code Security Permissions

User Security

Project:

Console Based Project

Railways Reservation System

Window Based Project

Library Management System

Chapter wise Lecture Notes

Bibliography

Index

Introduction of C#

1.1 INTRODUCTION

As we know that computer and software are the backbone of technology. A computer cannot understand person's spoken language like English. So that, we must familiar with any language that computer can easily understand. This is when programming comes in the picture. Programming is the act or process of planning or writing a program. When a programmer writes a code actually he is using any programming language. Hundreds of languages came but only some languages were popular due to their performance.

Since the invention of the computer many programming languages approaches have been tried. These include techniques like top down, bottom up, modular, structured & OOPs.

In continuation of change in technology software industry or program developers need some recent languages. Today Client site application & Server site application, **Mobile application** development is demand of software industry. So a language was needed for development, then a language came in existence which have many features of C, C++, Java, VB etc. This language helps programmer to create secure, robust, portable, distributed object oriented applications for the real world and global internet.

Can you guess it? Yes, it is C# language which supports maximum features of Object Oriented Programming & also helps us for easy doing programming in Console and window application. Learning a programming language is much like learning any other skill. It also requires lots of practices. In this book we teach you language C# learning is same as learning of Hindi & English. Here, we use basically two important terms of application development - The.Net and C#.

C# language also focuses on the Windows based application programs, visual programming concepts, interactive graphics fundamentals, and database connectivity concepts.

This book includes topics such as Windows Forms, Windows Controls, Windows programming data access with ADO .NET, and handling data access and data manipulation in codes and also all console based program.

The book provides deep insights into the .NET programming concepts and is designed to enhance the programming skills of the users of C#. This book is a practical introduction to programming in C# utilizing the services provided by .NET to build Web-based services and others. Here we introduces C#'s advanced object-oriented capabilities

early – helping you make the most of them to create software with unprecedented efficiency and power and also covers data types, formatting and conversions, exceptions, interfaces, collections, the callback mechanism, and attributes of languages .

Finally, this book will cover all aspect of C# programming language. So enjoy programming very easily. Here our team will guide you beginning to end of C# programming and we hope you are comfortable after reading this book.

The main objective of this book is-

- To understand the basic concept of C#.
- To understand the concept of .net framework
- To understand data types
- To understand data base connectivity
- To develop complete application using C# and .net
- To obtain a basic idea of development of future generation of application and learn many more with executed codes.

1.2 C# BASICS

A programming language is a language for expressing instructions to a computer. C# (pronounced C Sharp) is a very new powerful type-safe standard object oriented programming language developed by Microsoft corporation which facilitate programmer to create different type of secure and robust application that can easily run on Microsoft.Net framework a new technology provided by Microsoft.

As per definition given by Microsoft “C# is a simple, modern, object oriented and type safe programming language derived from C & C++ languages and easy to work. C# also combines the high productivity of C, C++, Visual Basic and Java. C# may well become the dominant language for building applications on Microsoft platforms.

C# used for creating traditional Windows client applications, XML Web services, distributed components, client-server applications, database applications, and more. Additionally Visual C# 2010 provides an advanced code editor, convenient user interface designers, integrated debugger, and many other tools to make it easier to develop applications based on version 4.0 of the C# language and version 4.0 of the .NET Framework.

C# is an international standard programming language used to create instructions that direct the computer about what to do, when to do it, and how to do something. It models real world well.

Programmers can develop different types of applications with C# language like-

1. Window based client applications
2. XML web services
3. Server side application
4. Dynamic web pages
5. Web services
6. Industrial application
7. Mobile applications
8. Distributed components
9. Database applications and many more.

Here we can say that

C# =most features of C+, C++, VB & Java

1.2.1 Versions of C#

1. C # 1.0
2. C # 1.2
3. C # 2.0
4. C # 3.0
5. C # 4.0 is the new version of the C# Programming language.
6. C # 5.0* (*->awaited)

1.2.2 Characteristic of C#

As other programming languages, C# has several characteristics as follows -

1. Class & object
2. Polymorphism
3. Inheritance
4. Dynamic binding
5. Overloading
6. Struct & enum
7. Boxing & Unboxing
8. Exception handling
9. Rapid action development(RAD)
10. Window application
11. Garbage collections
12. Multithreading
13. Null able types

In addition of these, C# makes easy to develop software with the help of following-

1. Delegates
2. Properties
3. Attributes
4. Inline XML documents
5. Language integrated query link (LINQ)

1.2.3 Special Characteristics of C#

1. Application component written in C# can be combined with components written in other language.
2. Compiled into MSIL and runs on CLR and is compiled just -in-time as each method is used the first time in program.
3. The CLR adds another layer between the operating system and application.
4. C # also provides Indexes (like arrays)

1.3 MICROSOFT .NET

The Microsoft .NET is an advance method of programming technology that greatly simplifies application development, both for traditional, proprietary applications and for the

emerging paradigm of Web-based services. .NET is a complete restructuring of Microsoft's whole system infrastructure and represents a major learning challenge for programmers developing applications on Microsoft platforms.

C# is a major compiler of .net framework. But learning the new programming language is only part of the challenge. The much greater challenge is learning the .NET Framework and all its capabilities.

The .net framework is a new technology for building the future generation of Application development designed by Microsoft Corporation for developing Window based Software's. This framework provides a complete and integral environment designing & executing console applications, window application, web applications, web services, class libraries and many more. The .net have a big library that can be utilized by the programmers for other languages. The .net frameworks execute in a Common Language Runtime (CLR) a software environment.

The .net = Class Library +CLR

.net framework:



The .NET Framework has two main components:

1. The common language runtime(CLR)
2. The .NET Framework class Library

The common language runtime is the foundation of the .NET Framework or you can say it's a heart of .NET Framework. The class library, the other main component of the .NET Framework, is a comprehensive, object-oriented collection of reusable types that you can use to develop applications. The .NET Framework provides an object-oriented programming model for multiple languages like Visual Basic, Visual C#, Visual C++ etc.

The following sections describe the main components and features of the .NET Framework in greater detail.

1.3.1 MS Visual Studio is a large IDE for .Net Framework.

The .NET Framework enable us to accomplish a range of common programming tasks, including tasks as string management, data collection, database connectivity, and file access. In addition to these common tasks, the class library includes types that support a variety of specialized development features.

We can use the .NET Framework to develop the following types of applications and services:

1. Console applications.
2. Windows GUI applications (Windows Forms).

3. Windows Presentation Foundation (WPF) applications.
4. ASP.NET applications.
5. Web services.
6. Windows services.
7. Service-oriented applications using Windows Communication Foundation (WCF).
8. Workflow-enabled applications using Windows Workflow Foundation (WWF).

The following sections describe the main components and features of the .NET Framework in greater detail.

1.3.2 Versions of .net framework are-

1. the .net framework 1.0
2. The.net framework 1.1
3. the.net framework 2.0
4. the.net framework 3.5,
5. the.net framework 4.0
6. the .net framework 4.5 *(awaited)

1.3.3 Language in .Net framework:

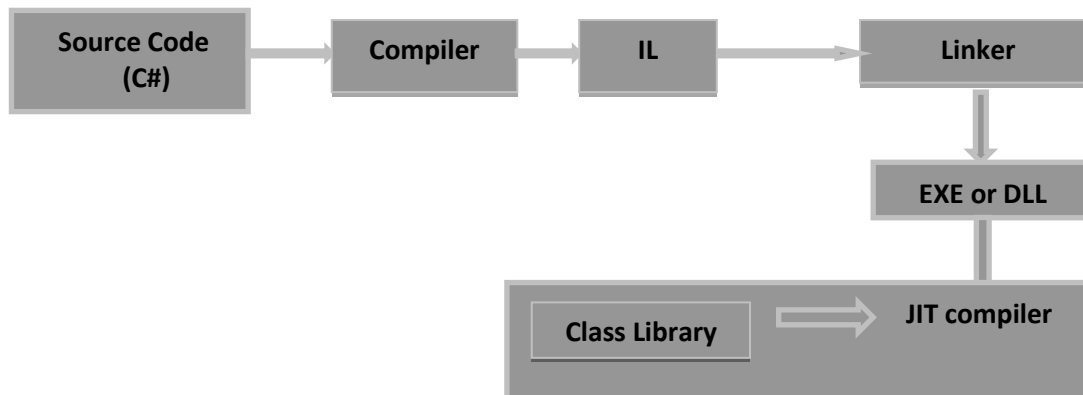
1. ADO.Net
2. ASP.Net
3. VB.Net
4. C#.Net

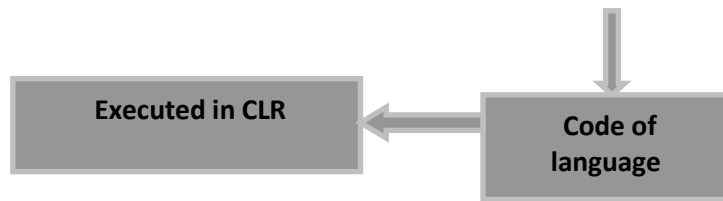
1.3.4 Software or tools for compilation & execution of C# & .Net applications:

Select any one in following as per need -

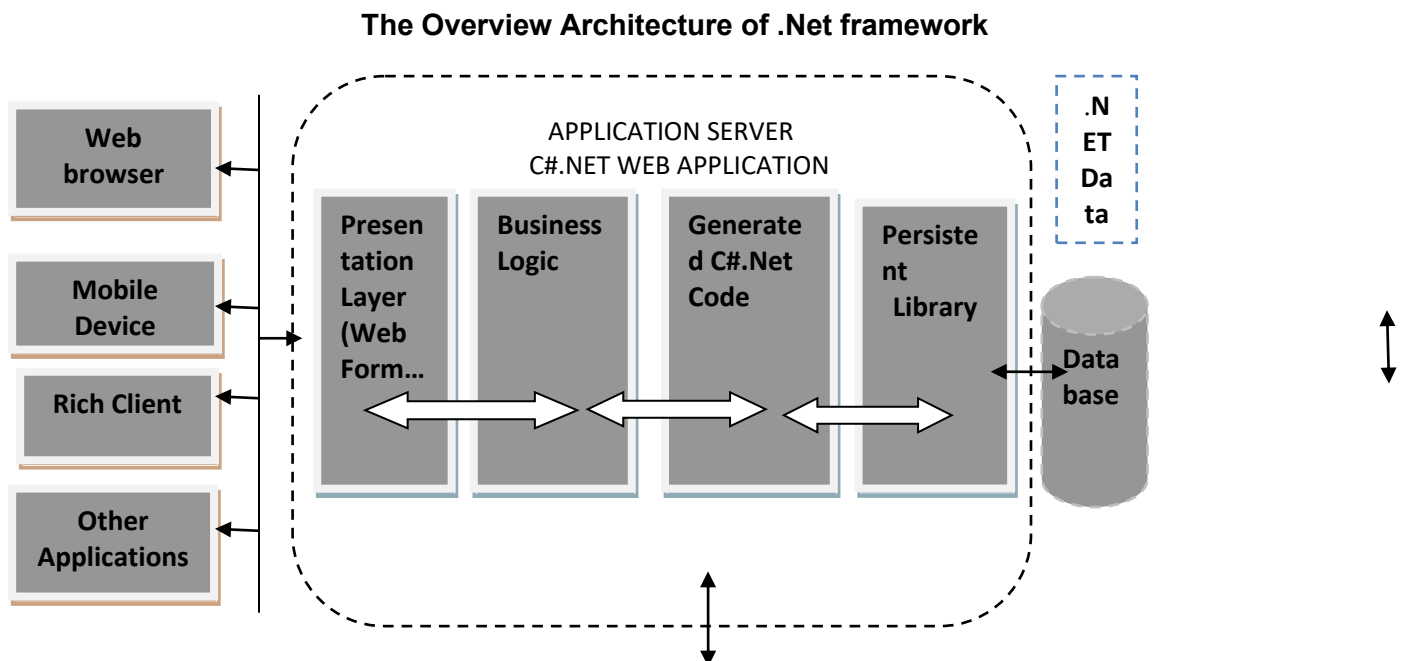
1. Visual Studio .Net 2002
2. Visual Studio .Net 2003
3. Visual Studio 2005
4. Visual Studio 2008(for MS- Window 7)
5. Visual Studio 2010
6. Visual studio 2010
7. Visual studio 11 (suitable for MS-Window 8 & win server 8 versions)
8. Visual C# 2010 Express
9. Sharp developer
10. The.net framework SDK.

1.3.5 Execution process in .net framework-





(Fig 1.1)





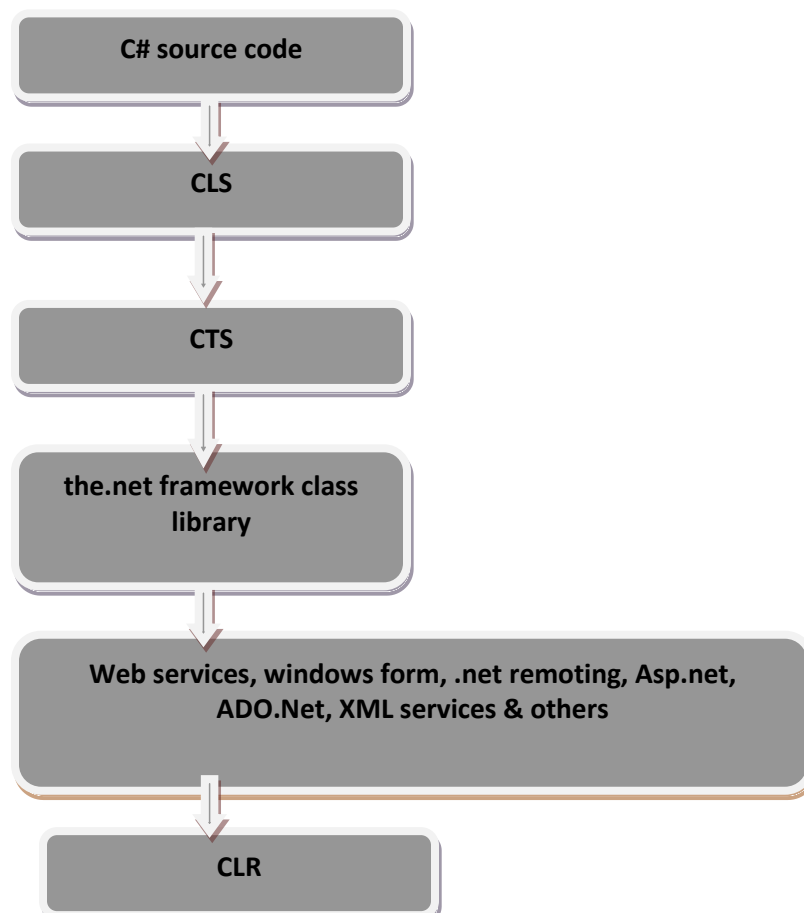
(Fig 1.2) General structure of C # data flow

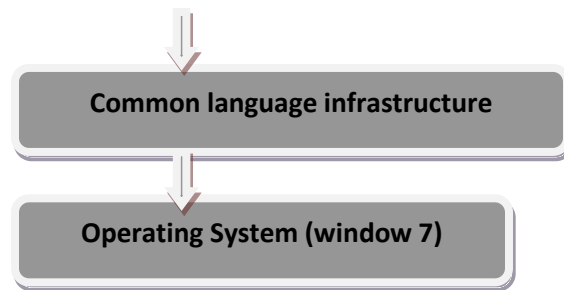
1.3.6 Common Language Infrastructure (CLI)

Common Language Infrastructure or CLI is the most important component of the .NET Framework. The purpose of the CLI is to provide a language-agnostic platform for application development and execution, including, but not limited to, components for exception handling, garbage collection, security, and interoperability. Microsoft's implementation of the CLI is called the Common Language Runtime or CLR.

The CLR is of four parts:

1. Common Type System (CTS)
2. Common Language Specification (CLS)
3. Just-in-Time Compiler (JIT)
4. Virtual Execution System (VES)





(Fig 1.3)

1.3.7 Component of .Net Framework

There mainly two components of .net –

1. CLR- The main components of CLs are-

1. MSIL
2. JIT
3. GC
4. Assemblies
5. CAS
6. Once click

2. Class Library

This table illustrates the components of the .NET framework in details-

Windows Forms, Web Forms, Web Services, etc. (Developed in .NET compliant languages)
.NET Framework Base Classes (ADO.NET, XML, Threading, I/O, Network)
Common Language Runtime (Memory Management, Common Type System, Lifecycle Monitoring)

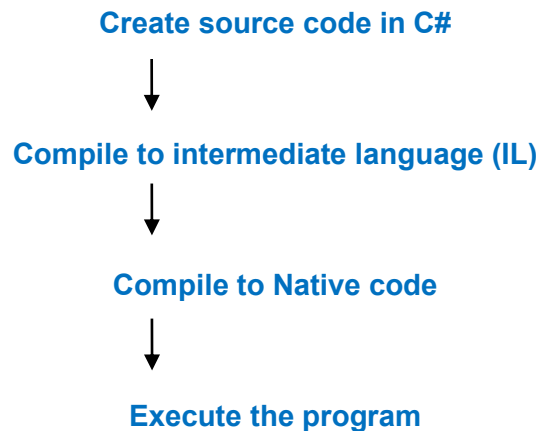
1.3.8 The Common Language Runtime (CLR)

The Common Language Runtime (CLR) is the virtual machine environment that all .NET languages run in. It is a managed execution environment to provide several services for running programs.

The common language runtime manages memory, thread execution, code execution, code safety verification, compilation, and other system services. The runtime enforces code access security. The runtime also enforces code robustness by implementing a strict type-and-code-verification infrastructure called the *common type system (CTS)*. The CTS ensures that all managed code is self-describing. The CLR also handle the automatically memory management. Common language runtime provides many standard runtime services, managed code is never interpreted. A feature called just-in-time (JIT) compiling enables all managed code to run in the native machine language of the system on which it is executing.

The CLR execution process is summarizes as:

Follow following steps-



(fig 1.4)

a) Create source code in C#-

The CLR first create a source code for source program which is to be executed. C# code is compiled with the csc compiler.

b) Compile to intermediate language-

Source code written in C# language is compiled into an assembly language like code called intermediate language (IL) that conforms to the CLI specification. The IL code and resources, such as bitmaps and strings, are stored on disk in an executable file called an assembly with an extension of .exe or .dll. It also called Microsoft Intermediate Language (MSIL).

c) Compile to native code-

In this stage, IL code must be compiled with the native machine code for the computer on which it's running.

d) Execution of program-

After compilation is performed by JIT, it can execute in the CLR's managed environment. During the execution process, the common language runtime manages memory, thread execution, code execution, code safety verification, compilation, and other system services.

Let us now take a look at other important features of .NET

1.3.9 Common Type System (CTS)

The common type system (CTS) defines how types are declared, used and managed in the runtime. CTS are an important part of the runtime's support for cross-language integration.

The common type system performs the following functions:

1. It provides a framework that helps enable cross-language integration, type safety, and high performance code execution.
2. It provides an object-oriented model that supports the complete implementation of many programming languages.
3. It defines a set of rules that languages must follow, which helps ensure that objects written in different languages can interact with each other.

The common type system supports two general categories of types:

1. **Values Type:** Value types directly contain their data, and instances of value types are either allocated on the stack or allocated inline in a structure.
2. **Reference Type:** Reference types store a reference to the value's memory address, and are allocated on the heap. Reference types can be self-describing types, pointer types, or interface types.

The common type system in the .NET Framework supports the following five categories of types:

1. Class
2. Structure
3. Enum
4. Interfaces
5. Delegates

The Common Language Specification (CLS) is a set of basic language features needed by many applications has been defined.

1. CLS is a subset of CTS.
2. The CLS provides a feature which helps ensure that objects written in different languages (out side of .NET Framework languages) can interact with each other.

1.3.10 The Base Class Library

The Base Class Library (BCL) is a standard library available to all languages using the .NET framework. The .NET Framework includes classes, interfaces, and value types that expedite and optimize the development process and provide access to system functionality.

The .NET Framework includes types that perform the following functions:

1. Represent base data types and exceptions.
2. Encapsulate data structures.
3. Perform I/O.
4. Access information about loaded types.
5. Invoke .NET Framework security checks.
6. Provide data access, rich client-side GUI, and server-controlled, client-side GUI.

The .NET Framework provides a rich set of interfaces, as well as abstract and concrete (non-abstract) classes. You can use the concrete classes as is or, in many cases, derive your own classes from them. To use the functionality of an interface, you can either create a class that implements the interface or derive a class from one of the .NET Framework classes that implements the interface.

The BCL is sometime referred to as the Framework Class Library (FCL), which is a superset including the Microsoft namespaces.

Followings are the same namespaces provides by the .NET Framework.

1.3.11 Standardized namespaces

1.3.11.1 System

This namespace include the core needs for programming. It includes base types like String, Date Time, Boolean, and so forth, support for environments such as the console, math functions, and base classes for attributes, exceptions, and arrays.

1.3.11.2 System.Collections

Defines many common containers or collections used in programming, such as lists,ques, hashtable , stack etc.

1.3.11.3 System.Diagnostics

Gives you the ability to diagnose your application. It includes event logging, performance counters, tracing, and interaction with system processes.

1.3.11.4 System.Globalization

Provides help for writing internationalized applications. "Culture-related information, including the language, the country/region, the calendars in use, [and] the format patterns for dates, currency, and numbers" can be defined

1.3.11.5 System.IO

Allows you to read from and write to different streams such as files or other data streams. Also provides a connection to the file system etc.

1.3.11.6 Non standardized namespaces

System.Configuration

Provides the infrastructure for handling configuration data.

System.Data

This namespace represents the ADO.NET architecture, which is a set of computer software components that can be used by programmers to access data and data services.

System.Drawing

Provides access to GDI+ graphics functionality, including support for 2D and vector graphics, imaging, printing, and text services.

System.Linq

Defines the IQueryable<T> interface and related methods, that lets LINQ providers to be plugged in.

1.3.11.7 System.Web

Provides various web related functionality. It enables browser-server communication and the creating XML Web Services. Most or all of these libraries are referred to as the ASP.NET architecture.

1.3.12 Language Interoperability (LI)

An important feature of the .NET Framework is language interoperability. It means program code written in one class can be execute in code written in another language.IL code produced by the C# compiler conforms to the Common Type Specification (CTS), IL code generated from C# can interact with code that was generated from the .NET versions of Visual Basic, Visual C++, or any of more than **32** other CTS-compliant languages. A single assembly may contain multiple modules written in different .NET languages, and the types can reference each other just as if they were written in the same language.

In addition to the run time services, the .NET Framework also includes an extensive library of over 4000 classes organized into namespaces that provide a wide variety of useful functionality for everything from file input and output to string manipulation to XML parsing, to Windows Forms controls. Some important features of Li are as under-

A class code written in one language can be used by another language. A class can be written in more than one language (they are supported by .net framework). By supporting more than one language means programmers need not to start from beginning of language to execute using .net application and .net remove the restrictions of language dependency and allow programmers to choose their language for development.

IL also makes a strong distinction between value types & reference types and also specifies the manner in which the data is stored in both manners.

1.3.13 Common Language Specification

Common language Specification (CLS) work with Common Type System (CTS) for ensuring language interoperability .CLS specifies a set of minimum standards that all language compiler targeting .net must support. It is designed to be a large enough and to include the language constructs which is needed by programmer .C# language by itself is CLS compliant to a large extent.

1.3.14 Microsoft Intermediate Language (MSIL)

The compilers first create a source code for source program which is to be executed. After compilation, the compiler produces an assembly language-like code called *intermediate language (IL code)*. It also called Microsoft Intermediate Language (MSIL).

MSIL includes instructions such as instructions for loading, storing, calling method and control flow of program etc.

A compiler produces metadata, which describe the information of the code, with the MSIL.

After that, MSIL code must be complied with the native machine code for the computer on which it's running. This task is performed with a just-in-time (JIT) compiler.

After compilation is performed by JIT, it can execute in the CLR's managed environment.

1.3.15 JUST –IN – TIME Compilation

After generating a MSIL code, it must be complied with the native machine code for the computer on which it's running. This task is performed with a just-in-time (JIT) compiler.

The CLR calls the JIT compiler to compile and delivery any methods that are not in memory during execution. This process continues as needed until the program is finished.

There are 3 types of JIT compilers:

1. **Econo-JIT:** The Econo JIT is used for fast compilation times and portability.
2. **Pre-JIT:** A Pre -JIT complies the code completely sometimes before the first execution.
3. **Standard-JIT:** Standard-JIT is used for normal execution mode for managed code.

The following diagram illustrates the compile-time and run-time relationships of C# source code files, the .NET Framework class libraries, assemblies, and the CLR.

1.3.16 Assemblies

An assembly is a unit containing IL code of a program. It is similar to a DLL file, but difference is that unlike DLL, an assembly is self-describing. Assemblies contain assembly

metadata that gives details of the assembly, type metadata describing the types, methods, etc, defined in the assembly and resources.

The intermediate MSIL code is housed in .NET assemblies, for the Windows implementation means a Portable Executable (PE) file (EXE or DLL). Assemblies are the .NET unit of deployment, version and security. The assembly consists of one or more files, but one of these must contain the metadata for the assembly. The complete name of an assembly contains its simple text name, version and public key token; it must contain the name, but the others are optional. The public key token is generated when the assembly is created, and is a value that uniquely represents the name and contents of all the assembly files.

1.3.17 Garbage collection

It is a program which is invoked by the CLR to free the memory that is not being used by the applications. Because of this technique the programmers no more need to take care of memory leakages, dangling pointers and cleanup of memory.

1.3.18 Benefits of .Net framework:

The .NET Framework has a number of advantages to developers.

1. Consistent Programming Model

With .NET accessing data with a VB .NET and a C# .NET looks very simple. Both the programs need to import the System. Data namespace, both the programs establish a connection with the database and both the programs run a query and display the data.

2. The .NET example explains that there's a unified means of accomplishing the same task by using the .NET Class Library. The functionality that the .NET Class Library provides is available to all .NET languages resulting in a consistent object model regardless of the programming language the developer for further uses.

3. Direct Support for Security

When an application accesses data on a remote machine or has to perform a privileged task on behalf of a no privileged user, security issue becomes important as the application is accessing data from a remote machine. With .NET, the Framework enables the developer and the system administrator to specify method level security.

4. It uses industry-standard protocols such as TCP/IP, XML, SOAP and HTTP to facilitate distributed application communications. This makes distributed computing more secure because .NET developers cooperate with network security devices instead of working around their security limitations.

5. Simplified Development Efforts

In Web applications, a developer with classic ASP needs to present data from a database in a Web page. He has to write the application logic (code) and presentation logic (design) in the

same file. ASP.NET and the .NET Framework simplify development by separating the application logic and presentation logic making it easier to maintain the code.

6. The design code (logic) and the actual code (application logic) is written separately eliminating the need to mix HTML code with ASP code. ASP.NET can also handle the details of maintaining the state of the controls, such as contents in a textbox, between calls to the same ASP.NET page. Another advantage of creating applications is debugging.

7. The .NET Framework simplifies debugging with support for Runtime. It helps us to track down bugs and also helps to determine how well an application performs. The .NET Framework provides three types of Runtime: Event Logging, Performance Counters and Tracing.

Easy Application Deployment and Maintenance

The .NET Framework makes it easy to develop applications. In the most common form, to install an application, all you need to do is copy the application along with the components it requires into a directory on the target computer. The .NET Framework handles the details of locating and loading the components an application needs, even if several versions of the same application exist on the target computer.

1.3.19 Security in .net framework

The.NET has its own security mechanism with two general features:

1. Code access security
2. Validation and verification

Code Access Security is based on evidence that is associated with a specific assembly. Code Access Security uses evidence to determine the permissions granted to the code. Other code can demand that calling code is granted a specified permission.

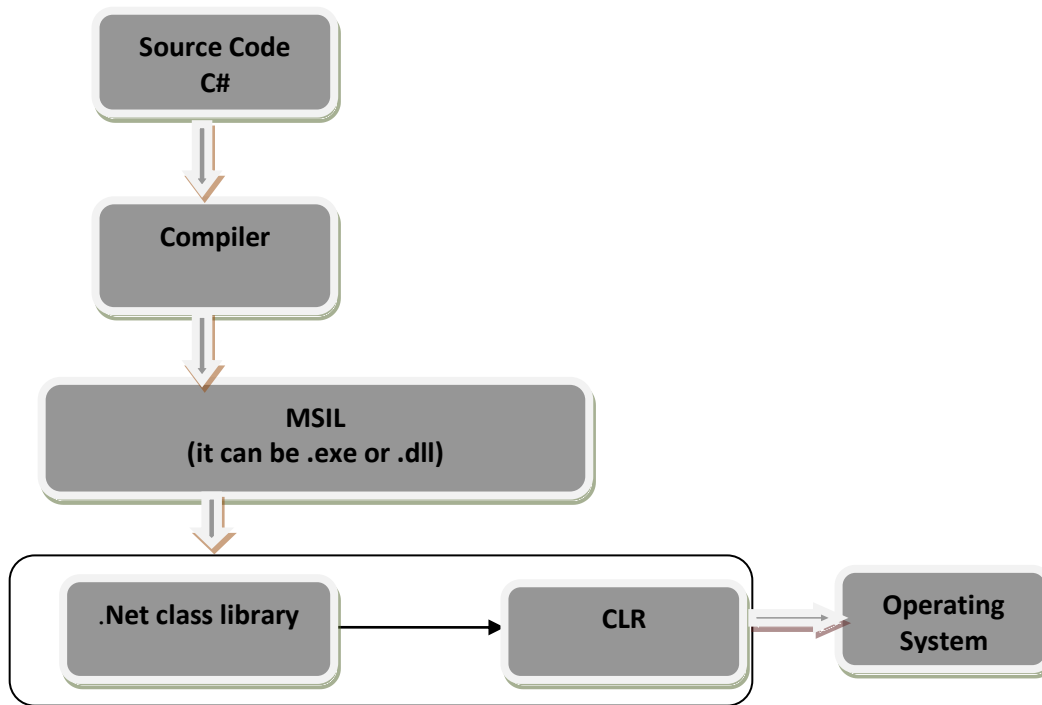
The demand causes the CLR to perform a call stack; every assembly of each method in the call stack is checked for the required permission; if any assembly is not granted the permission a security exception is thrown.

The.net languages are CLI programming language .Microsoft provides various such languages C#, F#, VB.Net and C++.CLI languages are programming language that are used to produce libraries and programs that conform to Common language infrastructure CLI . Generally .net call two main categories -

1. Type safe language (C#)
2. Dynamic language (Python).

Type safe languages are built on the .net common language runtime and dynamic languages are built on top of the .net dynamic language runtime.

Fig 1.6– Process of compilation & execution of C# program



1.4 Evolution of C# language

Language C# was introduced in 2000 by the Danish born Anders Hejlsberg a famous client server application developer. C# came in existence from C, C++, VB, Delphi, and Java.

Around 1997, Microsoft started a project that was internally known as Project 42. The name "Project 42" was most likely because DevDiv (the Microsoft Developer Division) is in the Building 42. There were several names proposed, one of which was the COM Object Runtime (COR), which is where the name for the mscorlib.dll assembly came from.

This is the assembly which contains all of the CLR's main types and is really the only one that must be loaded by every .NET application.

The codename of C# was project cool and was supposedly a "clean-room" implementation of Java. It was later changed to C# based on a musical scale. Just as C++ added the "++" to "C" since it was considered to be "adding to" or "one greater than" C, the sharp (#) on a musical scale means one semi-tone above the note.

And another Important in the history of C# was Microsoft's 1996 acquisition of Colusa Software. Colusa had released a product in 1995 called OmniVM. OmniVM was a virtual machine environment that offered two distinct advantages over early versions of Java.

First, by avoiding interpretation and using a virtual RISC architecture it provided near-native code execution performance. Second, it implemented robust 'application' isolation via a virtual memory manager.

This made it a very safe environment for running 'legacy' and 'mobile' code. OmniVM would form the basis for Microsoft's CLR- Common Language Runtime.

1.4.1 How to start Program Flow?

Let us start to do C# programming. First we start console applications.

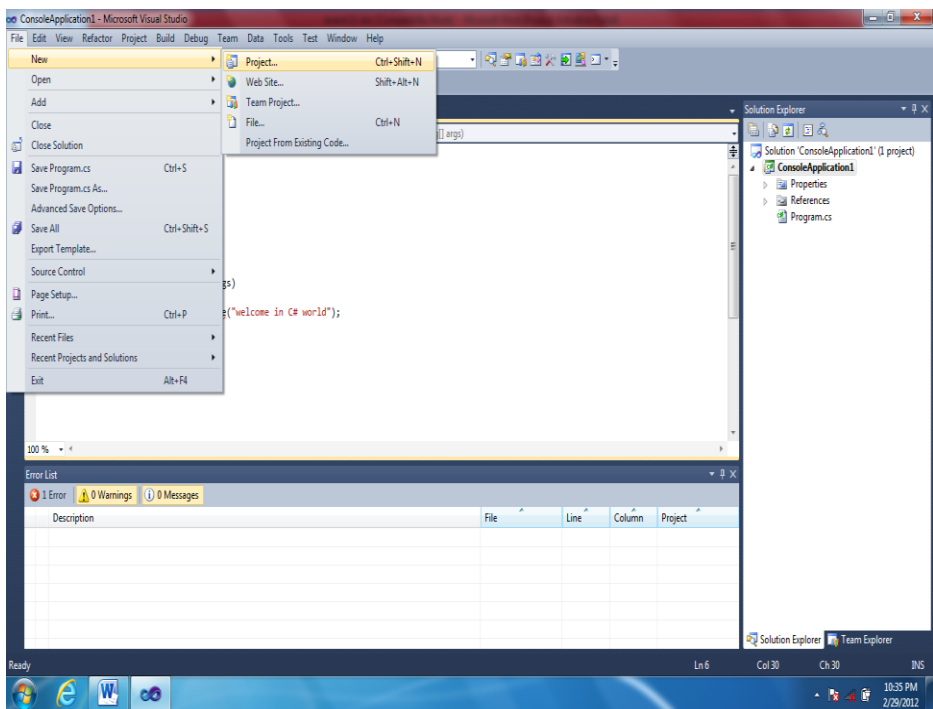
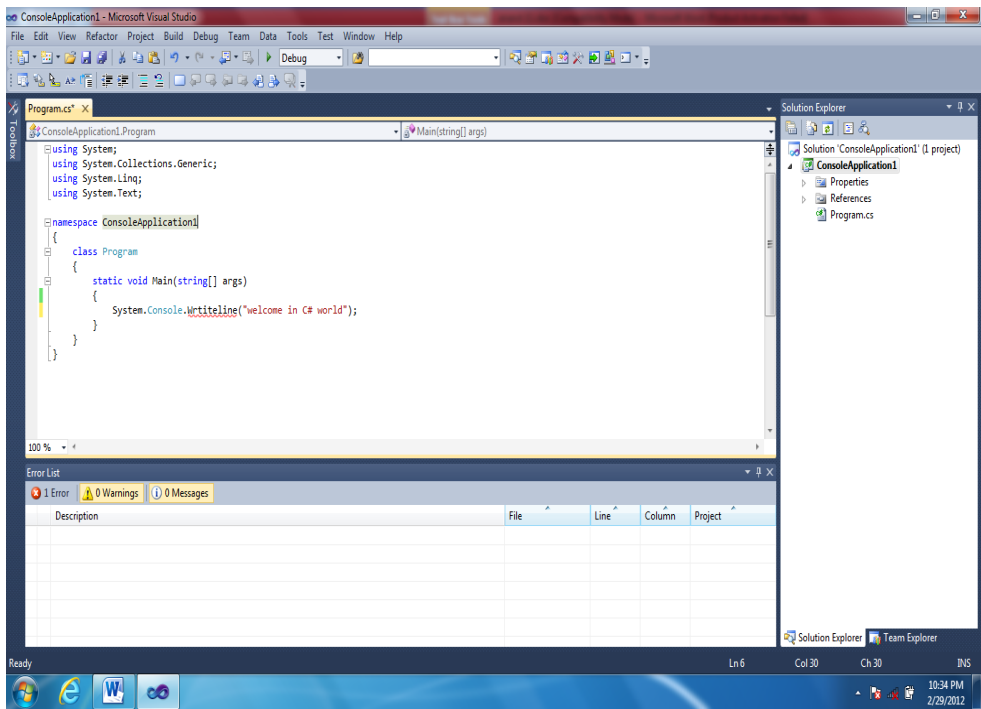
For writing complete C #programs follow following steps (to create and run a console application):

1. Start Visual Studio.(any version)
2. On the File menu, click New, and then click Project.
3. In the Templates Categories pane, expand Visual C#, and then click Windows.
4. In the Templates pane, click Console Application.
5. Type a name for your project in the Name field.
6. Click OK.

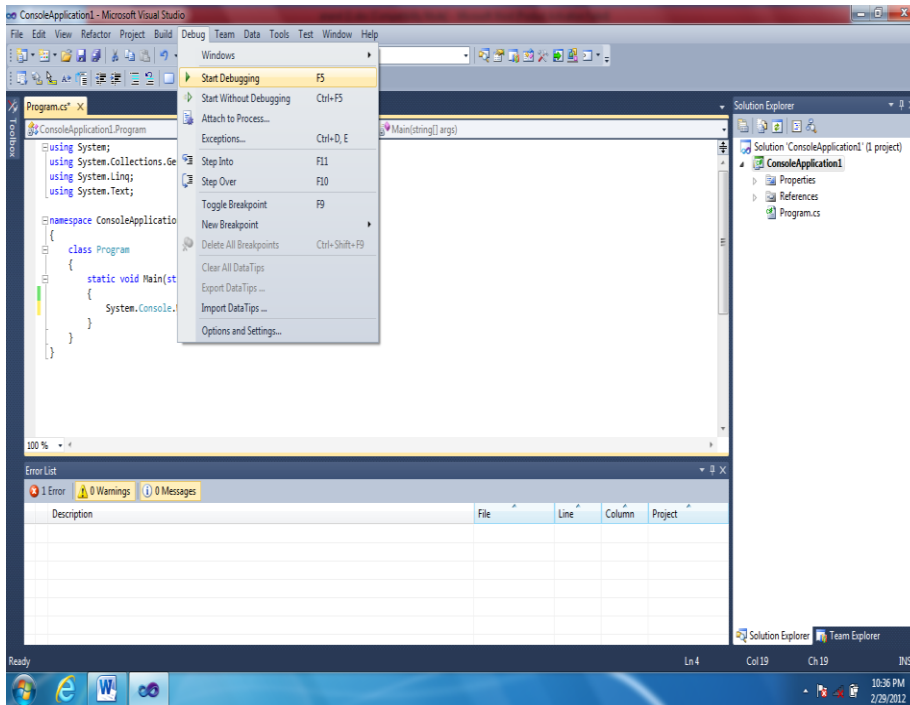
The new project appears in Solution Explorer.

1. If Program.cs is not open in the Code Editor, right-click Program.cs in Solution Explorer and then click View Code.
2. Replace the contents of Program.cs with the code.
3. Press F5 to run the project. A Command Prompt window appears that contains the line Hello World!

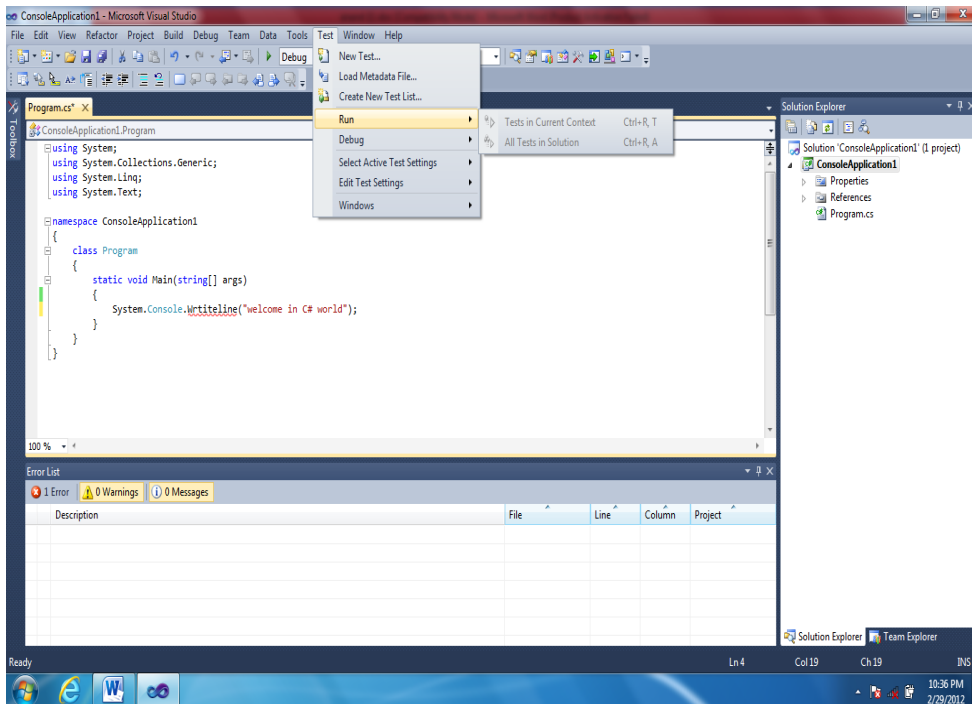
A snapshot to start Program in Visual Studio 2010



To debug:



To run a program:



First look of C#

```
// To Print Welcome in C#.
using System;
Namespace HelloWorld
{
    Class Hello
    {
        public static void Main()
        {
```

Output will be:-

```
Welcome in C# world
```

1.4.2 Basic structure of C# program

Let's enjoy your first code Of C# which displays "HelloWorld" to the console screen and analysis line by line to the program.

C# programs can consist of one or more files. Each file can contain zero or more namespaces. A namespace can contain types such as classes, structs, interfaces, enumerations, and delegates, in addition to other namespaces .C# helps the compiler to find bugs in your code during compilation time.

This code contains many things such as-

1. The **class** is a keyword(reserved word)
2. The curly brackets, {and} are the start and end parentheses, (and) is a starting brackets for Main (function).
3. String is a Class [] is an array that holds value and is a variable that means arguments.
4. The use of the **static** keyword.
5. The meaning of the **void** is blank means its return types is nothing.
6. **Main** is a function without Main () no any one program will be run. Execution begin from Main().
7. The word **System** is a namespace.
8. The period "." is an accesser.
9. **Console** is a class and **WriteLine** is a method of **Console** class.
10. The double-quotes "" will be formally known as separating message.

1.4.3 Comments in C#

The first line contains a comment. The characters `//` convert the rest of the line to a comment which can not affect the program only for programmer use & understanding.

Comments are two types-

Single line comments:

This type of comments used for single line statements only.

Exa- `// programming is very easy in C#`

`// written by Mr Kumar in year 2012`

Multiline comments:

This type of comment is used for multiline statements.

Like..

`/*`

Statements

.....

.....

.....

..... `*/`

The .net framework provides a method for display the output on screen.

`System.Console.WriteLine("Welcome in C#");`

1.4.4 Compile and execution of a program from a command prompt in C#

1. Open a Visual Studio Command Prompt window. A shortcut is available from the Start menu, under **Visual Studio Tools**.
2. Paste the code shown in the preceding procedure into any text editor and save the file as a text file. Name the file Program.cs. C# source code files use the extension .cs.
3. In the Command Prompt window, navigate to the folder that contains program.cs.
4. Enter the following command to compile Program.cs. If your program has no compilation errors, an executable file that is named Hello.exe is created.

`csc Program.cs`

5. To run the program, enter the following command:

Program

- The program contains four primary elements-
 - Namespace declaration
 - Class
 - Main method
 - and program statements.

It can be compiled with the following command line statements-

`csc.exe Welcome.cs`

This produces a file named *Welcome.exe*, which can then be executed. Other programs can be compiled similarly by substituting their file name instead of *Welcome.cs*. The file name and the class name can be totally different.

Visual C# 2010 provides an advanced code editor, easy for user interface designers, integrated debugger and so many tools to make is easy to develop applications based on C# version 4.0 and .net version 4.0 framework.

Remember that

1. C# is a case sensitive language.
2. C# class & methods are start with Capital letter.
3. Keywords are in small letter
4. C # is type safe language

1.4.5 Fundamental Types of C#

There are two basic fundamental types of C # -

1. The Simple Types

The simple types consist of Boolean and numeric types. The numeric types are further subdivided into integral and floating-Point types.

2. The Boolean Type

There's only a single Boolean type named `bool`. A `bool` can have a value of either `true` or `false`. The values `true` and `false` are also the only literal values that you can use for a `bool`.

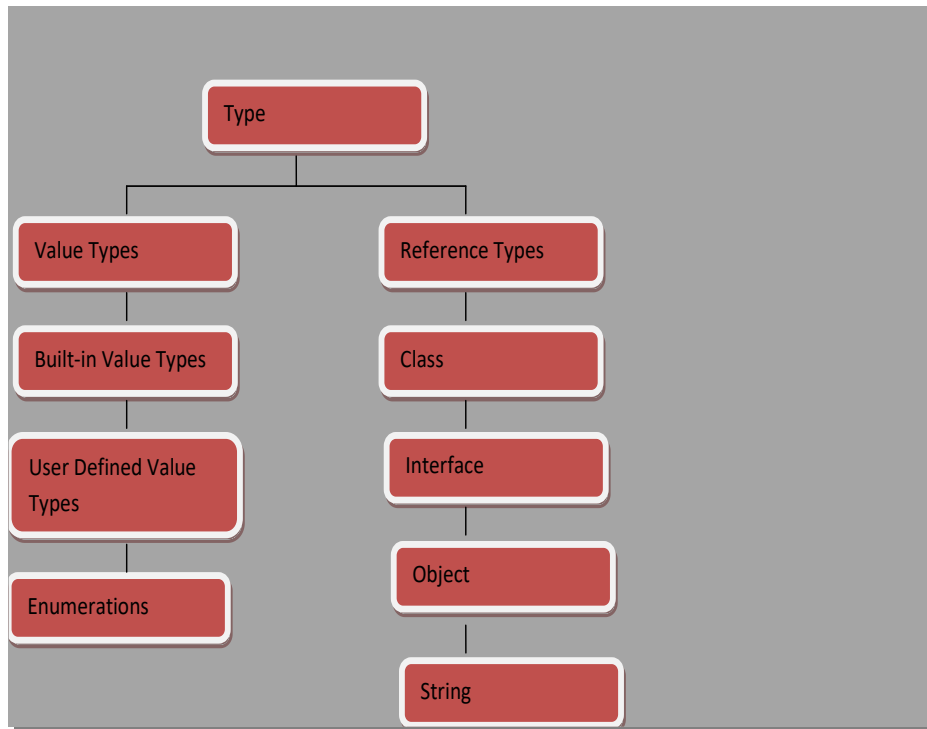
Here's an example of a `bool` declaration:

```
bool isProfitable = true;
```

Some other types are -

1. Integral Types
2. Floating-Point Types

3. Struct Types
4. Reference Types
5. Enumeration Types
6. String Type



5. Data types in C#

Data type helps us identify the type and the range of value that can be stored in a variable. C# language has a common Type System CTS also called unified type system (collection of data types). C# is strongly types language hence it is required to inform the compiler that which data types we want to use. It is required to declare every time. Here we take a look at some of the most used data types and how they work. We know that every variable or object you create in a C# program must have a specific type. C# has a common type system it tells that all types are sub class of System. Object class. The type indicates the characteristics of the object and what it can do.

A data type can be of two types –

1. Built in (intrinsic) data type- int , char , flaot
2. User defined data types – class , interface

CTS categories data types into two types –

Value Types	Reference type
-------------	----------------

Which store actual values? Exam-int, Float, char, Byte, long, Sbyte, Short, Double, Decimal, Bool	Which store references of actual data or value stored in the memory? Exam- struct , enum, String, object, class, delegate
--	--

** Also refer table in detailed.*

Boxing	Unboxing
Converting a value of a value type into a value of a corresponding reference type is known as boxing. In C# boxing is done implicit. Example- int a=10; Object b =a;(a is boxed to b)	Converting a value of a reference type (previously boxed) into a value of a value type is known as unboxing. In C# it requires an explicit type cast. Example- Example- int a=10; Object b =a;(a is boxed to b) Int c =(int) b;(unboxed to value type)

Table 1.1. The intrinsic types built into C#

C# data type	Size (bytes)	.NET type	Description
Byte	1	Byte	Unsigned (values between 0 and 255).
Char	2	Char	Unicode characters (a modern way of storing most characters, including international language characters).
Bool	1	Boolean	Can store True or false values .
Sbyte	1	SByte	Signed (values between -128 and 127).
Short	2	Int16	Signed (short) (values between -32,768 and 32,767).
Ushort	2	UInt16	Unsigned (short) (values between 0 and 65,535).
Int	4	Int32	Signed integer values between -2,147,483,648 and 2,147,483,647.
UInt	4	UInt32	Unsigned integer values between 0 and 4,294,967,295.
Float	4	Single	Floating-point number. Holds the values from approximately +/-1.5 x 10-45 to approximately +/-

			3.4 x 10 ³⁸ with seven significant figures.
Double	8	Double	Double-precision floating-point. Holds the values from approximately +/-5.0 x 10 ⁻³²⁴ to approximately +/-1.8 x 10 ³⁰⁸ with 15 to 16 significant figures.
decimal	12	Decimal	Fixed-precision up to 28 digits and the position of the decimal point. This type is typically used in financial calculations. Requires the suffix "m" or "M" when you declare a constant.
Long	8	Int64	Signed integers ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
Ulong	8	UInt64	Unsigned integers ranging from 0 to approximately 1.85 x 10 ¹⁹ .

6. Some common Difference between C# & others OOPs languages

C++	VB	Java	C#
OOPs Inheritance Polymorphism Data hiding Encapsulation Overloading reusability	RAD IDE GUI Interface Modularization Event driven programming Data base support	OOP Portability Platform independence Interpreter No pointer No preprocessor Automatic memory management Security Dynamic binding Threading Data base support Built in networking	Class & object Polymorphism Inheritance Dynamic binding Overloading Struct & enum Boxing & Unboxing Exception handling Rapid action development Window application Garbage collections Multithreading Null able types GUI Interface

*Here in the development of application two terms are very important.

IDE- Integrated development Environment

An IDE is a programming environment that has been packaged as an application program, consisting of code editor, a compiler, a debugger and a graphical user interface builder. A

programming environment, where all the tools required for programming are available under one roof is called IDE.

RAD- Rapid Application Development

A programming style which aims at building programs fast, quick application developments through the use of tools and wizards is called RAD. It was developed by James Martin in 1980 but introduced in around 1990.

7. Scope of Variables

Variable is a named storage location in computer memory whose contents can change during a program run. Variable is a named storage location in computer memory whose contents can change during a program run.

The scope of variable means the visibility of the variable rest of program's we know that variable will declare within the method and the variable will remain in the entire method and it is unavailable for another method. It can be used only in the method where it is declared.

Variable scope can be defined in other method-

- Class level -

The variable can be used by any method within the class

- Method level Scope -

the variable of the method can be used within only in the method.

- Loop level scope-

the variable are only visible within the said loop.

- Nested level scope-

The variable declared within the nested scope is not available to those who are outside of the code block.

8. Test Your Knowledge

While especially valuable for the new programmer, these exercises can also be helpful for more experienced programmers, who can go directly to their areas where they would like a little reinforcement of their understanding. The exercises can be an active aid in reviewing C# language with .NET Framework.

1. Define the following terms -data, information, statements, program, and software.
2. What is Object Oriented programming in C#?
3. Write the important features of C# language? Write the use of this language.
4. Why C# called type-safe language?
5. State five differences between C# and Java?
6. Name some language supported by .net framework.
7. Why the need of VOS (Virtual Object System) in C#?
8. Explain the following program.

```
class Example
{
```

```

        Static void Main()
        {
            System.Console.WriteLine(" welcome in C#");
            System.Console.WriteLine(" enjoy programming" );
        }
    }

```

9. Find out the error of this following program.

```

class First
{
    static void Main()
    {
        System.Console.WriteLine(Keep Smiling);
        System.Console.WriteLine(Hello !!! How are you? );
    }
}

```

10. What is the difference between WriteLine() and Write()?
11. What is used of placeholder when printing a value with WriteLine() or Write()?
12. When C # came in existence and how?
13. What is .net framework?
14. When .net was announced? What is its use in application development?
15. Write the stem to execute c# program.
16. Name the tools which you will use to develop the C # program & .net.
17. What is the CLI? Is it the same as the CLR?
18. What is IL? Give one example.
19. What is garbage collection?
20. What do you mean by variable scope?
21. Differentiate between RAD & IDE.
22. Explain Microsoft Intermediate Language.
23. What is the Common Language Runtime (CLR)?
24. Name the same namespaces provides by the .NET Framework.
25. Write the CLR execution process.
26. Define boxing & unboxing in C# with examples.

27. C# is case sensitive what does it means?
28. Difference between value type & reference type in C#.
29. Explain the architecture of .net.
30. Create a program using c# and execute on command prompt.

Operators & Expressions in C#

2.1 INTRODUCTION

As we know that computer and software are the backbone of technology. A computer cannot understand person's spoken language like English. So that, we must familiar with any language that computer can easily understand. This is when programming comes in the picture. Programming is the act or process of planning or writing a program. When a programmer writes a code actually he is using any programming language. Hundreds of languages came but only some languages were popular due to their performance.

Since the invention of the computer many programming languages approaches have been tried. These include techniques like top down, bottom up, modular, structured & OOPs.

In continuation of change in technology software industry or program developers need some recent languages. Today Client site application & Server site application, **Mobile application** development is demand of software industry. So a language was needed for development, then a language came in existence which have many features of C, C++, Java, VB etc. This language helps programmer to create secure, robust, portable, distributed object oriented applications for the real world and global internet.

Can you guess it? Yes, it is C# language which supports maximum features of Object Oriented Programming & also helps us for easy doing programming in Console and window application. Learning a programming language is much like learning any other skill. It also requires lots of practices. In this book we teach you language C# learning is same as learning of Hindi & English. Here, we use basically two important terms of application development - The.Net and C#.

C# language also focuses on the Windows based application programs, visual programming concepts, interactive graphics fundamentals, and database connectivity concepts.

This book includes topics such as Windows Forms, Windows Controls, Windows programming data access with ADO .NET, and handling data access and data manipulation in codes and also all console based program.

The book provides deep insights into the .NET programming concepts and is designed to enhance the programming skills of the users of C#. This book is a practical introduction to programming in C# utilizing the services provided by .NET to build Web-based services and others. Here we introduces C#'s advanced object-oriented capabilities

early – helping you make the most of them to create software with unprecedented efficiency and power and also covers data types, formatting and conversions, exceptions, interfaces, collections, the callback mechanism, and attributes of languages .

Finally, this book will cover all aspect of C# programming language. So enjoy programming very easily. Here our team will guide you beginning to end of C# programming and we hope you are comfortable after reading this book.

The main objective of this book is-

- To understand the basic concept of C#.
- To understand the concept of .net framework
- To understand data types
- To understand data base connectivity
- To develop complete application using C# and .net
- To obtain a basic idea of development of future generation of application and learn many more with executed codes.

2.2 OPERATOR AND OPERANDS

2.3 TYPES OF OPERATOR

2.3.1 ARITHMETIC OPERATOR

2.3.2 LOGICAL OPERATOR

2.3.3 RELATIONAL OPERATOR

2.3.4 INCREMENT & DECREMENT OPERATOR

2.3.5 ASSIGNMENT OPERATOR

2.3.6 SPECIAL OPERATOR

2.3.7 CONDITIONAL OPERATOR

2.3.8 BITWISE OPERATOR

2.4 PRECEDENCE OF OPERATOR

2.5 TYPE CONVERSION

8. Test Your Knowledge

While especially valuable for the new programmer, these exercises can also be helpful for more experienced programmers, who can go directly to their areas where they would like a little reinforcement of their understanding. The exercises can be an active aid in reviewing C# language with .NET Framework.

31. Define the following terms –data, information, statements, program, and software.
32. What is Object Oriented programing in C#?
33. Write the important features of C# language? Write the use of this language.
34. Why C# called type-safe language?
35. State five differences between C# and Java?
36. Name some language supported by .net framework.
37. Why the need of VOS (Virtual Object System) in C#?
38. Explain the following program.

class Example


```

{
    Static void Main()
    {
        System.Console.WriteLine(" welcome in C#");
        System.Console.WriteLine(" enjoy programming" );
    }
}

```

39. Find out the error of this following program.

```

class First
{
    static void Main()
    {
        System.Console.WriteLine(Keep Smiling);
        System.Console.WriteLine(Hello !!! How are you? );
    }
}

```

- 40.** What is the difference between WriteLine() and Write()?
- 41.** What is used of placeholder when printing a value with WriteLine() or Write()?
- 42.** When C # came in existence and how?
- 43.** What is .net framework?
- 44.** When .net was announced? What is its use in application development?
- 45.** Write the stem to execute c# program.
- 46.** Name the tools which you will use to develop the C # program & .net.
- 47.** What is the CLI? Is it the same as the CLR?
- 48.** What is IL? Give one example.
- 49.** What is garbage collection?
- 50.** What do you mean by variable scope?
- 51.** Differentiate between RAD & IDE.
- 52.** Explain Microsoft Intermediate Language.
- 53.** What is the Common Language Runtime (CLR)?
- 54.** Name the same namespaces provides by the .NET Framework.
- 55.** Write the CLR execution process.

56. Define boxing & unboxing in C# with examples.
57. C# is case sensitive what does it means?
58. Difference between value type & reference type in C#.
59. Explain the architecture of .net.
60. Create a program using c# and execute on command prompt.

Control Statement in C#

3.1 INTRODUCTION

In programming, the control statement tells the flow of control of data in any program and helps to take the decision depending upon the input and conditions. The program written in C# language is the flow of sequence of statements. A control statement helps to control the order of execution of program.

There are three categories of control statements in any language- sequential, selection, iterations.

3.2 STATEMENTS

Statements are the instructions given to the computer to perform any kind of task or action. It can be any data, value, decision, condition or any repeating statements.

Statements are terminated with a semicolon (;) in a programming language.

Examples of statement:

to start computer and do your program.

a=a+2;

C# is programming language.

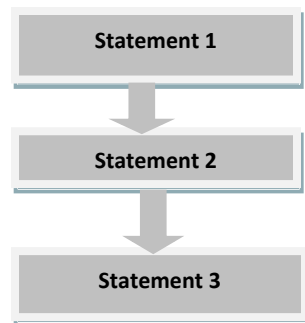
We are Indians.

Statements are three types:

1. Sequential statement
2. Selection statement
3. Iteration statement

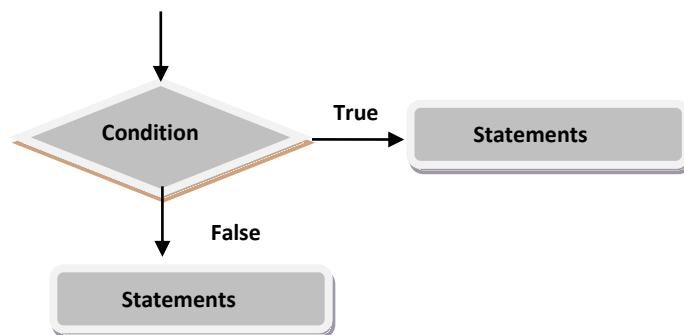
3.2.1 Sequential Statement

The sequence construct means the statement are being executed sequentially one by one or step by step. This represents the default flow of statements in a program.



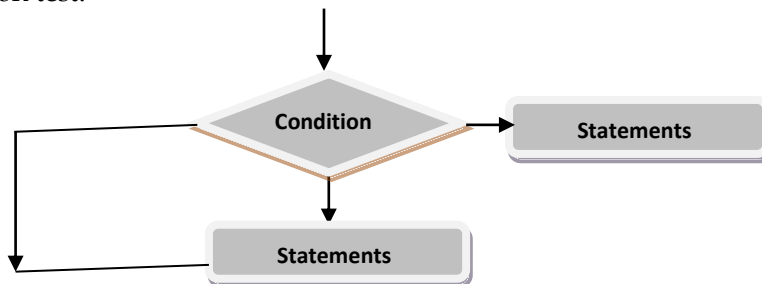
3.2.2 Selection Statement

The selection construct means the execution of statement depending upon certain condition test(true or false).



3.2.3 Iteration (Looping) Statement

The iteration constructs means repetition of a set of statements depending upon certain condition test.



3.3 TYPES OF CONTROL STATEMENTS

The three categories of program control statement in C# are:

3.3.1 Branching/Decision Making Control Statement (selection statements)

- i. If, if-else
- ii. switch-case

3.3.2 Jumping Control Statement(Transfer statements)

- i. break
- ii. return

- iii. continue
- iv. throw
- v. goto

3.3.3 Looping/Iteration Control Statement

- i. for
- ii. while
- iii. do-while
- iv. foreach

3.4 BRANCHING CONTROL STATEMENT

3.4.1 The if Statement

If statements help to execute a block of statements based on the result of a certain given condition. If statements allow for decision making depending on conditions. They are used with relational statements/operators.

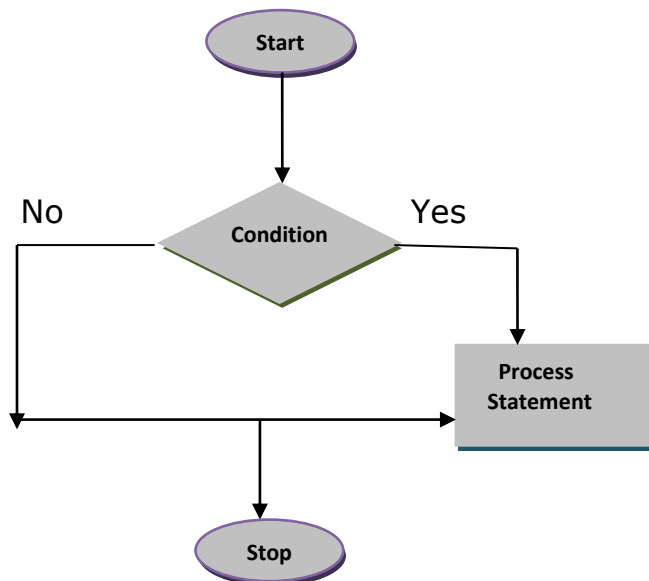
If the condition set evaluates to true the block of statement of if will be executed otherwise another block is executed. If another block is not present program execution will be terminated. We can use *if* statement, *if-else* statement, multiple *else if* statements etc.

The Syntax of if statement is shown here:

```
if (Condition)  
    Statement;
```

Example:

```
If(age >= 18)  
{  
    Console.WriteLine("you are eligible for voting");  
}
```



3.4.2 The if-else Statement

If-else statement is another part of if it also helps to execute a block of statement based on the result of a condition. It has two parts if and else. If the condition is false the statements of else block will be executed. Multiple *if-else* statements can be used in a program.

The complete syntax of if statement is shown here:

```
if (Condition)
    Statement;

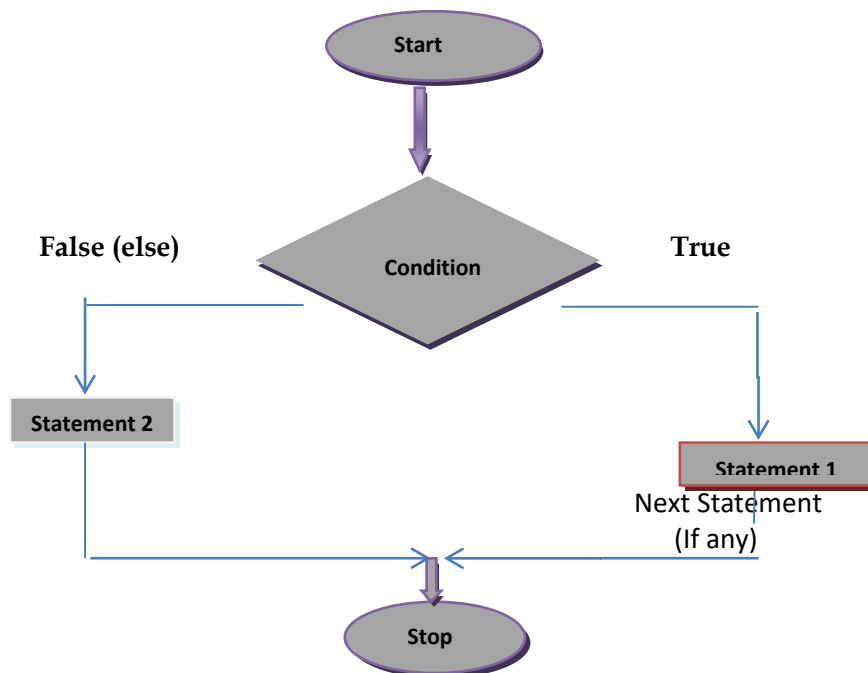
[else
    Statement;]
```

'else' part of 'if statement' is optional, if the user doesn't provide an else part and the condition evaluates to false, then nothing would happen.

Example:

```
        If(x < y)
    {
        Console.WriteLine("x is less than y");
    }
    else
    {
        Console.WriteLine("x is greater than y");
    }
}
```

Flow chart of if-else statement



The another form of if statements is if-else-if ladder.

Syntax:

```
if (condition)
{
    Statement;
}
else if(condition)
{
    Statement;
}
else if(condition)
{
    Statement;
}
.
.
.

else
{
    statement(s);
}
```

If the block of statements contains multiple statements then enclose { } brackets. If condition is true then if part statement is execute and if condition is false then else part statement are execute and the statement is bypassed to next statement.

Condition is a Boolean expression that is either true or false.

Program Example of if statement

```
using System;
```

```
class CampareExample
```

```
{
    public static void Main()
    {
        int x, y, z;
        x = 2;
        y = 3;

        if(x < y)
            Console.WriteLine("x is less than y");

        if(x == y)
```

```

        Console.WriteLine("you won't see this");

        Console.WriteLine();

        z = x - y;

        Console.WriteLine("z is",z);
        if(z >= 0)
            Console.WriteLine("z is non-negative");
        if(z < 0)
            Console.WriteLine("z is negative");

        Console.WriteLine();

        z = y - x;
        Console.WriteLine("z is",z);
        if(z >= 0)
            Console.WriteLine("z is non-negative");
        if(z < 0)
            Console.WriteLine("z is negative");
    }
}

```

Output will be

```

x is less than y
z is -1
z is negative
z is 1
z is non-negative

```

Program to check Boolean values in if statement.

```

using System;

class BooleanExample
{
    public static void Main()
    {
        bool b;
        b = false;
        Console.WriteLine("b= " + b);
        b = true;
        Console.WriteLine("b= " + b);

        // a boolean value can control the if statement
    }
}

```



```

        if(b)
            Console.WriteLine("b is true.");

        b = false;
        if(b)
            Console.WriteLine("b is false.");
    }
}

```

Output will be

```

b = false
b = true
b is true

```

Program to check a given values is even or odd using loop.

Using System;

```

class EvenOddExample
{
    public static void Main()
    {
        int i;

        for(i=1; i <= 20; i++) {
            Console.Write("Testing " + i + ": ");

            if((i%2) == 0)
                Console.WriteLine("even");
            else
                Console.WriteLine("odd");
        }
    }
}

```

Output will be

```

Testing 1 : odd
Testing 2 : even
Testing 3 : odd
Testing 4 : even
Testing 5 : odd
Testing 6 : even
Testing 7 : odd
Testing 8 : even
Testing 9 : odd

```

if statement with || and && operators.

```
using System;

class AndOrExample
{
    static void Main(string[] args)
    {
        int x = 5, y = 5, z = 10;

        if (x == y)
            Console.WriteLine(x);

        if ((x > z) || (x == y))
            Console.WriteLine(y);

        if ((x >= z) && (y <= z))
            Console.WriteLine(z);
    }
}
```

Output will be:-

5
5

1.4.3 Nested if statement

A statement within a statement is known as nesting of statement. When a conditional statement or a looping statement can be included inside of another, this situation is called nesting of loop or condition. This technique is used to create a condition, where any condition directly depends on another condition.

Any statement can be nested inside of another any times.

```

class NestedExample
{
    public static void Main()
    {
        int x,y,z;
        x=9;y=33;z=22;

        if (x>y)
            if(x>z)
                System.Console.WriteLine("x is the greatest no");
            else
                System.Console.WriteLine("z is the greatest no");
        else
            if(y>z)
                System.Console.WriteLine("y is the greatest no");
            else
                System.Console.WriteLine("z is the greatest no");
    }
}

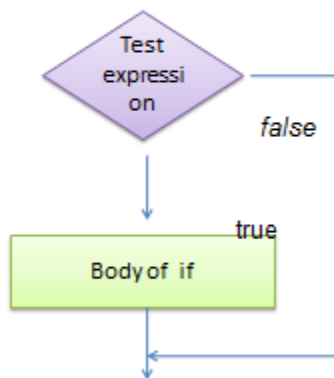
```

Output will be:-

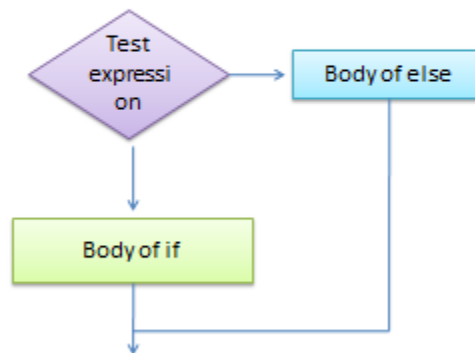
Y is the greatest no

if vs if else

if statement



if else statement



Various forms of if are-

If(age>18)

If(age)

If(age==20)

If(age<=21)

If(age>21)&& If(Sex='M')

If(a%10==2)

1.4.4 The *switch* Statement

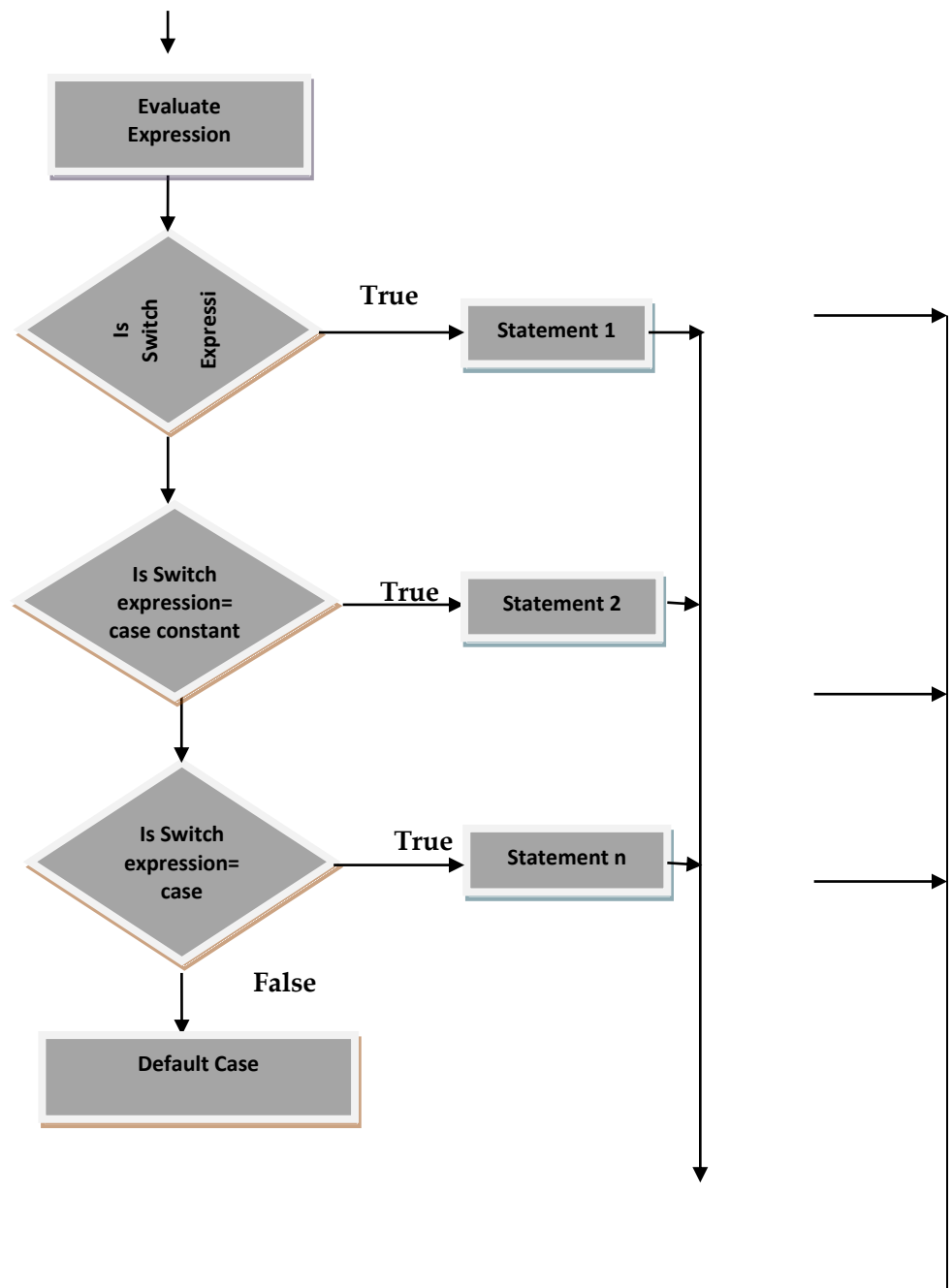
The *switch* statement (switch case statements) is the another form of selection statement, which executes a statement from a group of statements based on the result of an expressions. It provides multiple choices. It begins with a keyword Switch and further contains various case labels. When the switch statements is executed it compare the value of the controlling condition or expression to the value of each case labels. The types of the values in a switch statement can be char, constant or a value operates on Booleans, enumerations, integral types or strings.

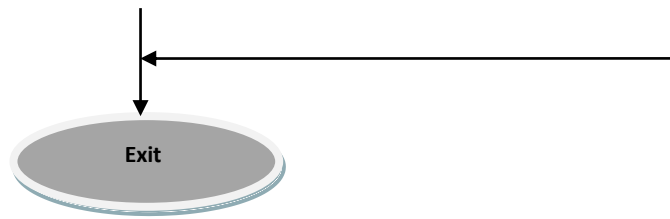
Syntax

```
switch (Expression)
{
    case value 1:
        statement(s);
        break;
    case value 2:
        statement(s);
        break;
    case value 3:
        statement(s);
        break;
    ----
    ----
    ----
    [default:
        statement(s);]
}
```

Note:

The default statement is executed when none of the above mention case matches with the result of the switch expression. Default is optional. Break is also optional and used to end the series of statements.





For Example

```
switch (day)
{
    case 1:
        Console.WriteLine("Monday");
        break;
    case 2:
        Console.WriteLine("Tuesday");
        break;
    case 3:
        Console.WriteLine("Wednesday");
        break;
    case 4:
        Console.WriteLine("Thursday");
        break;
    case 5:
        Console.WriteLine("Friday");
        break;
    case 6:
        Console.WriteLine("Saturday");
        break;
    case 7:
        Console.WriteLine("Sunday");
        break;
    default:
        Console.WriteLine("Wrong day entered");
        break;
}
```

The *switch* block follows the *switch* expression, where one or more choices are evaluated for a possible match with the *switch* expression. Each choice is labeled with the *case* keyword, followed by an example that is of the same type as the *switch* expression and followed by a colon (:).

In the example we have *case 1*; *case 2*; *case 3*; ... case 7:. When the result evaluated in the *switch* expression matches one of these choices, the statements immediately following the matching choice are executed, up to and including a branching statement, which could be either a *break*, *continue*, *goto* , *return*, or *throw* statement. If the switch statement do not matches with any choice then the default case will be executed.

Program of switch case

```
using System;
class switchexp
{
    public static void Main(string[] args)
    {
        Console.WriteLine("please enter a day number of the week");

        string choice=Console.ReadLine();

        switch(choice)
        {
            case "1":

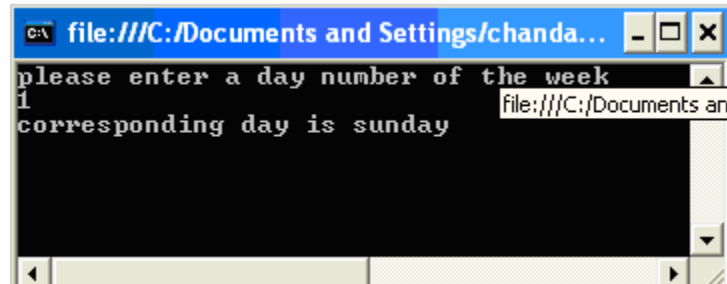
                Console.WriteLine("corresponding day is sunday");
                break;
            case "2":
                Console.WriteLine("corresponding day is monday");
                break;
            case "3":
                Console.WriteLine("corresponding day is tuesday");
                break;
            case "4":
                Console.WriteLine("corresponding day is wednesday");
                break;
            case "5":
                Console.WriteLine("corresponding day is thursday");
                break;
            case "6":
                Console.WriteLine("corresponding day is friday");
                break;
            case "7":
                Console.WriteLine("corresponding day is saturday");
                break;
            default:
                Console.WriteLine("please enter number between 1 to 7");
                break;

        }

    }
}
```

```
}
```

Output



We may also include a *default* choice following all other choices. If none of the other choices match, then the *default* choice is taken and its statements are executed. Use of the *default* label is optional.

Each *case* label must end with a branching statement, which is normally the *break* statement. The *break* statement will cause the program to leave the *switch* statement and begin execution with the next statement after the *switch* block.

We can also combine case statements this is called adjacent *case* statements with no code in between or using a *goto* statement. Following example that shows how to combine case statements:

```
switch (month)
{
    case 1:
    case 2:
    case 3: Console.WriteLine(" 1st Quarter of Year ");
        break;
    case 4:
    case 5:
    case 6: Console.WriteLine(" 2nd Quarter of Year ");
        break;
    case 7:
    case 8:
    case 9: Console.WriteLine(" 3rd Quarter of Year ");
        break;
    case 10:
    case 11:
    case 12: Console.WriteLine(" 4th Quarter of Year ");
        break;
    default:
        Console.WriteLine("Wrong year entered");
        break;
}
```

By placing *case* statements together, with no code in-between, we create a single case for multiple values. A case without any code will automatically fall through to the next case.

The example above shows how the three cases for *month* equal to 1, 2, or 3, where *case 1* and *case 2* will fall through and execute code for *case 3*.

1.5 JUMPING STATEMENTS

A jump statement can be used to transfer program control using keywords such as `break`, `continue`, `return`, and `throw`.

C# Branching Statements summary is below-

Branching statements	Description
<code>break</code>	It terminates a statement sequence .It also used to exit the switch block or loop.
<code>continue</code>	exit the switch block, skips remaining logic in enclosing loop, and goes back to loop condition to determine if loop should be executed again from the beginning.
<code>goto</code>	exit the switch block and jumps directly to a label of the form " <code><labelname>:</code> "
<code>return</code>	It helps to come out of the loop and exit the current method.
<code>throw</code>	Throws an exception

Syntax of jump statement is as follows

```
switch (expression)
{
    case expression:
        //here your code statement
        jump-statement
    default:
        //your code here
        jump-statement
}
```

1.5.1 Break Statements

A `break` statement is used to exit from a case of a switch statement and it is also used to exit from `for`, `foreach`, `while`, `do.....while` loops which will switch the control to the statement immediately after the end of the loop.

For Example

```
using System;

class BreakExample
{
    static void Main(string[] args)
```

```

{
    for(int i = 1; i <= 15; i++)
    {
        if (i >= 6)
            break; //if condition is met exit the loop

        Console.WriteLine(i);
    }
    Console.Read();
}
}

```

Output

```

1
2
3
4
5

```

When the if condition is evaluated to true the break statement is executed. The for loop will exited immediately and it does not execute any other code in the loop. The break statement also works for while, do/while, and switch statements.

1.5.2 Continue

The continue keyword is used to transfer program control just before the end of a loop. The condition for the loop is then checked, and if it is met, the loop performs iteration. It skips the proceeding code in the loop and continues to the next iteration in the loop. It does not exit the loop like the break will. Just like the break to work properly. The continue statement needs an if condition to tell the program that if a certain condition is true the continue statement must be executed.

For Example

using System;

```

class ContinueExample
{
    static void Main(string[] args)
    {
        for(int i = 1; i <= 10; i++)
        {
            if (i == 5)
                continue; //condition is met skip the code below

            Console.WriteLine(i);
        }
        Console.Read();
    }
}

```

Output will be

1
2
3
4
6
7
8
9
10

The output skipped the number 5. When the condition is true the continue statement is executed and the remaining code is skipped.

1.5.3 Return

The return keyword identifies the return value for the function or method (if any), and transfers control to the end of the function.

1.5.4 Throw

The throw keyword throws an exception or runtime error. If it is located within a try block, it will transfer control to a catch block that matches the exception - otherwise, it will check if any calling functions are contained within the matching catch block and transfer execution there. If no any functions contain a catch block, the program may terminate because of an unhandled type of exception occurred.

In C#, it is possible to throw an exception also. The 'throw' keyword is used for this purpose.

Syntax of throw statement is as follows

```
throw exception_object;
```

Example

```
using System;
```

```
public class ThrowExample
{
    public static void display(Int32 balance)
    {
        if (balance < 500)
        {
            // throw an argument out of range exception if the balance is less than 500.
            throw new ArgumentOutOfRangeException("Balance can't be less than 500 ");
        }
    }

    public static void Main()
```

```

{
    try
    {
        display(200);
    }

    catch (Exception e)
    {
        Console.WriteLine(e.Message);
        Console.ReadLine();
    }
}

```

In this example we have a function called `display`, which takes `balance` as an argument. In this function we check if the `balance` is less than 500, and if so, throw `ArgumentOutOfRangeException`.

1.5.5 Use of `goto` statements in `switch` statement

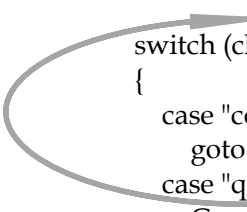
Another way to control the flow of logic in a *switch* statement is by using the *goto* statement. You can either jump to another case statement, or jump out of the switch statement.

switch with string type

```

begin:
    switch (choice)
    {
        case "continue":
            goto begin;
        case "quit":
            Console.WriteLine("Good Bye.");
            break;
        default:
            Console.WriteLine("Your input {0} is incorrect.", choice);
            goto decide;
    }

```



In the current example, `"continue"`, is a case of the `switch` statement - not the keyword.

The *goto* statement causes program execution to jump to the label following the *goto* keyword. During execution, if the user types in "continue", the switch statement matches this input (a *string* type) with the *case "continue":* label and executes the "*goto begin:*" instruction.

Summary

- (i) *goto* : A *goto* statement can transfer the program control anywhere in the program.
- (ii) *Break*: A *break* statement enables a program to terminate of the loop/block, skipping any code in between.
- (iii) *Continue*: A *break* statement enables a program to force the next iteration to take place, skipping any code in between.
- (iv) *Return*: A *return* statement is used to return from a function.

1.6 LOOPING/ITERATION CONTROL STATEMENTS

When we want to repeat certain things many times the we need loop in a program. It is the base of program & software. A looping statement creates a loop of code to execute a variable and statements number of times.

"Repetition of statements again & again till the condition is false is called looping".
There are basically three types of looping

1. *for*
2. *while*
3. *do-while*

but in C# Programming one another type of loop i.e. *foreach*

There are three necessary conditions for looping-

Initialization *i=1;*

Condition *I>2;*

Increment or decrement *i++;*

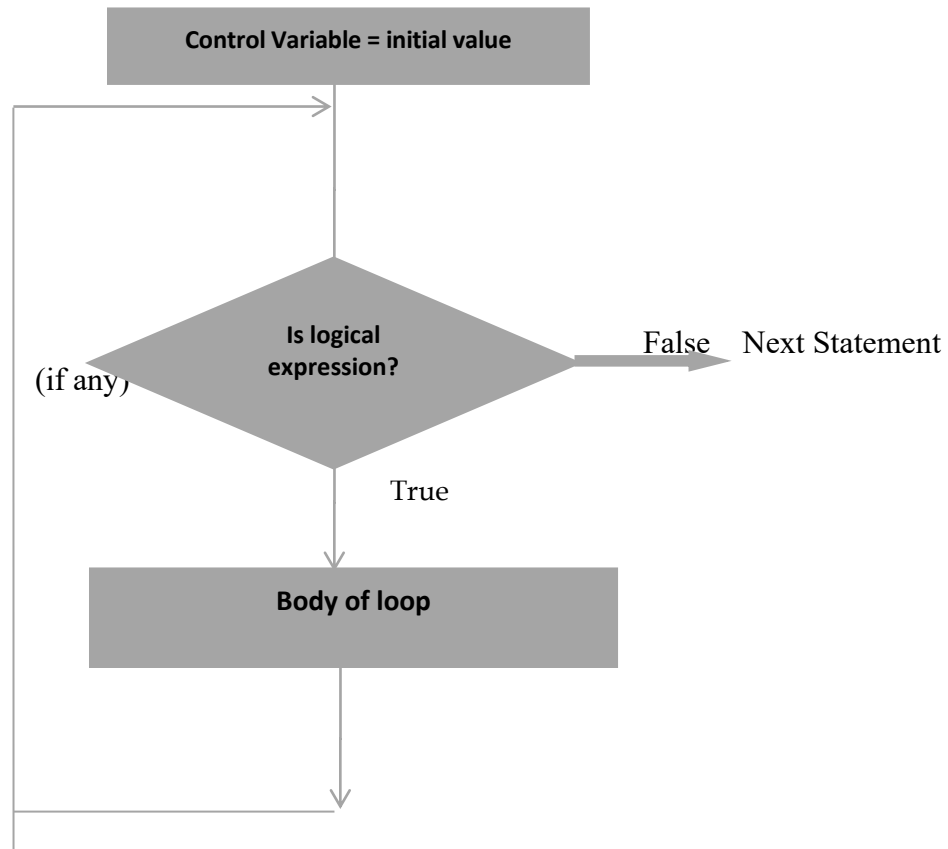
1.6.1 for loop

For loop can be used for repeatedly executing a sequence of code when the condition is known. The *for* loop has the same syntax as in other languages derived from Core C++. Writing of *for* loop is very short and easy. It is most popular used loop in programming. It is written in the following form:

```
for (initialization; condition; increment/decrement) {  
    Body of the loop ;}
```

Re-initialize the value of control variable





For example:-

Using System;

```
class ForExample
{
    public static void Main()
    {
        int i;
        for(i = 0; i <= 4; i++)
            Console.WriteLine("This is count: " + i);

        Console.WriteLine("Done!");
    }
}
```

Output will be

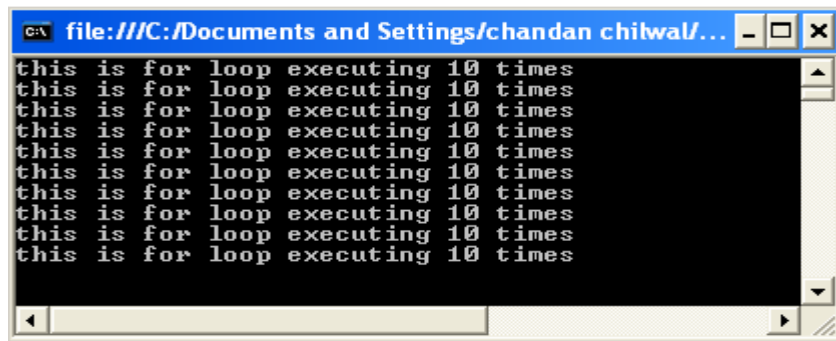
This is count: 0
This is count: 1
This is count: 2
This is count: 3

This is count: 4
Done!

Program of for loop

```
using System;
class forexp
{
    static void Main(string[] args)
    {
        for(int i=0;i<10;i++)
        {
            Console.WriteLine("this is for loop executing 10 times");
        }
    }
}
```

Output



1.6.2 While Loop

A while statement will check a condition and then continues to execute a block of code as long as the condition evaluates to a Boolean value of *true*. It is an entry condition loop, if the test condition is false to begin with the program never executes the body of the loop.

Syntax:

Its syntax is as follows

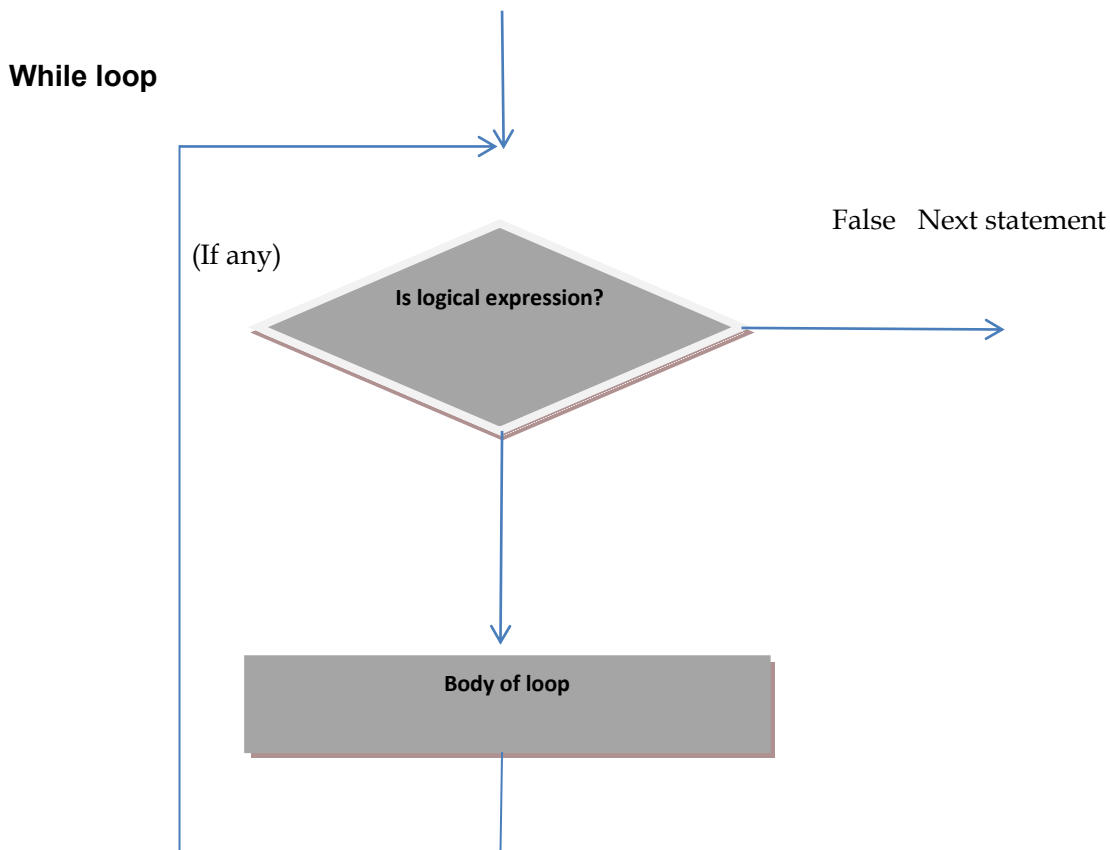
```
while (test expression )
{
    Statements;
}
```

The statements can be any valid C# statements. The condition is evaluated before any code in the following block has executed. When the condition evaluates to *true*, the statements will be executed. Once the statements have been executed then the control returns to the beginning of the *while* loop to check the boolean expression again.

When the condition evaluates to *false*, the while loop statements are jumped and execution begins after the closing braces of that block of code. Before entering the loop, we must be check that variables evaluated in the loop condition are set to an initial state.

```
(a) while (i<= 10)
{
    fact= fact * i;
    i++;
}
```

```
(b) while (i<= n)
{
    Console.Write("{0} ", i*i);
    i++;
}
```



For Example

```
using System;
```

```
class WhileExample
{
    public static void Main()
    {
        int x = 0;

        while (x < 10)
        {
            Console.Write("{0} ", x);
            x++;
        }
        Console.WriteLine();
    }
}
```

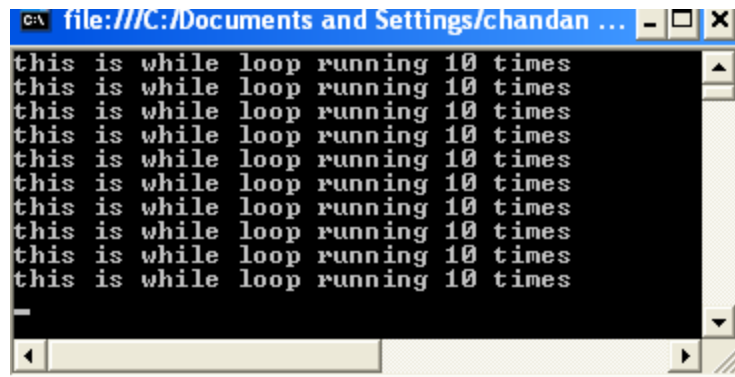
Program of while loop

```
using System;
```

```
class Class1
{
    static void Main(string[] args)
    {
        int i=0;

        while(i<10)
        {
            Console.WriteLine("this is while loop running 10 times");
            i++;
        }
    }
}
```

Output



```
C:\ file:///C:/Documents and Settings/chandan ...
this is while loop running 10 times
this is while loop running 10 times
this is while loop running 10 times
this is while loop running 10 times
this is while loop running 10 times
this is while loop running 10 times
this is while loop running 10 times
this is while loop running 10 times
this is while loop running 10 times
this is while loop running 10 times
```

1.6.3 do.....while loop

The do-while loop will always execute at least once. It is an exit condition loop; the body of the loop is always executed first then the test condition hence it execute at least once.

The general form of the do-while loop is-

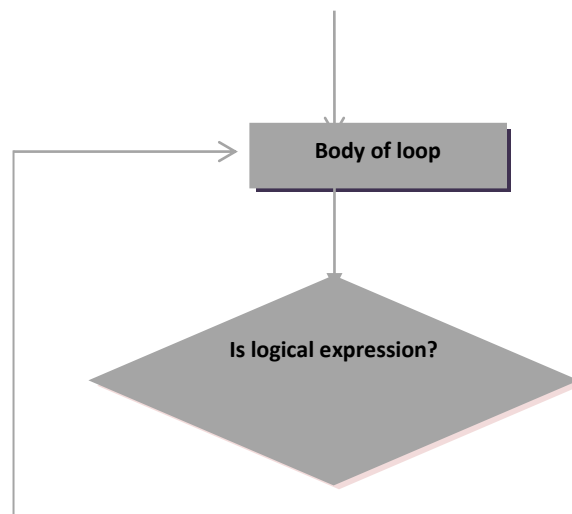
```
do
{
    Statements;
}
while (test expression);
```

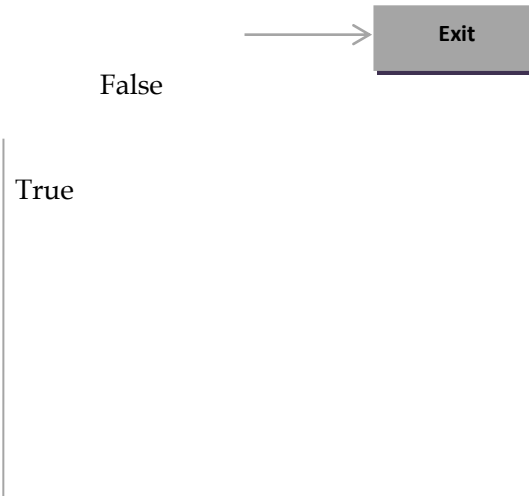
```
do
{
    statement(s);
} while(condition);
```

The do...while loop always runs its body once. After its first run, it evaluates its condition to determine whether to run its body again. If the condition is true, the body executes. If the condition evaluates to true again after the body has ran, the body executes again.

When the condition evaluates to false, the do...while loop end.

Do-While





```
using System;
class DoWhileExample
{
    public static void Main()
    {
        int x = 0;
        do
        {
            Console.WriteLine("Number is {0}", x);
            x++;
        } while (x < 10);
    }
}
```

Output will be

Number is 0
Number is 1
Number is 2
Number is 3
Number is 4
Number is 5
Number is 6
Number is 7
Number is 8
Number is 9

Program of do while loop

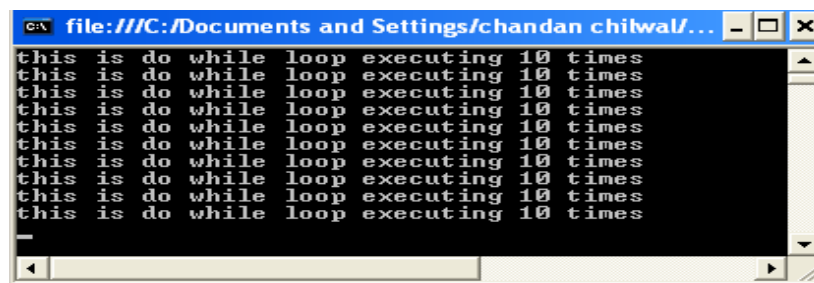
```
using System;
class Class1
```

```

{
    static void Main(string[] args)
    {
        int i=0;
        do
        {
            Console.WriteLine("this is do while loop executing 10 times");
            i++;
        }while(i<10);
    }
}

```

Output



```

C:\> file:///C:/Documents and Settings/chandan chitwal/...
this is do while loop executing 10 times
this is do while loop executing 10 times
this is do while loop executing 10 times
this is do while loop executing 10 times
this is do while loop executing 10 times
this is do while loop executing 10 times
this is do while loop executing 10 times
this is do while loop executing 10 times
this is do while loop executing 10 times
this is do while loop executing 10 times

```

1.6.4 foreach loop

The foreach statement is similar to the for statement as both allow code to iterate over the items of collections, but the foreach statement lacks an iteration index, so it works even with collections that lack indices altogether. It is written in the following form:

```

foreach("variable-declaration" in "enumerable-expression")
{
    body/ statement-or-statement-block;
}

```

The enumerable-expression is an expression of a type that implements Enumerable, so it can be an array or a collection. The variable-declaration declares a variable that will be set to the successive elements of the enumerable-expression for each pass through the body.

The foreach loop exits when there are no more elements of the enumerable-expression to assign to the variable of the variable-declaration.

Example-1

```

public class ForEachExample1

```

```

{    public void ForEachDay()

    {

        foreach (string day in "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday", "Sunday")

            System.Console.WriteLine(day);

    }

}

```

In the above code, the foreach statement iterates over the elements to write "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" to the console.

Example-2

```

public class ForEachSample2
{
    public void ForEachItem()
    {
        string[] n = {"One", "Two", "Three"};
        foreach (string item in n)
            System.Console.WriteLine(item);
    }
}

```

In the above code, the foreach statement iterates over the elements of the string array to write "One", "Two", and "Three" to the console.

Program of foreach loop

```

using System;

public class Class1

    {

        public static void Main(string[] args)

        {

            foreach(string str in args)

            {

                Console.WriteLine(str);

            }

        }

    }

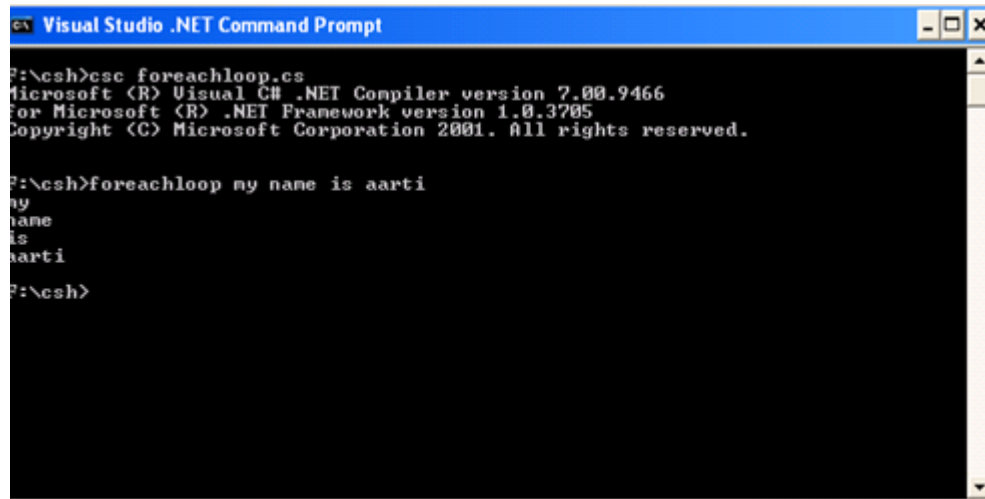
```

```

    }
}
}

```

Output



```

Visual Studio .NET Command Prompt

P:\csh>csc foreachloop.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

P:\csh>foreachloop my name is aarti
my
name
is
aarti
P:\csh>

```

1.7 Summary

In this session we learn about statements, looping like for, while, do-while other jumping control statements break, continue, throw ,goto etc. For controlling the flow of a program, the C# programming language has four loop constructs, a flexible if-else statement, a switch statement and branching statements. In C# the program introduces the Main entry point method, where control flow begins. In all cases, the string "Control flow reaches finally" is printed to the console window.

The finally block in the C# language provides a way to ensure a piece of logic is executed before the method is exited completely. A switch can be implemented as a series of if -else statements. A loop directs a program to perform a set of operations again and again until a condition is to be met.

In C# language there are several words are used to alter the flow of the program. When the program is run, the statements are executed from the top of the source file to the bottom of the file. The flow can be altered by specific keywords. Statements can be executed multiple times. Some statements are called conditional statements. They are executed only if a specific condition is found.an additional loop is added in C# foreach.

1.8 Exercise:

Part A:

1. What is statement? Explain about if statement. What is nesting of if-else?
2. What are three types of iterations? Explain each with examples. What is primary reason to use iterations?
3. What is difference between while & do -while loop.
4. What are three necessary conditions for looping?
5. What is the significance of test condition in a loop?
6. What is selection statement?
7. Name the jump statement provided by C#.
8. Compare if & which statements.
9. How does loop execution occurs?
10. What is foreach in C#? What are its uses?

Part B:

1. What command is used to jump to the next iteration of a loop?
2. WAP in C# to calculate the factorial upto n numbers.
3. What statement is used to the end a case expression in a select statement?
4. Write a program to find out the number between 1 to 100 who's divisible by 5 and 7?
5. Write a while loop program that counts number from 100 to 1?
6. Explain the need of type casting with suitable example?
7. What is ternary operator?
8. Write a program to hold the center of a circle and its radius?
9. Write a program to swap 2 numbers without using third variable?
10. Write a program if a three digit number is input through the keyboard, the number is reversed? (If number is 123 then output is 321).
11. Write a program to print all prime numbers from 1 to 100.
12. Write a menu driven program to perform the following action:
Even or Odd Number
Greater number between 3 numbers
Factorial of a number
Find out the year is leap or not
Exit When a menu item is selected then the appropriate action should be perform.
13. WAP to convert Fahrenheit into Celsius and vice versa using switch case.

14. What is the output of the following program

```
static void Main()
{
    int i;
    for(i=0;i<=50;i++);
    System.Console.WriteLine(i)
}
```

15. Write a program to take any five digit number and print the multiplication of all even digits and sum of all odd digits.

16. Write a program to find the greatest among the three numbers.

17. Write a program to find the table of a given number.

18. Write a program to print the pyramid and also print reverse of it.

```

*           1
**          11
***         111
****        1111
*****       11111
```

19. Write a program to print the pyramid and also print reverse of it.

```

1
2 3
4 5 6
7 8 9 10
```

20. Write a program to perform area of circle, area of parameters, and area of square using switch case.

21. Write a program to find the sum & difference of two numbers using switch case.

22. convert the following into do -while loop

```
int a;
for (a=1;a<=20;a++)
```

23. Give the output of following program-

```
for( int i=1,j=i;i<4;i++, j=i*3)
{
    System.Console.WriteLine("{0} {1}",i,j);
}
```


24. The following is a segment of a program-

```
int x=1, y=1;
```

```
If (n>0)
```

```
{
```

```
X=x+1;
```

```
Y=y+1;
```

```
}
```

What will be the value of x & y, if n assumes a value i)1ii)0.

25. WAP in C# to display the name of the day depending upon the number entered from the keyboard.

26. Write a program in C# to display alphabet from a-z using while loop.

27. WAP to find Armstrong no till n numbers. (eg- 153 - the sum of cube of these three is equal to the number.)

28. Write a Program in C# to find the GCD of two numbers.

29. WAP in C# to print first five odd numbers in a given numbers.

30. WAP in C# to Print following pyramids.

```
*
```

```
**
```

```
***
```

```
****
```

```
*****
```

```
1234*
```

```
123**
```

```
12***
```

```
1****
```

Structure and Array

4.1. INTRODUCTION

In any programming language, the data type is important. Structures are called mixed data types or collection of primary data types. An array is a collection of consecutive memory locations of same data type.

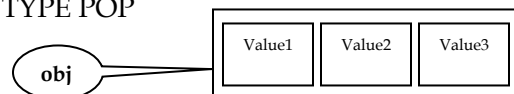
4.2. STRUCTURES

Structure is value type or custom data type. Also called user defined data types. Structures are declared using a keyword 'struct' with a valid name of structure. As-

```
struct pop
{
    public int value1;
    public int value2;
    public int value3;
}
```

How memory is allocated by the user defined data type 'pop'? It allocates the memory at the time of declaring its object (the variable of type pop). As-

```
public pop obj; // OBJECT IS DECLARED OF TYPE POP
```



4.3. NESTED STRUCTURES

In C#, Structures can also be nested. Nested Structure means a Structure inside the other Structure body. The nested structure is called the member structure of the parent structure. The accessibility of the nested structure depends upon the access modifier used to declare it inside the structure body.

There are two ways of nesting a C# **structure**. The first way is nesting the whole declaration of a structure inside the other structure and the second way is to use the structure as a member variable of another structure. As- First way of nesting is.

```
struct book
{
    public string book_name;
    public string book_author;

    public struct book_edition
```

```

    {
        public string book_edition_name;
    }
}
Second way of nesting is.
struct book
{
    public string book_name;
    public string book_author;
}
public struct book_edition
{
    public string book_edition_name;
    public book objb1;    // OBJECT IS DECLARED OF TYPE BOOK
}

```

4.4. DIFFERENCE BETWEEN CLASSES & STRUCTURES

Difference Between Structures and Classes		
S. No.	Structure	Classes
1.	Structures are non inheritable.	Classes support Inheritance.
2.	Structures are Values types.	Classes are Reference types
3.	Structures member variable initialization is not possible.	At the time of class declaration member variable initialization is possible.
4	All members are Public by default.	Class variables and constants are Private by default.

4.5. ENUMERATIONS

An enumeration is a special kind of value type. Enumerations, like classes and structures, also allow us to define new types. Enumeration types contain a list of named constants. The following code shows an example of such above definition:

```
public enum DAYS { Mon, Tues, Wed, Thu, Fri, Sat, Sun};
```

In the above example we declared an enumerator named 'DAYS' which is having the members (like- Mon, Tues, Wed, Thu, etc.) with their constant values. By default the values starts from 0 (zero) and if no values are assigned the next member value is more than one by its previous. Thus, in the above example 'Mon' is a fixed value 0, 'Tues' is 1, 'Wed' is 2, and so on.

4.6. ENUM TYPE CONVERSION

4.7. ARRAYS

A Simple variable of any data type can store only one value at a time where as an array of any data type can store more than one values in different – different consecutive locations of same data type. These different values are called elements of an array. In other words an array is a collection of similar type of data in various subscripts those are in sequence. Array is called reference types. By default array first index OR subscript starts by 0 (zero).

4.8. CREATING AN ARRAY

At the time of creating array we should specify its length. For Example-1

```
public class ArrPrac
{
    static void Main()
    {
        int[] arr = new int[6]; // DECLARATION OF AN ARRAY WITH
                                TOTAL // SIZE (0 TO 5) 6
    }
}
```

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]

4.9. TYPES OF ARRAY

4.9.1. 1-D ARRAY

In the above examples we worked with one OR single dimensional arrays. The number of indexes needs to specify an element of an array is called dimensional.

Initialization of Array – There are several ways to initialize an array.

(i) For Example- 1

```
public class ArrPrac
{
    static void Main()
    {
        int[] arr = new int[6];
        arr[0]=23; // INITIALIZATION OF ARRAY NAMED arr
        arr[1]=10;
        arr[2]=34;
        arr[3]=12;
        arr[4]=16;
        arr[5]=62;
        for (int i = 0; i < arr.Length; i++)
        {
            Console.WriteLine(arr[i]);
        }
    }
}
```

23	10	34	12	16	62
arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]

(ii) For Example- 2

```
public class ArrPrac
{
    static void Main()
    {
        int[] arr = new int[] {20, 14, 15, 26, 73, 3, 21 };
    }
}
```

```

// ARRAY DECLARATION AND INITIALIZATION AT SAME TIME

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

```

In the above initialization at the time of array declaration values are initialized are initialized to the array named 'arr'.

4.9.2. 2-D ARRAY

Two dimensional arrays sometimes called multi-dimensional arrays, also known as a rectangular array is an array with more than one dimension. It is can be of fixed sized or dynamic sized. As-

Creating Two D Arrays. **Example- 3**

```
int [, ] arr_twoD;
```

Initializing Two Dimensional Arrays- Example- 4

```
int[,] arr_twoD = new int[3, 2] { { 10, 21 }, { 34, 41 }, { 15, 60 } };
```

Example- 5

```
int[, ] values = new int[3, 2];
values[0, 0]=10;
values[0, 1]=13;
values[1, 0]=11;
values[1, 1]=40;
values[2, 0]=17;
values[2, 1]=70;
```

Accessing 2D Arrays. As-

```
Console.WriteLine(numbers[0,0]);
```

	Column [0] [1]	
Row [0]	10	21
Row [1]	34	41
Row [2]	15	60

4.10. DYNAMIC ARRAY

A dynamic array means we can change its size at the time of program execution. It does not have a predefined size. 'ArrayList' is a collection from a standard 'System.Collections' namespace. It is a dynamic array. An ArrayList automatically expands as data is added. An ArrayList can hold data of mixed or multiple types. Elements in the ArrayList are accessed via an integer index. As- Example- 3

```

using System;
using System.Collections;
public class DynArray
{
    static void Main()
    {
        ArrayList dyn_arr_list = new ArrayList();
        dyn_arr_list.Add("C Sharp Book");
        dyn_arr_list.Add(987);
    }
}

```

```

        dyn_arr_list.Add(123);
        dyn_arr_list.Add("abcd");
        dyn_arr_list.Remove(123);

        foreach(object obj_store in dyn_arr_list)
        {
            Console.WriteLine(obj_store);
        }
    }
}

```

In the above example, we created an `ArrayList` collection. We added few elements to '`dyn_arr_list`'. It is of various data type, string, int and a class object.

```
dyn_arr_list.Add(987);
```

It adds a dynamic value at run time using `Add()` method. Similarly when we used a method `Remove()` it removes a specified value.

INTRODUCTION OF STRUCTURE AND ARRAY:

➤ **STRUCTURE:**

Structure are similar to classes in C#. Although classes will be used to implements most objects, it is desirable to use structs where simple composite data types are required. Because they are value types stored on the stack, they have the following advantages compare to class objects stored on the heap:

They are created much more quickly than heap-allocated types.

They are automatically deallocated once they go out of scopes.

It is easy to copy value type variables on the stack.

Example:

```
using System;
```

```
struct way
```

```
{
```

```
public string direction;
```

```
public double distance;
```

```
}
```

```
class Class1
```

```
{
```

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

```
{
```

```
    way w;
```

```
    w.direction="north";
```

```
    w.distance=2.5;
```

```
    Console.WriteLine("the direction is="+w.direction);
```

```
    Console.WriteLine("the distance is="+ w.distance+"km");
```

```
}
```

```
}  
file:///C:/Documents and Settings/chand...  
the directon is=north  
the distance is=2.5km  
Minimize
```

➤ DEFINING A STRUCT:

A struct in C# provides a unique way of packing together data of different types. Structs are declared using the `structs` keyword. The simple form of struct definition is as follows:

Struct struct-name

```
{  
Data member1:  
Data member2:  
.....  
.....  
}
```

Example:

Struct student

```
{  
public string name;  
public int RollNumber;  
public double TotalMarkes;  
}
```

Here keyword `struct` and `Student` as a new data type that can hold the three variable of different data types.

STRUCTURES WITHIN STRUCTURES

Structure with in a structure means nesting of structures. Let us consider the following structure defined to store information about the salary of employees.

```
Str uct salary {  
char name[20];  
char department[10];  
int basic_pay;  
int dearness_allowance;  
int city_allowance;  
}  
employee;
```

This structure defines name, department, basic pay and 3 kinds of allowance. we can group all the items related to allowance together and declare them under a substructure are shown below:

```
struct salary
{
char name [20];
char department[10];
struct
{
int dearness;
int hous_rent;
int city;
}
allowance;
}
employee;
```

The salary structure contains a member named allowance which itself is a structure with 3 members. The members contained in the inner, structure namely dearness, hous_rent, and city can be referred to as :

```
employee allowance. dearness
employee. allowance. hous_rent
employee. allowance. city
```

An inner-most member in a nested structure can be accessed by chaining all the concerned. Structure variables (from outer-most to inner-most) with the member using dot operator. The following being invalid.

Structures are passed to functions by way of their pointers. Thus, the changes made to the structure members inside the function will be reflected even outside the function.

INTRODUCTION OF ARRAY:

In Computer memory every bite is an array element. Abstraction translates these bytes into object and gives them meaning. Arrays are a foundational data type. They are the basis of more usable collections.

“An Array is a fixed collection of same – type data that are stored contiguously and that are accessible by an index.”

“An array are the simplest and most common type of structured data.”

Example:-

Declaring an Array:

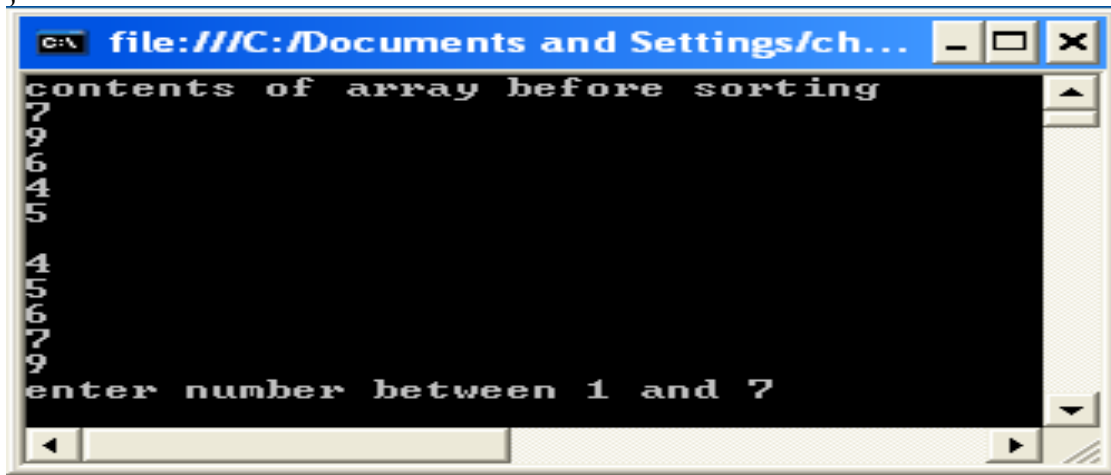
```
using System;
class test
{
    public static void Main()
    {
        int[] a={ 7, 9, 6, 4, 5 };
        Console.WriteLine("contents of array before sorting");
        dispme(a);
    }
}
```



```

        Array.Sort(a);
        Console.WriteLine();
        dispme(a);
        Console.WriteLine("enter number between 1 and 7");
        int c = Convert.ToInt16(Console.ReadLine());
        int d = Array.IndexOf(a, c);
        Console.WriteLine(c + " " + d);
    }
    public static void dispme(Array b)
    {
        foreach (int t in b)
        {
            Console.WriteLine(t);
        }
    }
}

```



Example:

```

Using System;
namespace console1
{
    Class arvind1
    {
        public static void Main()
        {
            int[] values = new int[3];
            values[0] = 5;
            values[1] = values[0] * 2;
            values[2] = values[1] * 2;
            foreach (int value in values)
            {
                Console.WriteLine(value);
            }
        }
    }
}

```

Output:



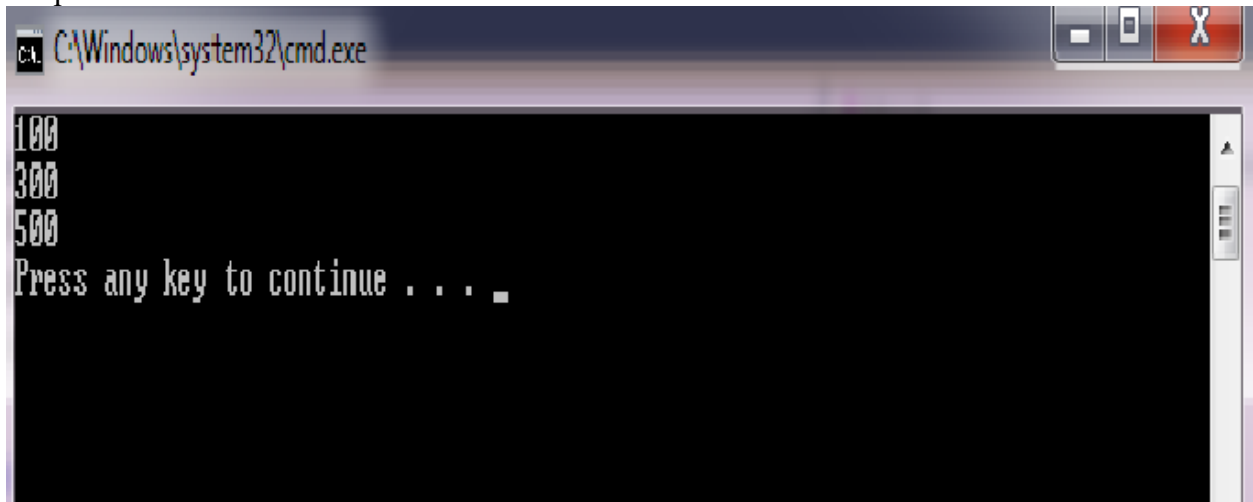
```
C:\Windows\system32\cmd.exe
5
10
20
Press any key to continue . . .
```

For Loop Array:

Program that uses for-loop on int array: C#

```
using System;
class arvind
{
    static void Main ()
    {
        Int[] array = { 100,300,500 };
        For(int I =0; i< array.Length; i++)
        {
            Console.WriteLine(array[i]);
        }
    }
}
```

Output: -



```
C:\Windows\system32\cmd.exe
100
300
500
Press any key to continue . . . .
```

String Array:

Program that create string array: C#

```
using System;
namespace consoleApplication4
{
    class arvind1
    {
```

```

public static void Main()
{
string[] array = new string[4];
array[0 ] = "DOT";
array[1 ] = "NET";
array[2 ] = "ANU";
{
Console.WriteLine("DOT");
Console.WriteLine("NET");
Console.WriteLine("ANU");
}
}
}
}
}

```

Output:

```

C:\Windows\system32\cmd.exe
DOT
NET
ANU
Press any key to continue . . .

```

Return Array:

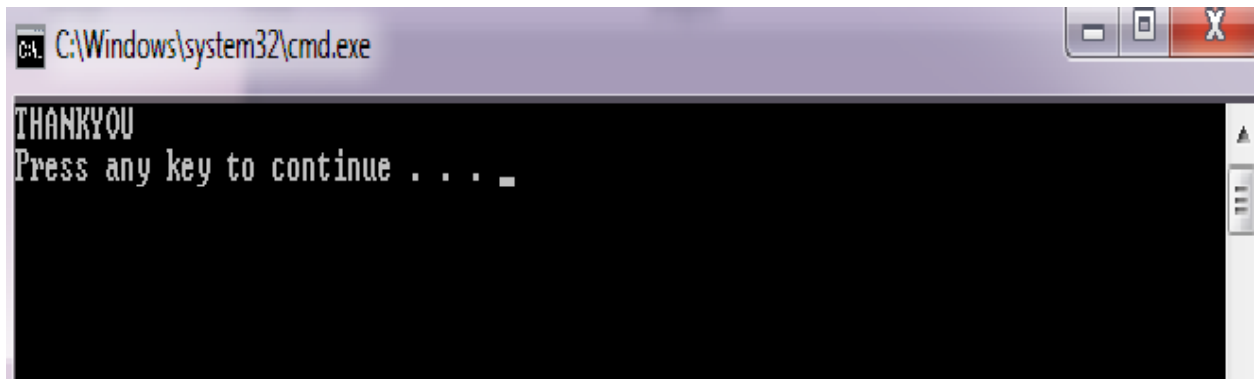
Program that returns array references: C#

```

using System;
class arvind2
{
public static void Main()
{
Console.WriteLine(string.Join(" ", Method()));
}
static string[] Method()
{
string[] array = new string[2];
array[0] = "THANK";
array[1] = "YOU";
return array;
}
}

```

Output:



NESTED STRUCTURE:

In C#, **Structs** can also be nested inside the other struct body. The **nested struct** is called the member struct of the parent struct. The accessibility of the nested struct depends upon the access modifier used to declare it inside the struct body. There are two ways of nesting the C# **structs** inside the other one. The first way is nesting the whole declaration of a struct inside the other struct and the second way is to use the struct as a member variable of another struct. In this tutorial we will learn the syntax to declare the nested structs, member structs and their implementation in the other code.

Simple Example:

```
struct A
{
    public struct B
    {
    }
}
```

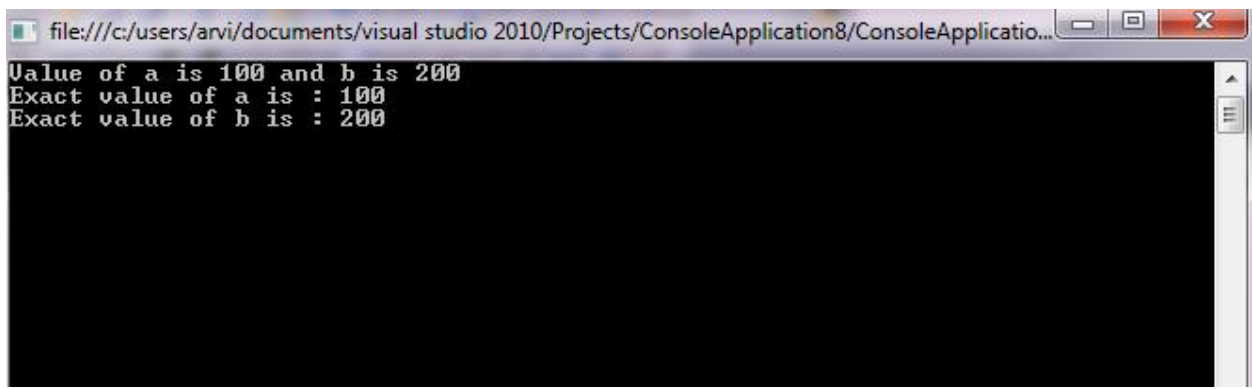
Example of nested structure:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace DecisionMaking
{
    class arvind
    {
        static void Main(string[] args)
        {
            int a = 100;
            int b = 200;
            if (a == 100)
            {
```

```

        if (b == 200)
        {
            Console.WriteLine("Value of a is 100 and b is 200");
        }
    }
    Console.WriteLine("Exact value of a is : {0}", a);
    Console.WriteLine("Exact value of b is : {0}", b);
    Console.ReadLine();
}
}
}

```



Difference between Classes & Structures:

Visual Basic .NET unifies the syntax for structures and classes, with the result that both entities support most of the same features. However, there are also important differences between structures and classes.

Similarities

Structures and classes are similar in the following respects:

Both are *container* types, meaning that they contain other types as members.

Both have members, which can include constructors, methods, properties, fields, constants, enumerations, events, and event handlers.

Members of both can have individualized accessibilities. For example, one member can be declared **Public** and another **Private**.

Both can implement interfaces.

Both can have shared constructors, with or without parameters.

Both can expose a default property, provided that property takes at least one argument.

Both can declare and raise events, and both can declare delegates.

Differences

Structures and classes differ in the following particulars:

Structures are value types; classes are reference types.

Structures use stack allocation; classes use heap allocation.

All structure members are **Public** by default; class variables and constants are **Private** by default, while other class members are **Public** by default. This behavior for class members provides compatibility with the Visual Basic 6.0 system of defaults.

A structure must have at least one nonshared variable or event member; a class can be completely empty.

Structure members cannot be declared as **Protected**; class members can.

A structure procedure can handle events only if it is a **Shared Sub** procedure, and only by means of the **AddHandler** statement; any class procedure can handle events, using either the **Handles** keyword or the **AddHandler** statement.

Structure variable declarations cannot specify initializers, the **New** keyword, or initial sizes for arrays; class variable declarations can.

Structures implicitly inherit from the **ValueType** class and cannot inherit from any other type; classes can inherit from any class or classes other than **ValueType**.

Structures are not inheritable; classes are.

Structures are never terminated, so the common language runtime (CLR) never calls the **Finalize** method on any structure; classes are terminated by the garbage collector, which calls **Finalize** on a class when it detects there are no active references remaining.

A structure does not require a constructor; a class does.

Enumerations:

Enums store special values. They make programs simpler. If you place constants directly where used, your C# program becomes complex. It becomes hard to change. Enums instead keep these magic constants in a distinct typeEnum Type Conversion.

In this first example, we see an enum type that indicates the importance of something. An enum type internally contains an enumerator list. You will use enum when you want readable code that uses constants.

An enum type is a distinct value type that declares a set of named constants.

Program that uses enum:C#

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

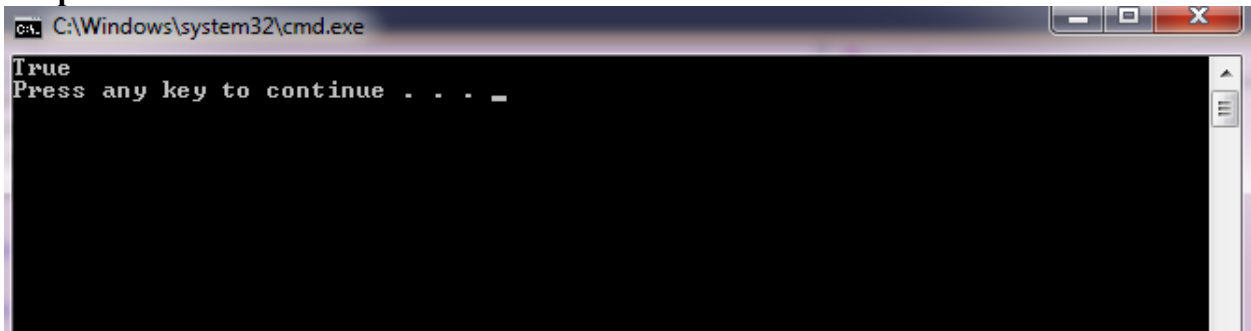
namespace DecisionMaking
{
    class Program
    {
        enum Importance
        {
            None,
            Trivial,
            Regular,
            Important,
            Critical
        };
        static void Main()
        {
            Importance value = Importance.Critical;
            if (value == Importance.Trivial)
            {
                Console.WriteLine("Not true");
            }
        }
    }
}
```

```

    }
    else if (value == Importance.Critical)
    {
        Console.WriteLine("True");
    }
}
}
}

```

Output:



1 D Array:

The one dimensional array or single dimensional array in C# is the simplest type of array that contains only one row for storing data. It has single set of square bracket ("[]"). To declare single dimensional array in C#, you can write the following code.

```
int[] age = new int[6];
```

The array age is a one dimensional array that contains only 6 elements in a single row.

Example:

```
using System;
```

```
namespace One_Dimensional_Array
```

```
{
```

```
    class Program
```

```
    {
```

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

```
        {
```

```
            string[] Books = new string[5];
```

```
            Books[0] = "C#";
```

```
            Books[1] = "Java";
```

```
            Books[2] = "VB.NET";
```

```
            Books[3] = "C++";
```

```
            Books[4] = "C";
```

```
            Console.WriteLine("All the element of Books array is:\n\n");
```

```
            int i=0;
```

```

        Console.WriteLine("\t1\t2\t3\t4\t5\n\n\t");
        for (i=0;i<5;i++)
        {
            Console.WriteLine("{0}\t",Books[i]);

        }
    Console.ReadLine();

}
}
}

```

Output:

All The Element of Books array is:

```

All the element of Books array is:

```

1	2	3	4	5
C#	Java	VB.NET	C++	C

2 D Array:

Data is sometimes two-dimensional. The C# language offers **2D arrays** which are useful here. Two-dimensional arrays are indexed with two numbers. They use a special syntax form. They store any element type, reference or value.

The rank of an array is the number of dimensions. The type of an array (other than a vector) shall be determined by the type of its elements and the number of dimensions.

The arrays we used so far were made of a uniform series, where all members consisted of a simple list, like a column of names on a piece of paper. Also, all items fit in one list. In some cases, you may want to divide the list in delimited sections. For example, if you create a list of names, you may want part of the list to include family members and another part of the list to include friends. Instead of creating a second list, you can add a second dimension to the list. In other words, you would like to create a list of a list, or one list inside of another list, although the list is still made of items with common characteristics.

A multi-dimensional array is a series of lists so that each list contains its own list. For example, if you create two lists of names, you would have an array of two lists. Each array or list would have its own list or array.

Example:

```

using System;
namespace DepartmentStore4
{
    class birla
    {

```

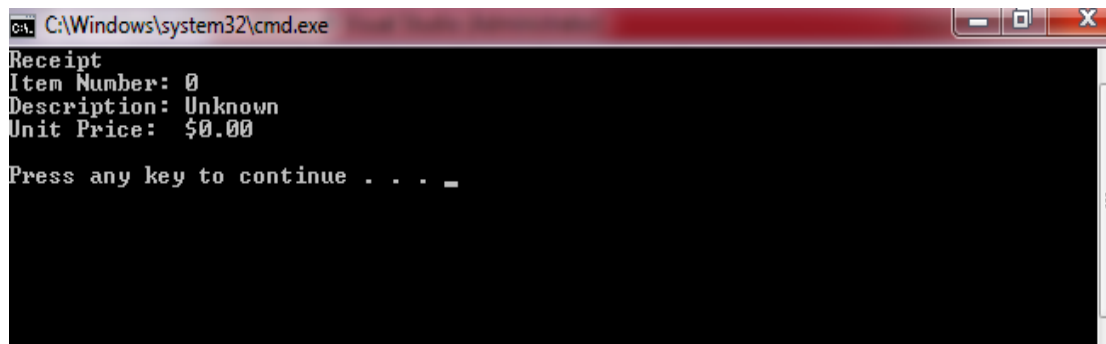


```

static int Main(string[] args)
{
    long ItemID = 0;
    string Description = "Unknown";
    decimal Price = 0.00M;

    Console.WriteLine("Receipt");
    Console.WriteLine("Item Number: {0}", ItemID);
    Console.WriteLine("Description: {0}", Description);
    Console.WriteLine("Unit Price: {0:C}\n", Price);
    return 0;
}
}
}

```



```

C:\Windows\system32\cmd.exe
Receipt
Item Number: 0
Description: Unknown
Unit Price: $0.00
Press any key to continue . . . _

```

Dynamic Array:-

When we talk about arrays which means that successive memory locations. Arrays are used to minimize using different variable and arrays allowed to access each element easily rather than by writing bulk of code for different variables. Static arrays can be defined easily in any language, but static arrays has a great disadvantage, which is that if you have not used full array it will always take same size as that was defined during its declaration. So in practice we use dynamic array. Dynamic Array means that you define the size of array at runtime, or making room for new elements in array. In c/c++ this has to be done with pointers or linked lists. But C# does it very easily i.e you do not have to play with pointers or any references. C# does it for you very easily. This can be achieved by using LIST class in C#. List in C# can be used for many purposes but major use of it is using it as dynamic array. List can be of any datatype or can also be the user defined datatype i.e objects.

Example of Dynamic Array:-

```

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

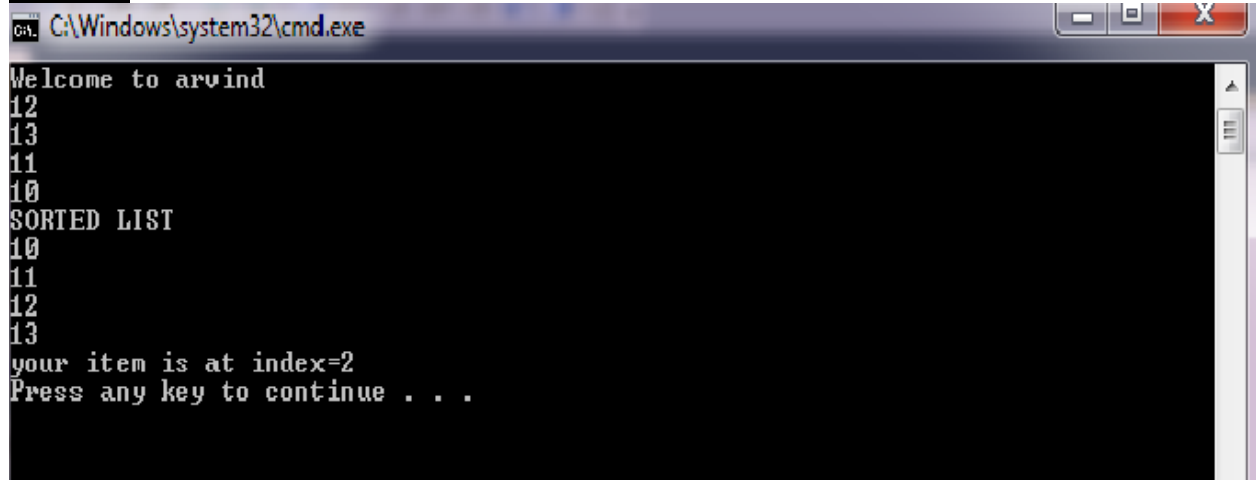
```

```

namespace ConsoleApplication8
{
class xyz
{
static void Main(string[] args)
{
Console.WriteLine("Welcome to arvind");
List<int> list=new List<int>();
list.Add(12);
list.Add(13);
list.Add(11);
list.Add(10);
int size = list.Count; //size of list// accessing the items in list
for(int i=0;i<list.Count;i++)
Console.WriteLine(list[i]);// printing every element on console //sorting the list
list.Sort();
Console.WriteLine("SORTED LIST");
for (int i = 0; i < list.Count; i++)
Console.WriteLine(list[i]);// printing every element on console// search operation
int item=list.BinarySearch(12);
Console.WriteLine("your item is at index="+item);
}
}
}

```

Output:-



```

C:\Windows\system32\cmd.exe
Welcome to arvind
12
13
11
10
SORTED LIST
10
11
12
13
your item is at index=2
Press any key to continue . . .

```

4.11. SUMMARY

In this chapter we explained structures and arrays with their suitable examples. Structures are value types. We discussed nested structures also. The audience also learns basics of arrays (single & two dimensional), dynamic arrays also.

4.12. EXERCISE

- (i) What is the use of enumerated data type?
- (ii) What is dynamic array? How can we use it?
- (iii) How nested structures are declared?
- (iv) What are the difference between Structure and Class?
- (v) What is the difference between Array and ArrayList?
- (vi) **What are value types and reference types?**
- (vii) Declare a structure with three member variables one of them is array type of any primary data type, and using the object of structure access these members.

BLOCK- 2

U n i t

6

Inheritance and Polymorphism

Interface:

Interfaces describe a group of related functionalities that can belong to any class or struct. Interfaces can consist of methods, properties, events, indexers, or any combination of those four member types. An **Interface** is a *reference type* and it contains only *abstract members*, so sometime it also called *pure abstract class*. An interface contains only declaration for its members. Any implementation must be placed in class that inherited them. An interface can't contain constants, data fields, constructors, destructors and static members. All the member declarations inside interface are implicitly public.

To implement an interface member, the corresponding member on the class must be public, non-static, and have the same name and signature as the interface member. Properties and indexers on a class can define extra accessors for a property or indexer defined on an interface.

Classes and structs can inherit from more than one interface. An interface can itself inherit from multiple interfaces.

Defining an interface:

interface IDemoInterface

```
{  
    void MethodDemo();  
}
```

We define an interface by using keyword interface. Above an interface name is IDemoInterface. A common naming convention is to prefix all interface names with a capital "I".

Example:

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

```
namespace DemoInterface  
{
```

```
    interface IDemoInterface // defining an interface  
    {  
        void MethodDemo();  
    }
```

```
        class MyClass : IDemoInterface // implementing an interface in the  
class
```

```

{
    static void Main(string[] args)
    {
        MyClass mco = new MyClass ();
        mco.MethodDemo();
    }

    public void MethodDemo()
    {
        Console.WriteLine("An example of Interface.");
        Console.ReadLine();
    }
}
}

```

OUTPUT:

An example of Interface.

Explanation:

In above example we define an interface named IDemoInterface which contains a method named MethodDemo().

Now, a class named MyClass inherits the interface IDemoInterface using : symbol.

```
class MyClass : IDemoInterface
```

The class MyClass implement the method MethodDemo() which is defined by the interface IDemoInterface. Then we create an object of class MyClass named *mco* and called the method MehtodDemo().

So, the above description defines the actual story of an interface.

Multiple Inheritance using C# interfaces:

Multiple inheritance is not allowed in C#. So, this problem can solve by using interface. This can be done using child class that inherits from any number of c# interfaces.

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Text;
```

```
namespace DemoInterface
```

```

{
    interface IParentInterafce
    {
        void MethodParentDemo();
    }
    interface IDemoInterface
    {
        void MethodDemo();
    }
}

```

```
class MyClass : IParentInterafce, IDemoInterface
```

```

{
    public static void Main(string[] args)
    {
        MyClass mco = new MyClass ();
        mco.MethodDemo();
        mco.MethodParentDemo();
    }

    public void MethodDemo()

```

```

    {
        Console.WriteLine("Implementing IDemoInterface");
    }
    public void MethodParentDemo()
    {
        Console.WriteLine("Implementing IParentInterafce");
        Console.ReadLine();
    }
}

```

}

OUTPUT:

Implementing IDemoInterface

Implementing IParentInterafce

Explanation:

In above example we have defined two interfaces named IParentInterafce and IDemoInterface. Then we have inherited both interfaces in class named MyClass.

So, by doing this we can resolve the problem of multiple inheritance.

An interface can inherit other interface as below:

```

interface IParentInterafce
{
    void MethodParentDemo();
}
interface IDemoInterface :IParentInterafce
{
    void MethodDemo();
}

```

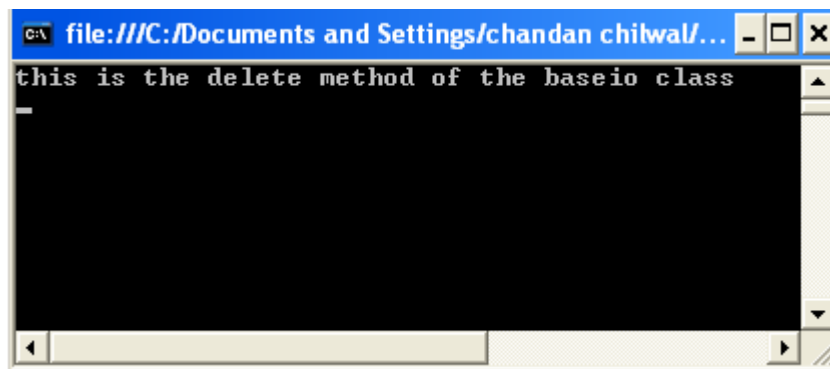
Implementation of both interface will same as in above exaple.

//example of inheritance

```
using System;
class baseio
{
    public string filename;
    public int delete()
    {
        Console.WriteLine("this is the delete method of the baseio
class");
        return(0);
    }
}

class imageio:baseio
{
    public string imgformat;
    public int alphablend()
    {
        Console.WriteLine("this is the alpha blending method of the
imageio class");
        return(0);
    }
}

class test
{
    static void Main(string[] args)
    {
        imageio obj=new imageio();
        obj.filename=("shaggy.jpg");
        obj.delete();
    }
}
```



Overloading and Overriding

Method Overloading

Method overloading occurs when a class contains two or more methods with the same name, but different signatures.

Method overloading use in such situations in which you want to expose a single method name where the behavior of the method is slightly different depending on the value types passed.

In method overloading, the arguments passed in methods should be different in types, number and in sequence.

//syntax of method overloading using different number of parameters

```
Public void admission(int t,string s)
{
//code
//admission1 implementation
}
Public void admission(int t,string s,int a)
{
//code
//admission2 implementation
}
Public admission()
{
//code
//admission3 implementation
}
```

//syntax of method overloading using different type of parameter

```
Public void admission(short t,string s)
{
//admission1 implementation
}
```

```

}
Public void admission(int t,string s)
{
//admission2 implementation
}
Public admission(bool b,int t)
{
//admission3 implementation
}

```

For example:

```

using System;
using System.Collections.Generic;
using System.Text;

class MyDemo
{
    public int Add(int a, int b)
    {
        return (a + b);
    }

    public int Add(int a, int b, int c)
    {
        return (a + b + c);
    }
    public string Add(string a, string b)
    {
        return (a + b );
    }

}

class MainClass
{
    public static void Main(string[] args)
    {

```

```

        MyDemo md = new MyDemo();

        Console.WriteLine(md.Add(4, 6));
        Console.WriteLine(md.Add("C# Programing", "Language"));
        Console.WriteLine(md.Add(4, 6, 8));
        Console.ReadLine();

    }
}

```

Output :

```

10
C# Proraming Language
18

```

Explanation:

In above example we have defined three methods with the same name *ADD* but with different arguments within a class *MyDemo*.

In first method we are passing two arguments int a and int b.

In second method we are passing three arguments int a, int b and int c. So in this method we have change the number of arguments to perform overloadind.

In third method we are passing two different arguments in type string a and string b. So in this method we have type of arguments.

The compiler automatically select the most appropriate method to call based on the arguments supplied.

Program 1:

```

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

namespace ConsoleApplication2{
    class myFirst{
        public void hello(){
            Console.WriteLine("Hello C#");
        }
    }
}

```

```

    }
    static void Main(string[] args){
        myFirst m = new myFirst();
        m.hello();
        Console.Read();
    }
}

```



Program 2: Internal Class

```

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

namespace ConsoleApplication2{
    internal class myFirst{
        public void hello(){
            Console.WriteLine("Hello C#");
        }
    }
    class Mainclass{
        static void Main(string[] args){
            myFirst m = new myFirst();
            m.hello();
        }
    }
}

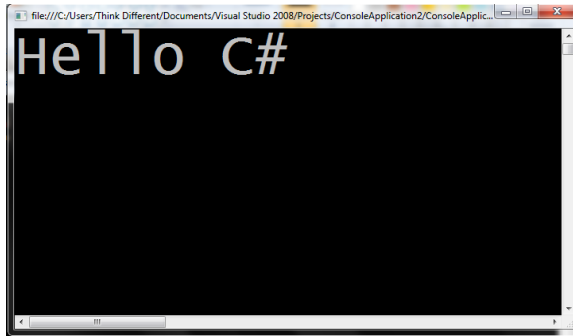
```



Program 3:

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

namespace ConsoleApplication2{
    abstract class abshello{
        protected string s = "";
        public abstract void callhello();
    }
    class absDerived : abshello{
        public override void callhello(){
            s = "Hello C#";
            Console.WriteLine(s);
        }
    }
    class mainclass{
        static void Main(string[] args){
            absDerived ad = new absDerived();
            ad.callhello();
            Console.Read();
        }
    }
}
```



Program 4:

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

namespace ConsoleApplication2{
    abstract class abshello{
        protected string s = "";
        public abstract void callhello();
        public abstract void sayhello(); //Not
implemented in the Derived class.
    }
    class absDerived : abshello{
        public override void callhello(){
            s = "Hello C#";
            Console.WriteLine("{0}",s);
        }
    }
    class mainclass{
        static void Main(string[] args){
            absDerived ad = new absDerived();
            ad.callhello();
        }
    }
}
```

Error'ConsoleApplication2.absDerived' does not implement inherited abstract member 'ConsoleApplication2.abshello.sayhello()'

Program 5

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

namespace ConsoleApplication2{
    class abshello{
        protected string s = "";
        public void callhello(){
            s = "No Hello C#";
        }
    }
    class absDerived : abshello{
        public override void callhello(){
            s = "Hello C#";
            Console.WriteLine(s);
        }
    }
    class Program{
        static void Main(string[] args)
        {
            absDerived ad = new absDerived();
            ad.callhello();
        }
    }
}
```

Error 'ConsoleApplication2.absDerived.callhello()': cannot override inherited member 'ConsoleApplication2.abshello.callhello()' because it is not marked virtual, abstract, or override

Program 6

```
using System;
using System.Collections.Generic;
```

```

using System.Linq;
using System.Text;

namespace ConsoleApplication2{
    class abshello{
        protected string s = "";
        public virtual void callhello(){
            s = "No Hello C#";
        }
    }
    class absDerived : abshello{
        public override void callhello(){
            s = "Hello C#";
            Console.WriteLine(s);
        }
    }
    class Program{
        static void Main(string[] args)
        {
            absDerived ad = new absDerived();
            ad.callhello();
        }
    }
}

```



Program 7

```

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

```



```

namespace ConsoleApplication2
{
    class abshello
    {
        string s = "Hello C#";
        public static void callhello()
        {
            Console.WriteLine("{0}",s);
        }
        public void normal()
        {
            Console.WriteLine("{0}",s);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            abshello.callhello();
            abshello ad = new abshello();
            ad.normal();
        }
    }
}

```

Error An object reference is required for the non-static field, method, or property
'ConsoleApplication2.abshello.s'

Program 7

```

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

namespace ConsoleApplication2
{

```

```
class shibi
{
    public void hand()
    {
        Console.WriteLine("shibi's Hand");
    }
    public virtual void eye()
    {
        Console.WriteLine("Shibi's Hand");
    }
}
class myson : shibi
{
    public void hand()
    {
        Console.WriteLine("myson's Hand");
    }
    public override void eye()
    {
        Console.WriteLine("myson's eye");
    }
}

class Program
{
    static void Main(string[] args)
    {
        shibi s = new shibi();
        myson m = s;
        s.hand();
        m.hand();
        s.eye();
        m.eye();
    }
}
```

//program of virtual function

```
using System;
class draw1
{
    public virtual void draw()
    {
        System.Console.WriteLine("this is the virtual draw
method");
    }
}
class rectangle : draw1
{
    public override void draw()
    {
        System.Console.WriteLine("this is the draw method of
rectangle");
    }
}
class triangle:draw1
{
    public override void draw()
    {
        System.Console.WriteLine("this is the draw method of
triangle");
    }
}
class polygon:draw1
{
    public override void draw()
    {
        System.Console.WriteLine("this is the draw method of
polygon");
    }
}
```

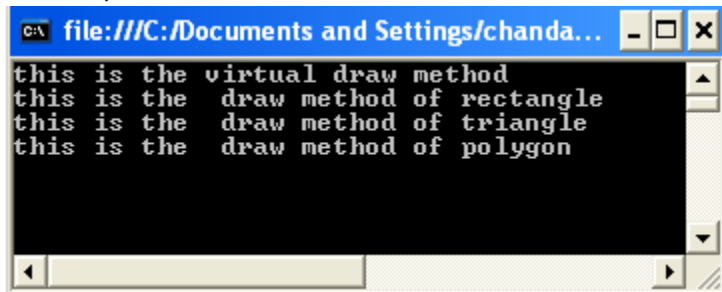
```
public class Class1
{
    public static void Main(string[] args)
    {
        draw1[] d=new draw1[4];
        d[0]=new draw1();
        d[1]=new rectangle();
        d[2]=new triangle();
        d[3]=new polygon();

        foreach(draw1 t in d)
        {
            t.draw();
        }
    }
}
```

```
}
```

```
}
```

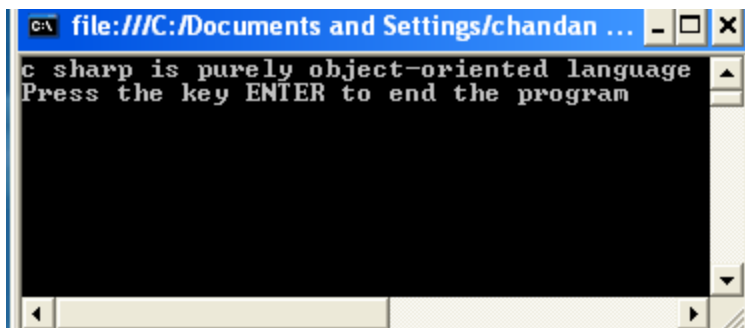
```
}
```



A screenshot of a Windows command prompt window. The title bar shows the file path "C:\ file:///C:/Documents and Settings/chanda...". The window contains the following text:

```
this is the virtual draw method  
this is the draw method of rectangle  
this is the draw method of triangle  
this is the draw method of polygon
```

```
using System;  
  
namespace Sealed  
{  
    public sealed class First  
    {  
        public void TestName()  
        {  
            Console.WriteLine("c sharp is purely object-oriented");  
        }  
    }  
  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            First f = new First();  
            f.TestName();  
            Console.WriteLine("Press the key ENTER to end the  
program");  
            Console.ReadLine();  
        }  
    }  
}
```



```
file:///C:/Documents and Settings/chandan ...  
c sharp is purely object-oriented language  
Press the key ENTER to end the program
```

8

Event & Delegates

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

namespace ConsoleApplication10
{
    public delegate void MulticastDelegate(int x, int y);
    public class MyClass
    {
        public static void Add(int x, int y)
        {
            Console.WriteLine(" Add() Method");

            Console.WriteLine("{0} + {1} = {2}\n", x, y, x + y);
        }

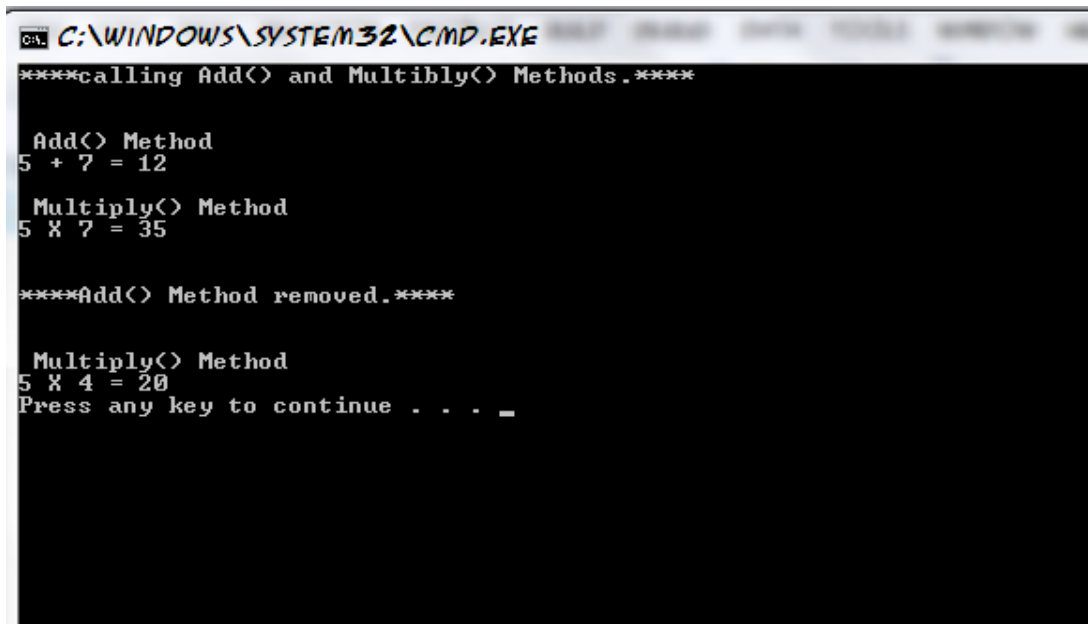
        public static void Multiply(int x, int y)
        {
            Console.WriteLine(" Multiply() Method");

            Console.WriteLine("{0} X {1} = {2}", x, y, x * y);
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        MulticastDelegate del = new MulticastDelegate(MyClass.Add);
        del += new MulticastDelegate(MyClass.Multiply);
        Console.WriteLine("****calling Add() and Multibly()
Methods.****\n\n");
        del(5, 7);
        del -= new MulticastDelegate(MyClass.Add);
        Console.WriteLine("\n\n****Add() Method
removed.****\n\n");
    }
}
```

```
        del(5, 4);  
    }  
}
```

Output :



```
C:\WINDOWS\SYSTEM32\CMD.EXE  
****calling Add() and Multibly() Methods.****  
  
Add() Method  
5 + 7 = 12  
  
Multiply() Method  
5 x 7 = 35  
  
****Add() Method removed.****  
  
Multiply() Method  
5 x 4 = 20  
Press any key to continue . . . _
```

```

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

namespace ConsoleApplication3
{
    public delegate int BinaryMath(int x, int y);

    public class SimpleMath
    {
        public int Add(int x, int y)
        { return x + y; }

        public int Subtract(int x, int y)
        { return x - y; }

        public static int SquareNumber(int a)
        { return a * a; }
    }

    class SimpleDelegate
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("***** A Simple Delegate Use *****\n");

            SimpleMath m = new SimpleMath();

            BinaryMath b = new BinaryMath(m.Add);

            DisplayDelegateInfo(b);

            Console.WriteLine("\n15 + 30 is {0}", b(15, 30));

            Console.ReadLine();
        }

        static void DisplayDelegateInfo(Delagate delObj)
        {

```

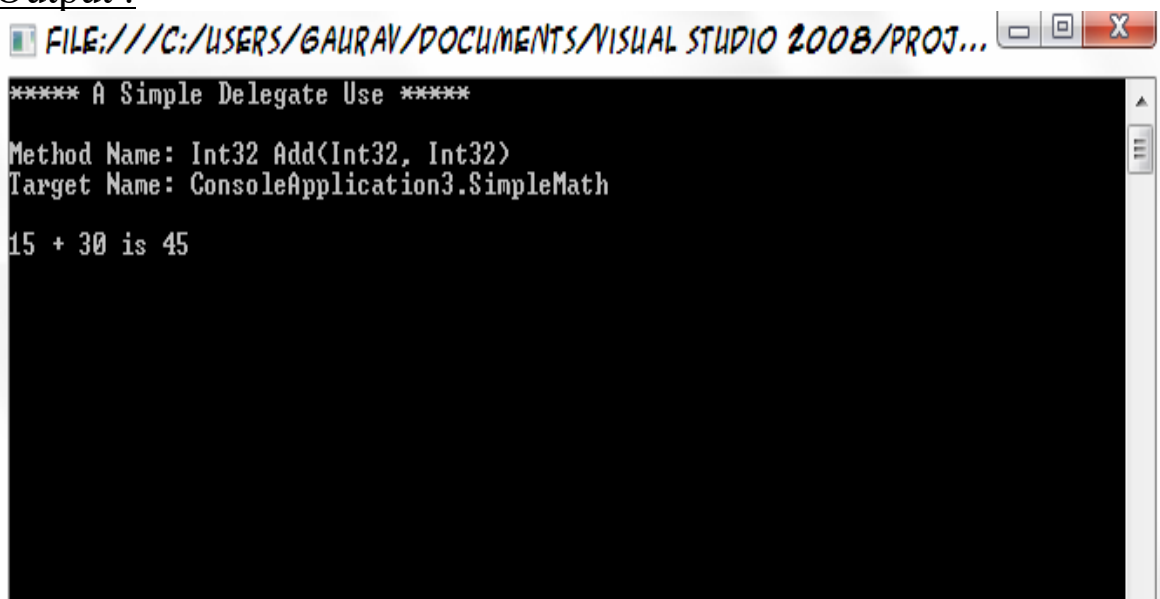


```

        foreach (Delegate d in delObj.GetInvocationList())
        {
            Console.WriteLine("Method Name: {0}", d.Method);
            Console.WriteLine("Target Name: {0}", d.Target);
        }
    }
}

```

Output :



```

***** A Simple Delegate Use *****
Method Name: Int32 Add(Int32, Int32)
Target Name: ConsoleApplication3.SimpleMath
15 + 30 is 45

```

```

using System;
using System.Collections.Generic;
using System.Linq;

```

```

using System.Text;

namespace ConsoleApplication4
{
    using System;

    public delegate int Comparer(object obj1, object obj2);

    public class Name
    {
        public string FirstName = null;
        public string LastName = null;

        public Name(string first, string last)
        {
            FirstName = first;
            LastName = last;
        }

        // this is the delegate method handler
        public static int CompareFirstNames(object name1, object name2)
        {
            string n1 = ((Name)name1).FirstName;
            string n2 = ((Name)name2).FirstName;

            if (String.Compare(n1, n2) > 0)
            {
                return 1;
            }
            else if (String.Compare(n1, n2) < 0)
            {
                return -1;
            }
            else
            {
                return 0;
            }
        }

        public override string ToString()
        {
            return FirstName + " " + LastName;
        }
    }

    class SimpleDelegate
    {
        Name[] names = new Name[5];

        public SimpleDelegate()
        {
            names[0] = new Name("RAM ", "SHARMA");
            names[1] = new Name("BASANT ", "PANDEY");
            names[2] = new Name("TINU", "JOSHI");
            names[3] = new Name("RAKESH", "BISHT");
        }
    }
}

```

```

        names[4] = new Name("ANUBHAV", "KHANDURI");
    }

    public static void Main(string[] args)
    {
        SimpleDelegate sd = new SimpleDelegate();

        // this is the delegate instantiation
        Comparer cmp = new Comparer(Name.CompareFirstNames);

        Console.WriteLine("\nBefore Sort: \n");

        sd.PrintNames();

        // observe the delegate argument
        sd.Sort(cmp);

        Console.WriteLine("\nAfter Sort: \n");

        sd.PrintNames();
    }

    // observe the delegate parameter
    public void Sort(Comparer compare)
    {
        object temp;

        for (int i = 0; i < names.Length; i++)
        {
            for (int j = i; j < names.Length; j++)
            {
                // using delegate "compare" just like
                // a normal method
                if (compare(names[i], names[j]) > 0)
                {
                    temp = names[i];
                    names[i] = names[j];
                    names[j] = (Name)temp;
                }
            }
        }
    }

    public void PrintNames()
    {
        Console.WriteLine("Names: \n");

        foreach (Name name in names)
        {
            Console.WriteLine(name.ToString());
        }
    }
}

```

Output :

```
C:\WINDOWS\SYSTEM32\CMD.EXE
Before Sort:
Names:
RAM SHARMA
BASANT PANDEY
TINU JOSHI
RAKESH BISHT
ANUBHAV KHANDURI
After Sort:
Names:
ANUBHAV KHANDURI
BASANT PANDEY
RAKESH BISHT
RAM SHARMA
TINU JOSHI
Press any key to continue . . .
```

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

namespace ConsoleApplication8
```

```

{
    delegate bool CompareOp(object lhs, object rhs);
    class BubbleSorter
    {
        static public void Sort(object [] sortArray, CompareOp gtMethod)
        {
            for(int i=0; i<sortArray.Length ; i++)
            {
                for(int j=i+1; j<sortArray.Length ; j++)
                {
                    if(gtMethod (sortArray[j], sortArray[i]))
                    {
                        object temp=sortArray [i];
                        sortArray [i]=sortArray [j];
                        sortArray [j]=temp;
                    }
                }
            }
        }
    }
}

class Employee
{
    private string name;
    private decimal salary;
    public Employee (string name, decimal salary)
    {
        this.name = name;
        this.salary = salary;
    }
    public override string ToString()
    {
        return string.Format (name + ", {0:C}", salary );
    }
    public static bool RhsIsGreater(object lhs, object rhs)
    {
        Employee empLhs = (Employee) lhs;
        Employee empRhs = (Employee) rhs;
        return (empRhs.salary > empLhs.salary) ? true : false;
    }
}

class Program
{
    public static void Main(string[] args)
    {
        Employee [] employees =
        {
            new Employee("AKASH SHARMA", 500000),
            new Employee("FAHEEM KHAN", 10000),
            new Employee("GANESH SHANKAR", 25000),
            new Employee("PANKAJ SINGH", (decimal)100000.35),
            new Employee("RAJKUMAR", 5000),
            new Employee("RAJ SHARMA", 100000),
        };
        CompareOp employeeCompareOp = new CompareOp
(Employee.RhsIsGreater);
    }
}

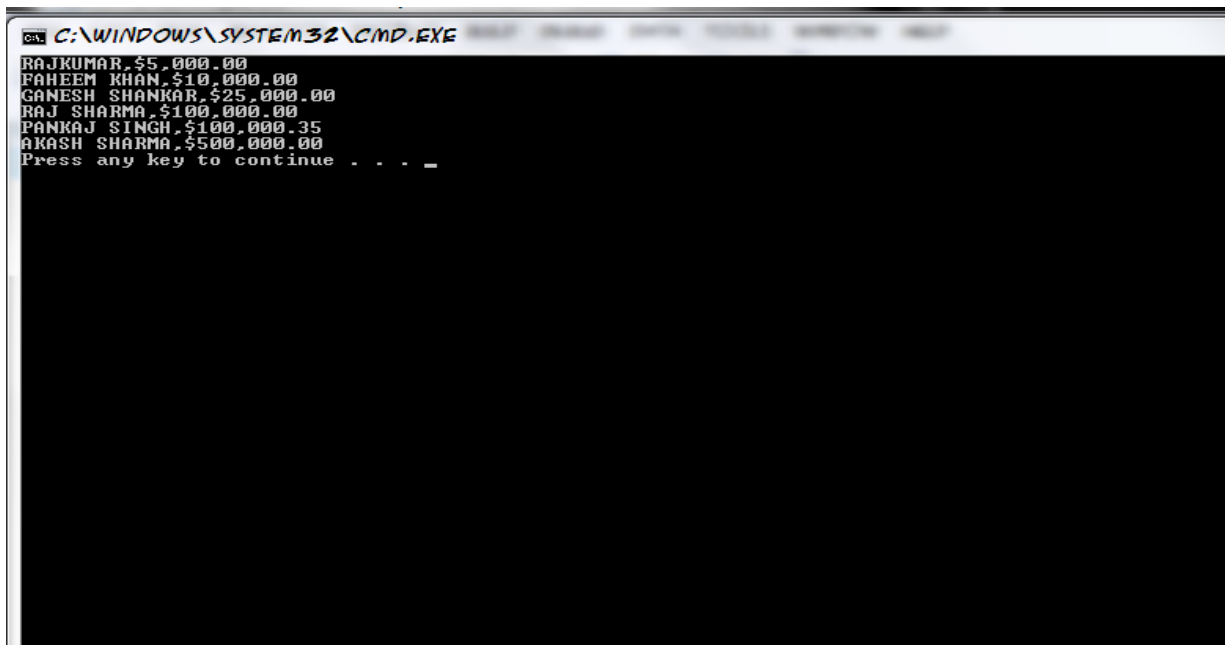
```

```

        BubbleSorter.Sort(employees, employeeCompareOp );
        for (int i=0;i<employees.Length ; i++)
        Console.WriteLine(employees[i].ToString());
    }
}

```

Output :



```

C:\WINDOWS\SYSTEM32\CMD.EXE
RAJKUMAR, $5,000.00
FAHEEM KHAN, $10,000.00
GANESH SHANKAR, $25,000.00
RAJ SHARMA, $100,000.00
PANKAJ SINGH, $100,000.35
AKASH SHARMA, $500,000.00
Press any key to continue . . . -

```

```

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

namespace ConsoleApplication14
{
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }
    class Program
    {
        public delegate bool FilterDelegate(Person p);

```

```

static void Main(string[] args)
{
    Person p1 = new Person() { Name = "RAM", Age = 40 };
    Person p2 = new Person() { Name = "JAIDUTT", Age = 69 };
    Person p3 = new Person() { Name = "SANSKRITI", Age = 12 };
    Person p4 = new Person() { Name = "BASANT", Age = 25 };
    List<Person> people = new List<Person>() { p1, p2, p3, p4
};

    DisplayPeople("Children:", people, IsChild);
    DisplayPeople("Adults:", people, IsAdult);
    DisplayPeople("Seniors:", people, IsSenior);

    Console.Read();
}

static void DisplayPeople(string title, List<Person> people,
FilterDelegate filter) {
    Console.WriteLine(title);
    foreach (Person p in people) {
        if (filter(p)) {
            Console.WriteLine("{0}, {1} years old", p.Name, p.Age);
        }
    }
    Console.Write("\n\n");
}

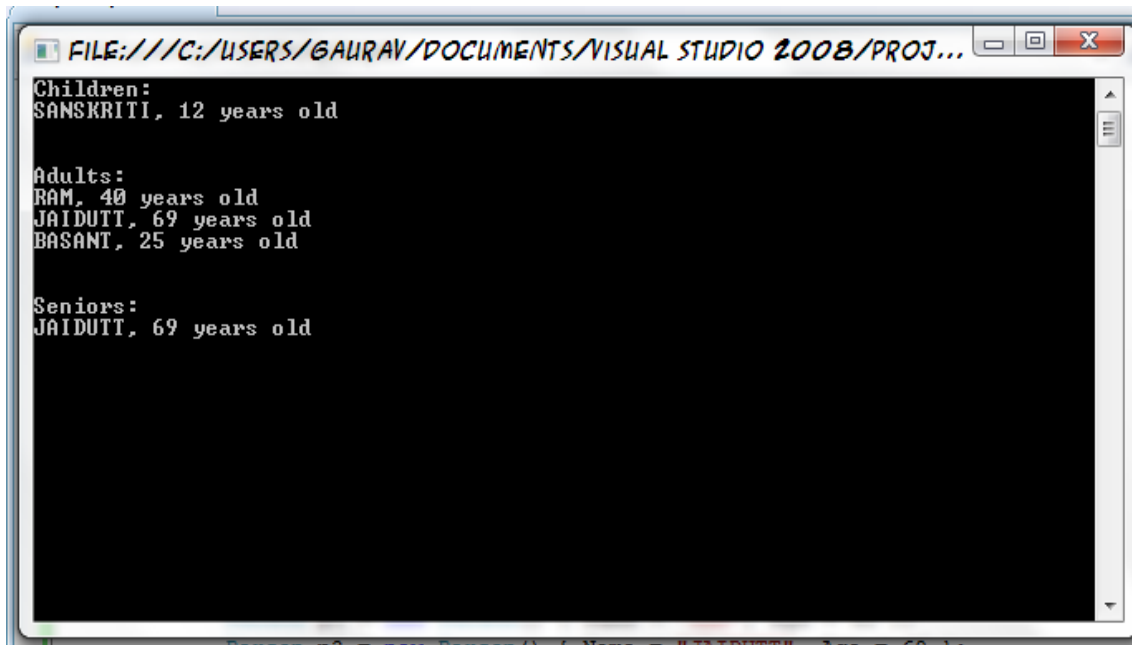
//=====FILTERS=====
static bool IsChild(Person p) {
    return p.Age <= 18;
}

static bool IsAdult(Person p) {
    return p.Age >= 18;
}

static bool IsSenior(Person p) {
    return p.Age >= 65;
}
}
}
}

```

Output ;



A screenshot of a Visual Studio 2008 console window. The title bar shows the file path: `FILE:///C:/USERS/GAURAV/DOCUMENTS/VISUAL STUDIO 2008/PROJ...`. The console output is as follows:

```
Children:
SANSKRITI, 12 years old

Adults:
RAM, 40 years old
JAIDUTT, 69 years old
BASANT, 25 years old

Seniors:
JAIDUTT, 69 years old
```

At the bottom of the console, a line of code is partially visible: `Person p2 = new Person() { Name = "JAIDUTT", Age = 69 };`

Properties & Indexer

Overview of Properties

Properties provide the opportunity to protect a field in a class by reading and writing to it through the property. In other languages, this is often accomplished by programs implementing specialized getter and setter methods. C# properties enable this type of protection while also letting you access the property just like it was a field.

Another benefit of properties over fields is that you can change their internal implementation over time. With a public field, the underlying data type must always be the same because calling code depends on the field being the same. However, with a property, you can change the implementation. For example, if a customer has an ID that is originally stored as an int, you might have a requirements change that made you perform a validation to ensure that calling code could never set the ID to a negative value. If it was a field, you would never be able to do this, but a property allows you to make such a change without breaking code. Now, let's see how to use properties.

Traditional Encapsulation Without Properties

Languages that don't have properties will use methods (functions or procedures) for encapsulation. The idea is to manage the values inside of the object, state, avoiding corruption and misuse by calling code. Listing 10-1 demonstrates how this traditional method works, encapsulating *Customer* information via accessor methods.

Listing 10-1. An Example of Traditional Class Field Access

```
using System;

public class Customer
{
    private int m_id = -1;

    public int GetID()
    {
        return m_id;
    }
}
```

```

    public void SetID(int id)
    {
        m_id = id;
    }

    private string m_name = string.Empty;

    public string GetName()
    {
        return m_name;
    }

    public void SetName(string name)
    {
        m_name = name;
    }
}

public class CustomerManagerWithAccessorMethods
{
    public static void Main()
    {
        Customer cust = new Customer();

        cust.SetID(1);
        cust.SetName("Amelio Rosales");

        Console.WriteLine(
            "ID: {0}, Name: {1}",
            cust.GetID(),
            cust.GetName());

        Console.ReadKey();
    }
}

```

Listing 10-1 shows the traditional method of accessing class fields. The *Customer* class has four methods, two for each private field that the class encapsulates: *m_id* and *m_name*. As you can see, *SetID* and *SetName* assign a new values and *GetID* and *GetName* return values.

Observe how *Main* calls the *SetXxx* methods, which sets *m_id* to 1 and *m_name* to "Amelio Rosales" in the *Customer* instance, *cust*. The call to *Console.WriteLine* demonstrates how to read *m_id* and *m_name* from *cust*, via *GetID* and *GetName* method calls, respectively.

This is such a common pattern, that C# has embraced it in the form of a language feature called properties, which you'll see in the next section.

Encapsulating Type State with Properties

The practice of accessing field data via methods was good because it supported the object-oriented concept of encapsulation. For example, if the type of *m_id* or *m_name* changed from an *int* type to *byte*, calling code would still work. Now the same thing can be accomplished in a much smoother fashion with properties, as shown in Listing 10-2.

Listing 10-2. Accessing Class Fields With Properties

```
using System;

public class Customer
{
    private int m_id = -1;

    public int ID
    {
        get
        {
            return m_id;
        }
        set
        {
            m_id = value;
        }
    }

    private string m_name = string.Empty;

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

public class CustomerManagerWithProperties
{
    public static void Main()
    {
        Customer cust = new Customer();

        cust.ID = 1;
        cust.Name = "Amelio Rosales";

        Console.WriteLine(
            "ID: {0}, Name: {1}",
            cust.ID,
            cust.Name);

        Console.ReadKey();
    }
}
```

```
}  
}
```

Listing 10-2 shows how to create and use a property. The *Customer* class has the *ID* and *Name* property implementations. There are also private fields named *m_id* and *m_name*; which *ID* and *Name*, respectively, encapsulate. Each property has two accessors, *get* and *set*. The *get* accessor returns the value of a field. The *set* accessor sets the value of a field with the contents of *value*, which is the value being assigned by calling code. The *value* shown in the accessor is a C# reserved word.

When setting a property, just assign a value to the property as if it were a field. The *CustomerManagerWithProperties* class uses the *ID* and *Name* properties in the *Customer* class. The first line of *Main* instantiates a *Customer* object named *cust*. Next the value of the *m_id* and *m_name* fields of *cust* are set by using the *ID* and *Name* properties.

To read from a property, use the property as if it were a field. *Console.WriteLine* prints the value of the *m_id* and *m_name* fields of *cust*. It does this by calling the *ID* and *Name* properties of *cust*.

This was a read/write property, but you can also create read-only properties, which you'll learn about next.

Creating Read-Only Properties

Properties can be made read-only. This is accomplished by having only a *get* accessor in the property implementation. Listing 10-3 demonstrates how you can create a read-only property.

Listing 10-3. Read-Only Properties

```
using System;  
  
public class Customer  
{  
    private int m_id = -1;  
    private string m_name = string.Empty;  
  
    public Customer(int id, string name)  
    {  
        m_id = id;  
        m_name = name;  
    }  
  
    public int ID  
    {  
        get  
        {  
            return m_id;  
        }  
    }  
}
```

```

        public string Name
        {
            get
            {
                return m_name;
            }
        }
    }

    public class ReadOnlyCustomerManager
    {
        public static void Main()
        {
            Customer cust = new Customer(1, "Amelio Rosales");

            Console.WriteLine(
                "ID: {0}, Name: {1}",
                cust.ID,
                cust.Name);

            Console.ReadKey();
        }
    }

```

The *Customer* class in Listing 10-3 has two read-only properties, *ID* and *Name*. You can tell that each property is read-only because they only have *get* accessors. At some time, values for the *m_id* and *m_name* must be assigned, which is the role of the constructor in this example.

The *Main* method of the *ReadOnlyCustomerManager* class instantiates a new *Customer* object named *cust*. The instantiation of *cust* uses the constructor of *Customer* class, which takes *int* and *string* type parameters. In this case, the values are *1* and *"Amelio Rosales"*. This initializes the *m_id* and *m_name* fields of *cust*.

Since the *ID* and *Name* properties of the *Customer* class are read-only, there is no other way to set the value of the *m_id* and *m_name* fields. If you inserted *cust.ID = 7* into the listing, the program would not compile, because *ID* is read-only; the same goes for *Name*. When the *ID* and *Name* properties are used in *Console.WriteLine*, they work fine. This is because these are read operations which only invoke the *get* accessor of the *ID* and *Name* properties.

One question you might have now is "If a property can be read-only, can it also be write-only?" The answer is yes, and explained in the next section.

Creating a Write-Only Property

You can assign values to, but not read from, a write-only property. A write-only property only has a *set* accessor. Listing 10-4 shows you how to create and use write-only properties.

Listing 10-4. Write-Only Properties

```

using System;

public class Customer
{
    private int m_id = -1;

    public int ID
    {
        set
        {
            m_id = value;
        }
    }

    private string m_name = string.Empty;

    public string Name
    {
        set
        {
            m_name = value;
        }
    }

    public void DisplayCustomerData()
    {
        Console.WriteLine("ID: {0}, Name: {1}", m_id, m_name);
    }
}

public class WriteOnlyCustomerManager
{
    public static void Main()
    {
        Customer cust = new Customer();

        cust.ID = 1;
        cust.Name = "Amelio Rosales";

        cust.DisplayCustomerData();

        Console.ReadKey();
    }
}

```

This time, the *get* accessor is removed from the *ID* and *Name* properties of the *Customer* class, shown in Listing 10-1. The *set* accessors have been added, assigning *value* to the backing store fields, *m_id* and *m_name*.

The *Main* method of the *WriteOnlyCustomerManager* class instantiates the *Customer* class with a default constructor. Then it uses the *ID* and *Name* properties of *cust* to set the *m_id* and *m_name* fields of *cust* to 1 and "Amelio Rosales", respectively. This invokes the *set* accessor of *ID* and *Name* properties from the *cust* instance.

When you have a lot of properties in a class or struct, there can also be a lot of code associated with those properties. In the next section, you'll see how to write properties with less code.

Creating Auto-Implemented Properties

The patterns you see here, where a property encapsulates a property with *get* and *set* accessors, without any other logic is common. It is more code than we should have to write for such a common scenario. That's why C# 3.0 introduced a new syntax for a property, called an *auto-implemented property*, which allows you to create properties without *get* and *set* accessor implementations. Listing 10-5 shows how to add auto-implemented properties to a class.

Listing 10-5. Auto-Implemented Properties

```
using System;

public class Customer
{
    public int ID { get; set; }
    public string Name { get; set; }
}

public class AutoImplementedCustomerManager
{
    static void Main()
    {
        Customer cust = new Customer();

        cust.ID = 1;
        cust.Name = "Amelio Rosales";

        Console.WriteLine(
            "ID: {0}, Name: {1}",
            cust.ID,
            cust.Name);

        Console.ReadKey();
    }
}
```

Notice how the *get* and *set* accessors in Listing 10-5 do not have implementations. In an auto-implemented property, the C# compiler creates the backing store field behind the scenes, giving the same logic that exists with traditional properties, but saving you from having to use all of the syntax of the traditional property. As you can see in the *Main* method, the usage of an auto-implemented property is exactly the same as traditional properties, which you learned about in previous sections.

Summary

You now know what properties are for and how they're used. Traditional techniques of encapsulation have relied on separate methods. Properties allow you to access objects state with field-like syntax. Properties can be made read-only or write-only. You also learned how to write properties with less code by using auto-implemented properties.

In C#, properties are nothing but natural extension of data fields. They are usually known as 'smart fields' in C# community. We know that data encapsulation and hiding are the two fundamental characteristics of any object oriented programming language. In C#, data encapsulation is possible through either classes or structures. By using various access modifiers like private, public, protected, internal etc it is possible to control the accessibility of the class members.

Usually inside a class, we declare a data field as private and will provide a set of public SET and GET methods to access the data fields. This is a good programming practice, since the data fields are not directly accessible outside the class. We must use the set/get methods to access the data fields.

An example, which uses a set of set/get methods, is shown below.

```
//SET/GET methods
//Author: rajeshvs@msn.com
using System;
class MyClass
{
    private int x;
    public void SetX(int i)
    {
        x = i;
    }
    public int GetX()
    {
        return x;
    }
}
class MyClient
{
    public static void Main()
    {
        MyClass mc = new MyClass();
        mc.SetX(10);
        int xVal = mc.GetX();
        Console.WriteLine(xVal);//Displays 10
    }
}
```

But C# provides a built in mechanism called properties to do the above. In C#, properties are defined using the property declaration syntax. The general form of declaring a property is as follows.


```

<access_modifier> <return_type> <property_name>
{
get
{
}
set
{
}
}

```

Where <access_modifier> can be private, public, protected or internal. The <return_type> can be any valid C# type. Note that the first part of the syntax looks quite similar to a field declaration and second part consists of a get accessor and a set accessor.

For example the above program can be modified with a property X as follows.

```

class MyClass
{
private int x;
public int X
{
get
{
return x;
}
set
{
x = value;
}
}
}

```

The object of the class MyClass can access the property X as follows.

```
MyClass mc = new MyClass();
```

mc.X = 10; // calls set accessor of the property X, and pass 10 as value of the standard field 'value'.

This is used for setting value for the data member x.

Console.WriteLine(mc.X); // displays 10. Calls the get accessor of the property X.

The complete program is shown below.

```

//C#: Property
//Author: rajeshvs@msn.com
using System;
class MyClass
{
private int x;
public int X
{

```

```

get
{
return x;
}
set
{
x = value;
}
}
}
class MyClient
{
public static void Main()
{
MyClass mc = new MyClass();
mc.X = 10;
int xVal = mc.X;
Console.WriteLine(xVal);//Displays 10
}
}

```

Remember that a property should have at least one accessor, either set or get. The set accessor has a free variable available in it called value, which gets created automatically by the compiler. We can't declare any variable with the name value inside the set accessor.

We can do very complicated calculations inside the set or get accessor. Even they can throw exceptions.

Since normal data fields and properties are stored in the same memory space, in C#, it is not possible to declare a field and property with the same name.

Static Properties

C# also supports static properties, which belongs to the class rather than to the objects of the class. All the rules applicable to a static member are applicable to static properties also.

The following program shows a class with a static property.

```

//C# : static Property
//Author: rajeshvs@msn.com
using System;
class MyClass
{
private static int x;
public static int X
{
get
{
return x;
}
}
}

```

```

set
{
x = value;
}
}
}
class MyClient
{
public static void Main()
{
MyClass.X = 10;
int xVal = MyClass.X;
Console.WriteLine(xVal);//Displays 10
}
}

```

Remember that set/get accessor of static property can access only other static members of the class. Also static properties are invoking by using the class name.

Properties & Inheritance

The properties of a Base class can be inherited to a Derived class.

```

//C# : Property : Inheritance
//Author: rajeshvs@msn.com
using System;
class Base
{
public int X
{
get
{
Console.Write("Base GET");
return 10;
}
set
{
Console.Write("Base SET");
}
}
}
class Derived : Base
{
}
class MyClient
{
public static void Main()
{
Derived d1 = new Derived();
d1.X = 10;
Console.WriteLine(d1.X);//Displays 'Base SET Base GET 10'
}
}

```

```
}
```

The above program is very straightforward. The inheritance of properties is just like inheritance any other member.

Properties & Polymorphism

A Base class property can be polymorphically overridden in a Derived class. But remember that the modifiers like virtual, override etc are using at property level, not at accessor level.

```
//C# : Property : Polymorphism
//Author: rajeshvs@msn.com
using System;
class Base
{
    public virtual int X
    {
        get
        {
            Console.Write("Base GET");
            return 10;
        }
        set
        {
            Console.Write("Base SET");
        }
    }
}
class Derived : Base
{
    public override int X
    {
        get
        {
            Console.Write("Derived GET");
            return 10;
        }
        set
        {
            Console.Write("Derived SET");
        }
    }
}
class MyClient
{
    public static void Main()
    {
        Base b1 = new Derived();
        b1.X = 10;
        Console.WriteLine(b1.X);//Displays 'Derived SET Derived GET 10'
    }
}
```

}

Abstract Properties

A property inside a class can be declared as abstract by using the keyword `abstract`. Remember that an abstract property in a class carries no code at all. The get/set accessors are simply represented with a semicolon. In the derived class we must implement both set and get assessors.

If the abstract class contains only set accessor, we can implement only set in the derived class.

The following program shows an abstract property in action.

```
//C# : Property : Abstract
//Author: rajeshvs@msn.com
using System;
abstract class Abstract
{
    public abstract int X
    {
        get;
        set;
    }
}
class Concrete : Abstract
{
    public override int X
    {
        get
        {
            Console.WriteLine(" GET");
            return 10;
        }
        set
        {
            Console.WriteLine(" SET");
        }
    }
}
class MyClient
{
    public static void Main()
    {
        Concrete c1 = new Concrete();
        c1.X = 10;
        Console.WriteLine(c1.X);//Displays 'SET GET 10'
    }
}
```

The properties are an important features added in language level inside C#. They are

very useful in GUI programming. Remember that the compiler actually generates the appropriate getter and setter methods when it parses the C# property syntax.

Indexers allow your class to be used just like an array. On the inside of a class, you manage a collection of values any way you want. These objects could be a finite set of class members, another array, or some complex data structure. Regardless of the internal implementation of the class, its data can be obtained consistently through the use of indexers. Here's an example.

Listing 11-1. An Example of An Indexer: IntIndexer.cs

```
using System;

/// <summary>
///   A simple indexer example.
/// </summary>
class IntIndexer
{
    private string[] myData;

    public IntIndexer(int size)
    {
        myData = new string[size];

        for (int i=0; i < size; i++)
        {
            myData[i] = "empty";
        }
    }

    public string this[int pos]
    {
        get
        {
            return myData[pos];
        }
        set
        {
            myData[pos] = value;
        }
    }
}

static void Main(string[] args)
{
    int size = 10;

    IntIndexer myInd = new IntIndexer(size);

    myInd[9] = "Some Value";
    myInd[3] = "Another Value";
    myInd[5] = "Any Value";
}
```

```

        Console.WriteLine("\nIndexer Output\n");

        for (int i=0; i < size; i++)
        {
            Console.WriteLine("myInd[{0}]: {1}", i, myInd[i]);
        }
    }
}

```

Listing 11-1 shows how to implement an Indexer. The *IntIndexer* class has a *string* array named *myData*. This is a private array that external users can't see. This array is initialized in the constructor, which accepts an *int size* parameter, instantiates the *myData* array, and then fills each element with the word "empty".

The next class member is the Indexer, which is identified by the *this* keyword and square brackets, *this[int pos]*. It accepts a single position parameter, *pos*. As you may have already guessed, the implementation of an Indexer is the same as a Property. It has *get* and *set* accessors that are used exactly like those in a Property. This indexer returns a *string*, as indicated by the *string* return value in the Indexer declaration.

The *Main()* method simply instantiates a new *IntIndexer* object, adds some values, and prints the results. Here's the output:

```

Indexer Output

myInd[0]: empty
myInd[1]: empty
myInd[2]: empty
myInd[3]: Another Value
myInd[4]: empty
myInd[5]: Any Value
myInd[6]: empty
myInd[7]: empty
myInd[8]: empty
myInd[9]: Some Value

```

Using an *integer* is a common means of accessing arrays in many languages, but the C# Indexer goes beyond this. Indexers can be declared with multiple parameters and each parameter may be a different type. Additional parameters are separated by commas, the same as a method parameter list. Valid parameter types for Indexers include *integers*, *enums*, and *strings*. Additionally, Indexers can be overloaded. In listing 11-2, we modify the previous program to accept overloaded Indexers that accept different types.

Listing 11-2. Overloaded Indexers: OvrIndexer.cs

```

using System;

/// <summary>
///     Implements overloaded indexers.
/// </summary>

```

```

class OvrIndexer
{
    private string[] myData;
    private int    arrSize;

    public OvrIndexer(int size)
    {
        arrSize = size;
        myData = new string[size];

        for (int i=0; i < size; i++)
        {
            myData[i] = "empty";
        }
    }

    public string this[int pos]
    {
        get
        {
            return myData[pos];
        }
        set
        {
            myData[pos] = value;
        }
    }

    public string this[string data]
    {
        get
        {
            int count = 0;

            for (int i=0; i < arrSize; i++)
            {
                if (myData[i] == data)
                {
                    count++;
                }
            }
            return count.ToString();
        }
        set
        {
            for (int i=0; i < arrSize; i++)
            {
                if (myData[i] == data)
                {
                    myData[i] = value;
                }
            }
        }
    }
}

```



```

static void Main(string[] args)
{
    int size = 10;
    OvrIndexer myInd = new OvrIndexer(size);

    myInd[9] = "Some Value";
    myInd[3] = "Another Value";
    myInd[5] = "Any Value";

    myInd["empty"] = "no value";

    Console.WriteLine("\nIndexer Output\n");

    for (int i=0; i < size; i++)
    {
        Console.WriteLine("myInd[{0}]: {1}", i, myInd[i]);
    }

    Console.WriteLine("\nNumber of \"no value\" entries: {0}", myInd["no value"]);
}
}

```

Listing 11-2 shows how to overload Indexers. The first Indexer, with the *int* parameter, *pos*, is the same as in Listing 11-1, but there is a new Indexer that takes a *string* parameter. The *get* accessor of the new indexer returns a *string* representation of the number of items that match the parameter value, *data*. The *set* accessor changes each entry in the array that matches the *data* parameter to the value that is assigned to the Indexer.

The behavior of the overloaded Indexer that takes a *string* parameter is demonstrated in the *Main()* method of Listing 11-2. It invokes the *set* accessor, which assigns the value of "no value" to every member of the *myInd* class that has the value of "empty". It uses the following command: *myInd["empty"] = "no value"*; After each entry of the *myInd* class is printed, a final entry is printed to the console, indicating the number of entries with the "no value" string. This happens by invoking the *get* accessor with the following code: *myInd["no value"]*. Here's the output:

Indexer Output

```

myInd[0]: no value
myInd[1]: no value
myInd[2]: no value
myInd[3]: Another Value
myInd[4]: no value
myInd[5]: Any Value
myInd[6]: no value
myInd[7]: no value
myInd[8]: no value
myInd[9]: Some Value

```

Number of "no value" entries: 7

The reason both Indexers in Listing 11-2 can coexist in the same class is because they have different signatures. An Indexer signature is specified by the number and type of parameters in an Indexers parameter list. The class will be smart enough to figure out which Indexer to invoke, based on the number and type of arguments in the Indexer call. An indexer with multiple parameters would be implemented something like this:

```
public object this[int param1, ..., int paramN]
{
    get
    {
        // process and return some class data
    }
    set
    {
        // process and assign some class data
    }
}
```

Summary

You now know what Indexers are for and how they're used. You can create an Indexer to access class members similar to arrays. Overloaded and multi-parameter Indexers were also covered.

INDEXER IN C#:

In c# introduce new concept is Indexer. This is very useful for some situation. Let as discuss something about Indexer.

- Indexer Concept is object act as an array.
- Indexer an object to be indexed in the same way as an array.
- Indexer modifier can be private, public, protected or internal.
- The return type can be any valid C# types.
- Indexers in C# must have at least one parameter. Else the compiler will generate a compilation error.

```
this [Parameter]
{
    get
    {
        // Get codes goes here
    }
    set
    {
        // Set codes goes here
    }
}
```

For Example:

```
using System;
```

```

using System.Collections.Generic;
using System.Text;

namespace Indexers
{
    class ParentClass
    {
        private string[] range = new string[5];
        public string this[int indexrange]
        {
            get
            {
                return range[indexrange];
            }
            set
            {
                range[indexrange] = value;
            }
        }
    }

    /* The Above Class just act as array declaration using this pointer */

    class childclass
    {
        public static void Main()
        {
            ParentClass obj = new ParentClass();

            /* The Above Class ParentClass create one object name is obj */

            obj[0] = "ONE";
            obj[1] = "TWO";
            obj[2] = "THREE";
            obj[3] = "FOUR ";
            obj[4] = "FIVE";
            Console.WriteLine("WELCOME TO C# CORNER HOME PAGE\n");
            Console.WriteLine("\n");

            Console.WriteLine("{0}\n,{1}\n,{2}\n,{3}\n,{4}\n", obj[0], obj[1], obj[2],
obj[3], obj[4]);
            Console.WriteLine("\n");
            Console.WriteLine("ALS.Senthur Ganesh Ram Kumar\n");
            Console.WriteLine("\n");
            Console.ReadLine();
        }
    }
}

```

BLOCK- 3

U n i t

10

Assembly & Attributes

1. INTRODUCTION

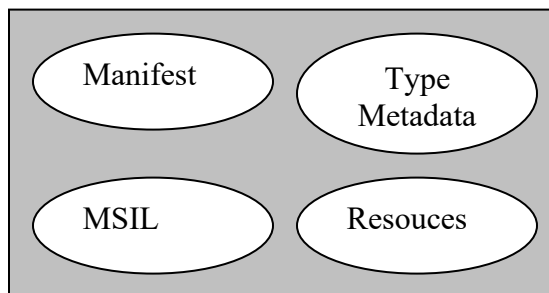
Advance Concepts of C# programming is consists of Assemblies and Attributes. This unit will provide information about assembly creation, internal access modifier, types of assemblies and also discuss how to create user define attributes, use of WIN32 API, Reflection and versioning.

2. ASSEMBLIES

Assemblies can be stated as “building block of .NET API (Application Programming Interfaces).” They are the collection of DLL (Dynamic Link Library) files, grouped as a single unit. Assemblies provide a means for reusing and sharing of any code in different APIs.

Each Assembly has a unique structure consists of four parts:-

- a) Manifest
- b) Type Metadata
- c) MSIL &
- d) Resources



Structure of an Assembly

Assembly manifest provides information about the assembly itself such as its version, culture etc. This is also called Assembly Metadata. The second part is type Metadata which describes the type of an assembly and the data (about methods, fields, properties and

events) contained in it. The third part MSIL is Microsoft Intermediate Language and contains the program code of assembly. The fourth part is the outside resources used in Assembly. These resources can be any outside file such as images, sound file or XML file etc.

2.1. CREATING ASSEMBLIES

a) Creating assembly DLL file from source files

We can create an assembly from a single .cs or multiple .cs files. C# Syntax for creating assembly from multiple file is-

```
D:\>csc /out:DLLfile_name /target:library <csfile1 csfile2 csfile3...>
```

For example to create assembly.dll from assembly1.cs and assembly2.cs files, we use-

```
D:\>csc /out : assembly.dll /target:library assembly1.cs assembly2.cs
```

b) Creating .exe file and referencing an assembly

The following command will create an executable file combining two or more C# files:-

```
D:\>csc /out:EXEfile_name /target:exe <csfile1 csfile2 csfile3...>
```

For example to create **assembly.exe** from **assembly1.cs** and **assembly2.cs** files, we use-

```
D:\>csc /out : assembly.exe /target:exe assembly1.cs assembly2.cs
```

Command to reference an assembly-

```
D:\>csc /out:EXEfile_name /target:exe /reference:<assembly1;assembly2;...;>  
<csfile1 csfile2 csfile3...>
```

For example to set reference of **assembly1.dll** in **assembly2.cs** file the following command is used:-

```
D:\>csc /out:assembly2.exe /target:exe /reference:assembly1.dll assembly2.cs
```

2.2. TYPES OF ASSEMBLIES

An assembly can be private or shared.

a) Private Assemblies:- When an assembly is created it is by default a private assembly. This assembly must be present at the same folder where its calling application existed. Private assemblies can have any user define name.

b) Shared Assemblies:- This is also called a global assembly. As the name suggests the shared assembly can be used by multiple applications at the same time. Shared assemblies must be installed in GAC(Global Assembly Cache) and have unique name. The following two steps are required to create a shared assembly:-

a. First we have to create a Strong name

b. Install assembly in GAC using GACUtil tool.

Strong Name works as a unique identifier for a shared assembly. To create a strong name .NET framework provides sn.exe tool. The following command is used to create a strong name:-

sn -k MyStrongName.snk

This will create a file named **MyStrongName.snk** & the strong name will be stored inside it.

Now to assign this strong name to an assembly, we should use AssemblyKeyFile attribute as

[assembly: AssemblyKeyFile("MyStrongName.snk")]

in our assembly program. This statement should be placed outside the namespace in the program. The following example elaborate the process:-

University.dll

```
using System;
using System.Reflection;
[assembly: AssemblyKeyFile("MyStrongName.snk")]

namespace University
{
    public class College
    {
        public void Course()
        {
            Console.WriteLine("This college provides various
Courses");
        }
        public void Faculty()
        {
            Console.WriteLine("Faculty Information");
        }
    }
}
```

Now to install an assembly in a Global Assembly Cache, we have to use GACUtil tool. This tool provides four options:-

- a) /l : Gives the total list of assemblies from GAC.
- b) /I : used to install an assembly in GAC.
- c) /u : uninstall a particular assembly from GAC.
- d) /upre : uninstall a particular assembly from Native Image Cache.

The following command is used to install an assembly:-

GACUtil /i University.dll

After installing an assembly in GAC, we can use it in any of the C# program without coping it in the same folder. We have to pass a reference to the shared assembly as

```
D:\>csc /r:University.dll Using_shared_assembly.cs
```

The following program uses methods created in a shared assembly

University.dll

Using_Shared_Assembly.cs

```
using System;
using University;
class A
{
    public static void main()
    {
        College cobj=new College();    // creating object of a class
                                        // inside University
        cobj.Faculty();                // using methods
        cobj.Course();
    }
}
```

2.3. ASSEMBLIES AND THE INTERNAL ACCESS MODIFIER

As we know that an access modifier is a keyword which is used to change the accessibility level of any member of a class such as variable, methods, properties etc. C# provides a new access modifier named “internal”. This modifier makes the member accessible upto assembly level i.e any member declared as internal will be available for any classes inside the same assembly only. In other words we can say that an internal modifier makes the member unavailable outside the assembly.

The following program displays the use of **internal** modifier:-

Assembly1.cs

```
using System;
using System.Reflection;

namespace namespace1
{
    public class Using_Internal
    {
        public String name;
    }
    public class Assembly1
    {
        public static void Main()
        {
            Using_Internal obj = new Using_Internal();
        }
    }
}
```

```

        obj.name = "Internal";                // available in same file
        Console.WriteLine("Internal variable name = " + obj.name);
    }
}
}

```

Compiling and executing this file:-

```

D:\assemblies and attributes>csc assembly1.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.30319.1
Copyright (C) Microsoft Corporation. All rights reserved.

D:\assemblies and attributes>assembly1
Internal variable name = Internal

D:\assemblies and attributes>_

```

When **assembly1.cs** file is converted into .dll file and we use the variable **name** in other program as:-

Assembly2.cs

```

using System;
using System.Reflection;
using namespace1;
public class Assembly2
{
    public static void Main()
    {
        Using_Internal obj = new Using_Internal();
        obj.name = "Internal";
        Console.WriteLine("Internal variable name = " + obj.name);
    }
}

```


On compiling and executing:

```
D:\assemblies and attributes>csc /out:assembly1.dll /target:library assembly1.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.30319.1
Copyright (C) Microsoft Corporation. All rights reserved.

D:\assemblies and attributes>csc /r:assembly1.dll assembly2.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.30319.1
Copyright (C) Microsoft Corporation. All rights reserved.

D:\assemblies and attributes>assembly2
Internal variable name = Internal
D:\assemblies and attributes>_
```

3. CUSTOMIZING AN ATTRIBUTES

The term attribute means the properties or characteristics. Attributes are the techniques to provide additional information about runtime behaviour of a method, class, structures or any other element of C# program. Or we can say that attributes are used to add Metadata to classes or assemblies.

The following example shows the use of Obsolete attribute:-

```
using System;
class attr
{
    [Obsolete()]
    public void method1()
    {
        Console.WriteLine("This is an obsolete method");
    }
    public void method2()
    {
        Console.WriteLine("Second method of a class");
    }
    public static void Main()
    {
        attr a1=new attr();
        a1.method1();
        a1.method2();
    }
}
```

This program consists of a class with two methods in which **method1()** is marked with obsolete attribute. This program shows a warning message when compiled that “method1 is obsolete”.

To provide our own message as a warning on compiling, we can customize this obsolete attribute as shown in following example:-

```

using System;
class attr
{
[Obsolete("Using user define message")]
public void method1()
{
Console.WriteLine("This is an obsolete method");
}
public void method2()
{
Console.WriteLine("Second method of a class");
}
public static void Main()
{
attr a1=new attr();
a1.method1();
a1.method2();

}
}

```



```

D:\assemblies and attributes>csc attr1.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.30319.1
Copyright (C) Microsoft Corporation. All rights reserved.

attr1.cs(16,1): warning CS0618: 'attr.method1()' is obsolete: 'Using user define
message'
D:\assemblies and attributes>

```

Now the class contain obsolete attribute with a string passed as an argument. This string works as a user define message in compile time.

4. IN-BUILT ATTRIBUTES

There are two attributes in C# .NET which are marked as in-built.

- a) Conditional Attribute
- b) DllImport Attribute

Here conditional attribute is used to prevent execution of any method of a class according to a certain condition and DllImport attribute is used to interact with some outside DLL files.

- **Conditional Attribute**

Conditional attribute prevents the execution of any method according to the definition of a preprocessor password or a code. It does not affect the code of any method but does affect the call of method. **System.Diagnostics** namespace provides the definition of conditional attribute. The method written with conditional

attribute is called only when the symbol used in conditional attribute as a string args is defined. This can be done in following two ways:-

➤ Using #define

The statement #define will be the first statement in a program and will define the password or symbol used in conditional attribute. The following example shows the process:-

```
#define code
using System;
using System.Diagnostics;
class conditional
{
    [Conditional("code")]
    public void sqr(int x)
    {
        Console.WriteLine("Square of {0} is = {1}",x,(x*x));
    }
    public static void Main()
    {
        conditional c=new conditional();
        c.sqr(10);
    }
}
```

Here a class is created with a single method marked with conditional attribute. This will give the following output:-

Square of 10 is = 100

➤ Using /define

In this method the element is marked as conditional and the symbol is defined at runtime using following command:-

D:\> csc /define: symbol csfile.cs

The following program shows the process:-

```
using System;
using System.Diagnostics;
class conditional
{
    [Conditional("code")]
    public void sqr(int x)
    {
        Console.WriteLine("Square of {0} is = {1}",x,(x*x));
    }
}
```

```

        public static void Main()
        {
            conditional c=new conditional();
            c.sqr(10);
        }
    }
}

```

Here a class is created with a single method marked with conditional attribute.

```

D:\assemblies and attributes>csc /define:code attr2.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.30319.1
Copyright (C) Microsoft Corporation. All rights reserved.

D:\assemblies and attributes>attr2
Square of 10 is = 100

D:\assemblies and attributes>_

```

- **DllImport Attribute**

This attribute provides the interoperability with windows DLLs. Window gives all its DLL files to .NET to import the predefined codes. DllImport attribute is used to call a method outside its managed application. The extern keyword is used before the function declaration. This outside code can be of C++, visual basic etc. **System.Runtime.InteropServices** namespace is used to work with this attribute and outside code.

The following program elaborates the process:-

```

using System;
using System.Runtime.InteropServices;
class inbuilt1
{
    [DllImport("User32.dll")]
    public static extern int MessageBox(int x, string
message, string caption, int type);

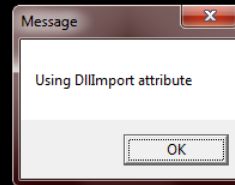
    public static void Main()
    {
        int msg=MessageBox(0,"Using DllImport attribute",
"Message", 1);
    }
}

```

Output of the programme is

```
D:\assemblies and attributes>csc attr4.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.30319.1
Copyright (C) Microsoft Corporation. All rights reserved.

D:\assemblies and attributes>attr4
```



This program display a message box with a custom message.

5. USING WIN32 API

6. CREATING CUSTOM ATTRIBUTES

User define attributes are subclasses of System.Attribute class. This subclass contains the implementation code for the attribute. AttributeUsage attribute is used to create custom attributes. This attribute has three properties:-

- AllowMultiple
- Inherited
- AttributeTargets

Syntax for AttributeUsage is:-

```
[AttributeUsage (AttributeTargets.Method | AttributeTargets.class |
AttributeTargets.Module | ..... ,AllowMultiple=true/
false,Inherited=true /false)]
public class classname : Attribute
{
/*class members*/
}
```

AllowMultiple property indicates that the attribute can be applied to multiple members at the same time. Second property **inherited** indicates that the attribute will be applied to inherited classes or not. And the third **AttributeTargets** uses the following members to describe that the attribute can be applied to which element:-

Members	Working
All	Attribute can be applied to any application element.
Assembly	Attribute can be applied to an assembly.
Class	Attribute can be applied to a class.
Constructor	Attribute can be applied to a constructor.
Delegate	Attribute can be applied to a delegate.
Enum	Attribute can be applied to an

Program demonstrate the process of creating User define Attribute:-

```
using System;  
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,Allow  
Multiple=false,Inherited=false)]
```

```
public class MyAttribute : Attribute  
{  
    private String msg;  
    public MyAttribute(String custom_msg)  
    {  
        msg=custom_msg;  
        Console.WriteLine("Main");  
    }  
}
```

```
class A  
{  
    [MyAttribute("Custom Attribute")]  
    public void display()  
    {  
        Console.WriteLine("MyAttribute Applied");  
    }  
}
```

```
[MyAttribute("Custom Attribute")]
```

```

class B
{
public void B_display();
{
Console.WriteLine("MyAttribute Applied to class");
}
}
public static void Main()
{
A a=new A();
a.display();
B b=new B();
b.B_display();

}
}

```

In this program an attribute is created names **MyAttribute**, which can only assigns to a class or a method. There are also two classes one of them is marked with **MyAttribute** and other class contains a method where **MyAttribute** is applied.

This attribute does not affect the output but you can check the metadata about the applied attribute by using ildasm tool as:-

D:> ILDasm Attributefile.exe

This command will display a window showing all the members of classes in attributefile.exe.

7. VERSIONING

As we know that we can create private as well as shared assemblies. In case of private assembly, it is identified by a unique name as this assembly should be available in the same location where it is used. When a shared assembly is created it is installed in GAC and two or more assemblies can have same name. to overcome this version number is provided. This works as a unique identifier or each of the assembly. Version number consists of four parts:-

- a) Major
- b) Minor
- c) Revision
- d) Build

Each parts are separated by dot(.) and represented by an integer greater than zero. Assembly version can be represented as,



We can assign a version number to an assembly by using *AssemblyVersion* attribute by adding the following line in an assembly,

[assembly: AssemblyVersion("major.minor.revision.buld")]

The following program shows the process:-

University.dll

```
using System;
using System.Reflection;
[assembly: AssemblyKeyFile("MyStrongName.snk")]
[assembly: AssemblyVersion("1.2.555.6")]

namespace University
{
    public class College
    {
        public void Course()
        {
            Console.WriteLine("This college provides various Courses");
        }
        public void Faculty()
        {
            Console.WriteLine("Faculty Information");
        }
    }
}
```

8. REFLECTION

Reflection is a mechanism which provides information about an object, class , methods , variables at runtime only. In other words we can say that reflection provides metadata about objects, classes or any member of a class. Type class of Reflection package is used to retrieve this information. It provides various methods and properties. Some of them are as follows:-

Method /Property	Provides
ConstructorInfo[]GetConstructors()	gives a list of the constructors for the specified type.
EventInfo[] GetEvents()	gives a list of events for the specified type.
FieldInfo[] GetFields()	gives a list of the fields for the specified type.
MemberInfo[] GetMembers()	gives a list of the members for the specified type.
MethodInfo[] GetMethods()	gives a list of methods for the specified type.

PropertyInfo[] GetProperties() bool IsAbstract	gives a list of properties for the specified type returns true if the specified type is abstract.
bool IsArray	returns true if the specified type is an array.
bool IsClass	returns true if the specified type is a class.
bool IsEnum	returns true if the specified type is an enumeration.

To retrieve information about a class or object using reflection, typeof() and GetType() methods are used. The following program elaborates the use:-

```
using System;
using System.Reflection;
namespace N
{
    public class reflection_methods
    {
        public void display()
        {
            String name;
            name=Console.ReadLine();
            Console.WriteLine("Welcome! {0}",name);
        }
        public static void Main()
        {
            reflection_methods rm=new reflection_methods();
            Type t1,t2;
            t1=typeof(reflection_methods);
            t2=rm.GetType();
            rm.display();
            Console.WriteLine(t1);
            Console.WriteLine(t2);
        }
    }
}
```

Output of the above program is as :-

```
D:\assemblies and attributes>csc ref.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.30319.1
Copyright (C) Microsoft Corporation. All rights reserved.

D:\assemblies and attributes>ref
Neha
Welcome! Neha
N.reflection_methods
N.reflection_methods

D:\assemblies and attributes>
```

The following program depicts the use of some other methods of Type class:-

```
using System;
using System.Reflection;
namespace N
{
    public class reflection_methods
    {
        String name;
        private int x;
        public int x_value
        {
            get
            {
                return x;
            }
            set
            {
                x=value;
            }
        }
        public reflection_methods()
        {
            Console.WriteLine("Constructor of class");
        }
        public reflection_methods(String s)
        {
            Console.WriteLine("Constructor of class");
        }
        public void display()
        {
            name=Console.ReadLine();
            Console.WriteLine("Welcome! {0}",name);
        }
    }
}
```

```

}
public static void Main()
{
    reflection_methods rm=new reflection_methods();
    rm.x_value=100;
    Type t1;
    t1=typeof(reflection_methods);
    Console.WriteLine("");
    Console.WriteLine("****Members of class****");

    MemberInfo [] m1=t1.GetMembers();
    ConstructorInfo [] c1=t1.GetConstructors();
    PropertyInfo [] p1=t1.GetProperties( );
    foreach(MemberInfo m in m1)
    {
        Console.WriteLine(m);
    }
    Console.WriteLine("");
    Console.WriteLine("****Constructors of class****");
    foreach(ConstructorInfo c in c1)
    {
        Console.WriteLine(c);
    }
    Console.WriteLine("");
    Console.WriteLine("****Properties of class****");
    foreach(PropertyInfo p in p1)
    {
        Console.WriteLine(p);
    }
}
}
}
}
}

```

Output of the program:-

```

D:\assemblies and attributes>ref
Constructor of class

****Members of class****
Int32 get_x_value()
Void set_x_value(Int32)
Void display()
Void Main()
System.String ToString()
Boolean Equals(System.Object)
Int32 GetHashCode()
System.Type GetType()
Void .ctor()
Void .ctor(System.String)
Int32 x_value

****Constructors of class****
Void .ctor()
Void .ctor(System.String)

****Properties of class****
Int32 x_value

D:\assemblies and attributes>_

```

9. SUMMARY

10. EXERCISE

- a) Explain Assemblies in C#. Create a shared assembly named Arithmetic.dll. This should contain appropriate methods for arithmetical operations.
- b) Fill in the blanks:-
 - a. GAC stands for _____.
 - b. Version number consists of _____, _____, _____ & _____.
 - c. _____ namespace is used to create conditional attribute.
 - d. Three properties of AttributeUsage are _____, _____ & _____.
- c) Create an assembly and assign it version 1.3.025.5.
- d) Define Attributes and write a programme to display user define message as a warning in compile time.
- e) Create a program with conditional attribute and use both methods of calling function marked with conditional.
- f) Create a user define attribute named Student, which can only be applied to classes and its subclasses.

Directive and Debugging

Session 11: Directive and Debugging

Introduction

Error

Types of Errors

Finding Errors

Preprocessor Directive

Using Debuggers

Error

A wandering; a roving or irregular course.

A wandering or deviation from the right course or standard; irregularity; mistake; inaccuracy; something made wrong or left wrong; as, an error in writing or in printing; a clerical error.

A departing or deviation from the truth; falsity; false notion; wrong opinion; mistake; misapprehension.

A moral offense; violation of duty; a sin or transgression; iniquity; fault.

The difference between the approximate result and the true result; -- used particularly in the rule of double position.

The difference between an observed value and the true value of a quantity.

The difference between the observed value of a quantity and that which is taken or computed to be the true value; -- sometimes called residual error.

A mistake in the proceedings of a court of record in matters of law or of fact.

A fault of a player of the side in the field which results in failure to put out a player on the other side, or gives him an unearned base.

This section discusses the C# language's preprocessor directives:

#if

#else

#elif

#endif

#define

#undef

#warning

#error

#line

#region

#endregion

#pragma

#pragma warning

#pragma checksum

While the compiler does not have a separate preprocessor, the directives described in this section are processed as if there was one; these directives are used to aid in conditional compilation. Unlike C and C++ directives, you cannot use these directives to create macros.

A preprocessor directive must be the only instruction on a line.

Debugging refers to the process of trying to track down errors in your programmes. It can also refer to handling potential errors that may occur. There are three types of errors that we'll take a look at:

- Design-time errors
- Run-Time errors
- Logical errors

The longer your code gets, the harder it is to track down why things are not working. By the end of this section, you should have a good idea of where to start looking for problems. But bear in mind that debugging can be an art in itself, and it gets easier with practice.

Errors at Design-Time

Design-Time errors are ones that you make before the programme even runs. In fact, for Design-Time errors, the programme won't run at all, most of the time. You'll get a popup message telling you that there were build errors, and asking would you like to continue.

Design-Time errors are easy enough to spot because the C# software will underline them with a wavy coloured line. You'll see three different coloured lines: blue, red and green. The blue wavy lines are known as **Edit and Continue** issues, meaning that you can make change to your code without having to stop the programme altogether. Red wavy lines are **Syntax** errors, such as a missing semicolon at the end of a line, or a missing curly bracket in an IF Statement. Green wavy lines are **Compiler Warnings**. You get these when C# spots something that could potentially cause a problem, such as declaring a variable that's never used.

is article describes how to use the **Debug** and the **Trace** classes. These classes are available in the Microsoft .NET Framework. You can use these classes to provide information about the performance of an application either during application development, or after deployment to production. These classes are only one part of the instrumentation features that are available in the .NET Framework.

Requirements

The following list outlines the recommended hardware, software, network infrastructure, and service packs that you need:

- Microsoft Windows 2000 or Microsoft Windows XP or Microsoft Windows Server 2003
- Microsoft Visual C#

This article also assumes that you are familiar with program debugging.

Description Of Technique

The steps in the Create a Sample with the Debug Class section demonstrate how to create a console application that uses the **Debug** class to provide information about the program

execution.

When the program is run, you can use methods of the **Debug** class to produce messages that help you to monitor the program execution sequence, to detect malfunctions, or to provide performance measurement information. By default, the messages that the **Debug** class produces appear in the Output window of the Visual Studio Integrated Development Environment (IDE).

The sample code uses the **WriteLine** method to produce a message that is followed by a line terminator. When you use this method to produce a message, each message appears on a separate line in the Output window.

When you use the **Assert** method of the **Debug** class, the Output window displays a message only if a specified condition evaluates to false. The message also appears in a modal dialog box to the user. The dialog box includes the message, the project name, and the **Debug.Assert** statement number. The dialog box also includes the following three command buttons:

- **Abort:** The application stops running.
- **Retry:** The application enters debug mode.
- **Ignore:** The application proceeds.

The user must click one of these buttons before the application can continue.

You can also direct output from the **Debug** class to destinations other than the Output window. The **Debug** class has a collection named **Listeners** that includes **Listener** objects.

Each **Listener** object monitors **Debug** output and directs the output to a specified target.

Each **Listener** in the **Listener** collection receives any output that the **Debug** class generates. Use the **TextWriterTraceListener** class to define **Listener** objects. You can specify the target for a **TextWriterTraceListener** class through its constructor.

Some possible output targets include the following:

- The Console window by using the **System.Console.Out** property.
- A text (.txt) file by using the **System.IO.File.CreateText("FileName.txt")** statement.

After you create a **TextWriterTraceListener** object, you must add the object to the **Debug.Listeners** collection to receive Debug output.

Create a Sample with the Debug Class

1. Start Visual Studio or Visual C# Express Edition.
2. Create a new Visual C# Console Application project named conInfo. Class1 is created in Visual Studio .NET. Program.cs is created in Visual Studio 2005.
3. Add the following namespace at top in Class1 or Program.cs.

```
using System.Diagnostics;
```

4. To initialize variables to contain information about a product, add the following declaration statements to **Main** method:
 5. `string sProdName = "Widget";`
 6. `int iUnitQty = 100;`
`double dUnitCost = 1.03;`
7. Specify the message that the class produces as the first input parameter of the **WriteLine** method. Press the CTRL+ALT+O key combination to make sure that the Output window is visible.

```
Debug.WriteLine("Debug Information-Product Starting ");
```

8. For readability, use the **Indent** method to indent subsequent messages in the Output window:

```
Debug.Indent();
```

9. To display the content of selected variables, use the **WriteLine** method as follows:
 10. `Debug.WriteLine("The product name is " + sProdName);`
 11. `Debug.WriteLine("The available units on hand are" +
iUnitQty.ToString());`
`Debug.WriteLine("The per unit cost is " + dUnitCost.ToString());`
12. You can also use the **WriteLine** method to display the namespace and the class name for an existent object. For example, the following code displays the **System.Xml.XmlDocument** namespace in the Output window:
 13. `System.Xml.XmlDocument oxml = new System.Xml.XmlDocument();`
`Debug.WriteLine(oxml);`
14. To organize the output, you can include a category as an optional, second input parameter of the **WriteLine** method. If you specify a category, the format of the Output window message is "category: message." For example, the first line of the following code displays "Field: The product name is Widget" in the Output window:
 15. `Debug.WriteLine("The product name is " + sProdName,"Field");`
 16. `Debug.WriteLine("The units on hand are" + iUnitQty,"Field");`
 17. `Debug.WriteLine("The per unit cost is" +
dUnitCost.ToString(),"Field");`

```
Debug.WriteLine("Total Cost is " + (iUnitQty *  
dUnitCost), "Calc");
```

18. The Output window can display messages only if a designated condition evaluates to true by using the **WriteLineIf** method of the **Debug** class. The condition to be evaluated is the first input parameter of the **WriteLineIf** method. The second parameter of **WriteLineIf** is the message that appears only if the condition in the first parameter evaluates to true.

```
19. Debug.WriteLineIf(iUnitQty > 50, "This message WILL appear");
```

```
20. Debug.WriteLineIf(iUnitQty < 50, "This message will NOT  
appear");
```

21. Use the **Assert** method of the **Debug** class so that the Output window displays the message only if a specified condition evaluates to false:

```
22. Debug.Assert(dUnitCost > 1, "Message will NOT appear");
```

```
23. Debug.Assert(dUnitCost < 1, "Message will appear since dUnitcost  
< 1 is false");
```

24. Create the **TextWriterTraceListener** objects for the Console window (tr1) and for a text file named Output.txt (tr2), and then add each object to the **Debug Listeners** collection:

```
25. TextWriterTraceListener tr1 = new  
    TextWriterTraceListener(System.Console.Out);
```

```
26. Debug.Listeners.Add(tr1);
```

```
27.
```

```
28. TextWriterTraceListener tr2 = new  
    TextWriterTraceListener(System.IO.File.CreateText("Output.txt"));  
    Debug.Listeners.Add(tr2);
```

29. For readability, use the **Unindent** method to remove the indentation for subsequent messages that the **Debug** class generates. When you use the **Indent** and the **Unindent** methods together, the reader can distinguish the output as group.

```
30. Debug.Unindent();
```

```
    Debug.WriteLine("Debug Information-Product Ending");
```

31. To make sure that each **Listener** object receives all its output, call the **Flush** method for the **Debug** class buffers:

```
Debug.Flush();
```

Using the Trace Class

You can also use the **Trace** class to produce messages that monitor the execution of an application. The **Trace** and **Debug** classes share most of the same methods to produce output, including the following:

- **WriteLine**
- **WriteLineIf**
- **Indent**
- **Unindent**
- **Assert**

- **Flush**

You can use the **Trace** and the **Debug** classes separately or together in the same application. In a Debug Solution Configuration project, both **Trace** and **Debug** output are active. The project generates output from both of these classes to all **Listener** objects. However, a Release Solution Configuration project only generates output from a **Trace** class. The Release Solution Configuration project ignores any **Debug** class method invocations.

```
Trace.WriteLine("Trace Information-Product Starting ");
Trace.Indent();

Trace.WriteLine("The product name is "+sProdName);
Trace.WriteLine("The product name is"+sProdName,"Field" );
Trace.WriteLineIf(iUnitQty > 50, "This message WILL appear");
Trace.Assert(dUnitCost > 1, "Message will NOT appear");

Trace.Unindent();
Trace.WriteLine("Trace Information-Product Ending");

Trace.Flush();

Console.ReadLine();
```

Verify That It Works

1. Make sure that **Debug** is the current solution configuration.
2. If the **Solution Explorer** window is not visible, press the CTRL+ALT+L key combination to display this window.
3. Right-click **conInfo**, and then click **Properties**.
4. In the left pane of the conInfo property page, under the **Configuration** folder, make sure that the arrow points to **Debugging**.

Note In Visual C# 2005 and in Visual C# 2005 Express Edition, click **Debug** in the **conInfo** page.

5. Above the **Configuration** folder, in the **Configuration** drop-down list box, click **Active (Debug)** or **Debug**, and then click **OK**. In Visual C# 2005 and in Visual C# 2005 Express Edition, click **Active (Debug)** or **Debug** in the **Configuration** drop-down list box in the **Debug** page, and then click **Save** on the **File** menu.
6. Press CTRL+ALT+O to display the Output window.
7. Press the F5 key to run the code. When the **Assertion Failed** dialog box appears, click **Ignore**.
8. In the Console window, press ENTER. The program should finish, and the Output window should display the output that resembles the following
9. Debug Information-Product Starting
10. The product name is Widget
11. The available units on hand are100
12. The per unit cost is 1.03
13. System.Xml.XmlDocument
14. Field: The product name is Widget
15. Field: The units on hand are100

```

16.      Field: The per unit cost is1.03
17.      Calc: Total Cost is  103
18.      This message WILL appear
19.      ---- DEBUG ASSERTION FAILED ----
20. ---- Assert Short Message ----
21. Message will appear since dUnitcost  < 1 is false
22. ---- Assert Long Message ----
23.
24.
25.      at Class1.Main(String[] args)  <%Path%>\class1.cs(34)
26.
27.      The product name is Widget
28.      The available units on hand are100
29.      The per unit cost is 1.03
30. Debug Information-Product Ending
31. Trace Information-Product Starting
32.      The product name is Widget
33.      Field: The product name isWidget
34.      This message WILL appear
35. Trace Information-Product Ending
36.
37. The Console window and the Output.txt file should display the following output:
38. The product name is Widget
39.      The available units on hand are 100
40.      The per unit cost is 1.03
41. Debug Information-Product Ending
42. Trace Information-Product Starting
43.      The product name is Widget
44.      Field: The product name is Widget
45.      This message WILL appear
46. Trace Information-Product Ending

```

Note The Output.txt file is located in the same directory as the conInfo executable (conInfo.exe). Typically, this is the \bin folder where the project source is stored. By default, this is C:\Documents and Settings*User login*\My Documents\Visual Studio Projects\conInfo\bin. In Visual C# 2005 and in Visual C# 2005 Express Edition, the Output.txt file is located in the following folder:

C:\Documents and Settings*User login*\My Documents\Visual Studio
2005\Projects\conInfo\conInfo\bin\Debug

Complete Code Listing

```

using System;
using System.Diagnostics;

class Class1
{
    [STAThread]
    static void Main(string[] args)
    {
        string sProdName = "Widget";
        int iUnitQty = 100;
        double dUnitCost = 1.03;
    }
}

```

```

        Debug.WriteLine("Debug Information-Product Starting ");
        Debug.Indent();
        Debug.WriteLine("The product name is "+sProdName);
        Debug.WriteLine("The available units on hand
are"+iUnitQty.ToString());
        Debug.WriteLine("The per unit cost is "+
dUnitCost.ToString());

        System.Xml.XmlDocument oxml = new System.Xml.XmlDocument();
        Debug.WriteLine(oxml);

        Debug.WriteLine("The product name is "+sProdName,"Field");
        Debug.WriteLine("The units on hand are"+iUnitQty,"Field");
        Debug.WriteLine("The per unit cost
is"+dUnitCost.ToString(),"Field");
        Debug.WriteLine("Total Cost is  "+(iUnitQty *
dUnitCost),"Calc");

        Debug.WriteLineIf(iUnitQty > 50, "This message WILL appear");
        Debug.WriteLineIf(iUnitQty < 50, "This message will NOT
appear");

        Debug.Assert(dUnitCost > 1, "Message will NOT appear");
        Debug.Assert(dUnitCost < 1, "Message will appear since
dUnitcost < 1 is false");

        TextWriterTraceListener tr1 = new
TextWriterTraceListener(System.Console.Out);
        Debug.Listeners.Add(tr1);

        TextWriterTraceListener tr2 = new
TextWriterTraceListener(System.IO.File.CreateText("Output.txt"));
        Debug.Listeners.Add(tr2);

        Debug.WriteLine("The product name is "+sProdName);
        Debug.WriteLine("The available units on hand are"+iUnitQty);
        Debug.WriteLine("The per unit cost is "+dUnitCost);
        Debug.Unindent();
        Debug.WriteLine("Debug Information-Product Ending");
        Debug.Flush();

        Trace.WriteLine("Trace Information-Product Starting ");
        Trace.Indent();

        Trace.WriteLine("The product name is "+sProdName);
        Trace.WriteLine("The product name is"+sProdName,"Field" );
        Trace.WriteLineIf(iUnitQty > 50, "This message WILL appear");
        Trace.Assert(dUnitCost > 1, "Message will NOT appear");

        Trace.Unindent();
        Trace.WriteLine("Trace Information-Product Ending");

        Trace.Flush();

        Console.ReadLine();

```

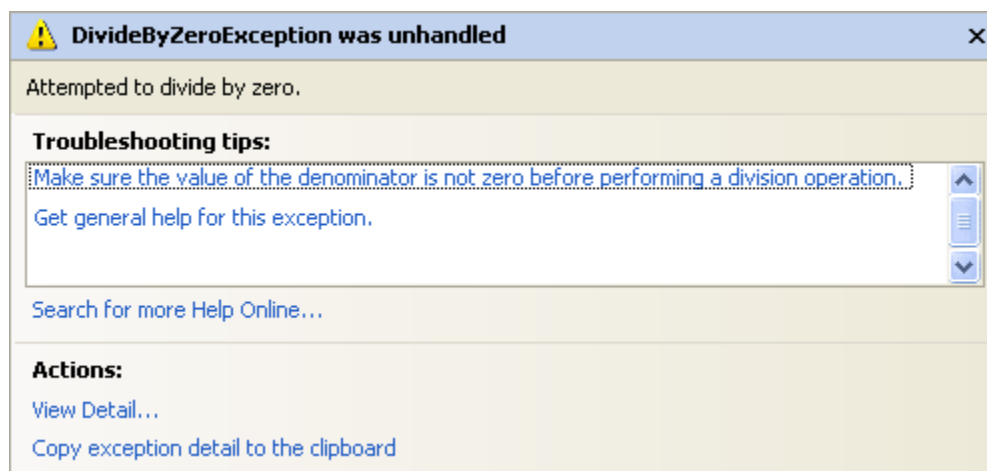
```
}  
}
```

Run Time Errors in C# .NET

Run-Time errors are ones that crash your programme. The programme itself generally starts up OK. It's when you try to do something that the error surfaces. A common Run-Time error is trying to divide by zero. In the code below, we're trying to do just that:

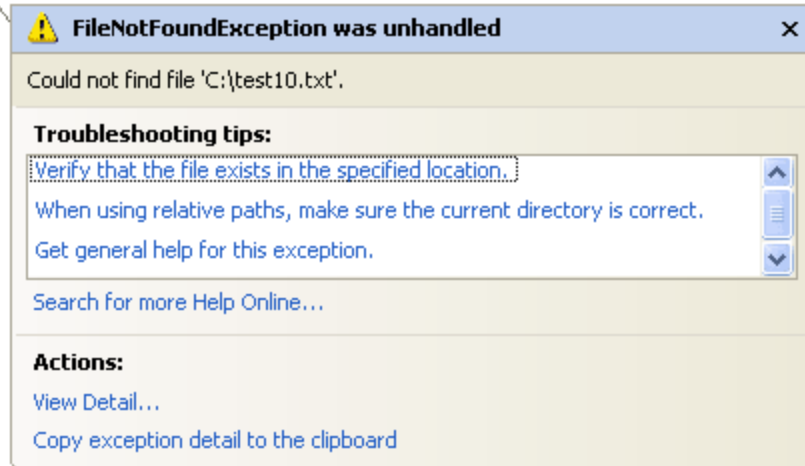
```
private void button1_Click(object sender, EventArgs e)  
{  
  
    int Num1 = 10;  
    int Num2 = 0;  
    int answer;  
  
    answer = Num1 / Num2;  
  
}
```

The programme itself reports no problems when it is started up, and there's no coloured wavy lines. When we click the button, however, we get the following error message:



Had we left this in a real programme, it would just crash altogether ("bug out"). But if you see any error message like this one, it's usually a Run-Time error. Here's another one. In the code below, we're trying to open a file that doesn't exist:

```
private void button1_Click(object sender, EventArgs e)
{
    rt1.LoadFile("C:/test10.txt", RichTextBoxStreamType.PlainText);
}
```



As the message explains, it can't find the file called "C:/test10.txt". Because we didn't tell C# what to do if there was no such file, it just crashes.

Look out for these type of error messages. It does take a bit of experience to work out what they mean; but some, like the one above, are quite straightforward.

You'll see how to handle errors like this, soon. But there's one final error type you have to know about - Logic Errors.

Logic Errors in C# .NET

Logic errors are ones where you don't get the result you were expecting. You won't see any coloured wavy lines, and the programme generally won't "bug out" on you. In other words, you've made an error in your programming logic. As an example, take a look at the following code, which is attempting to add up the numbers one to ten:

```

private void button1_Click(object sender, EventArgs e)
{
    int startLoop = 11;
    int endLoop = 1;
    int answer = 0;

    for (int i = startLoop; i < endLoop; i++)
    {
        answer = answer + i;
    }

    MessageBox.Show("answer =" + answer.ToString());
}

```

When the code is run, however, it gives an answer of zero. The programme runs OK, and didn't produce any error message or wavy lines. It's just not the correct answer!

The problem is that we've made an error in our logic. The **startLoop** variable should be 1 and the **endLoop** variable 11. We've got it the other way round, in the code. So the loop never executes.

Logic errors can be very difficult to track down. To help you find where the problem is, C# has some very useful tools you can use. To demonstrate these tools, here's a new programming problem. We're trying to write a programme that counts how many times the letter "g" appears in the word "debugging".

Start a new C# Windows Application. Add a button and a textbox to your form. Double click the button, and then add the following code:

```

private void button1_Click(object sender, EventArgs e)
{
    int LetterCount = 0;
    string strText = "Debugging";
    string letter;

    for (int i = 0; i < strText.Length; i++)
    {
        letter = strText.Substring(i, 1);

        if (letter == "g")
        {
            LetterCount++;
        }
    }

    textBox1.Text = "g appears " + LetterCount + " times";
}

```


The answer should, of course, be 3. Our programme insists, however, that the answer is zero. It's telling us that there aren't any g's in Debugging. So we have made a logic error, but where?

C# .NET has some tools to help you track down errors like this. The first one we'll look at is called the BreakPoint.

C# features "preprocessor directives" (though it does not have an actual preprocessor) based on the C preprocessor that allow programmers to define symbols but not macros. Conditionals such as `#if`, `#endif`, and `#else` are also provided. Directives such as `#region` give hints to editors for code folding.

```
public class Foo
{
    #region Procedures
    public void IntBar(int firstParam) {}
    public void StrBar(string firstParam) {}
    public void BoolBar(bool firstParam) {}
    #endregion

    #region Constructors
    public Foo() {}
    public Foo(int firstParam) {}
    #endregion
}
```

12

Exception Handling

C# Preprocessor Directives Explained

By using pre-processor directives you can exclude parts of your code from being seen by the compiler. Excluded from the assembly they are never seen at run-time. This is different from a regular *if (x) {}* block where code is actually compiled in, evaluated at runtime and added to the assembly.

To understand why you would want to do this, a little history: One of the good things about C is that it works on every imaginable platform, the bad thing was of course that it works on every imaginable platform. There was always a little tweaking required to get your code to compile. Your program might have needed to compile on Amiga, DOS , OS/2, Windows or Linux. The invention of the preprocessor made this much easier. As a separate step prior to compilation it combs through the source code and modifies it according to the pre-processing instructions found in the source code. The C compiler never gets to see the things that don't apply to it.

In C# there is no separate preprocessor step instead the compiler itself skims through the code as it reads it. Fortunately it is also nowhere near as complex as the C pre-processor.

Contents

- [The #define and #undef directives](#)
- [Unlike C and C++ there are no macros in C#](#)
- [Conditional Compilation with #if / #else / #endif](#)
- [Avoid the clutter – use conditional attributes instead](#)
- [Using #error and #warning during compilation](#)
- [Generated code builds and error reporting with #line](#)
- [Region Directives](#)

The #define and #undef directives

The basis of the C# preprocessor directives are `#define` and `#undef`. They allow you to create and destroy symbols used by the preprocessing step.

```
#define TESTVERSION
#undef DEBUG
```

A common use is for a programmer to turn on and off the inclusion of debugging code. When you are coding and debugging this allow you to test your code but this code shouldn't be present in the production version of your software.

It is possible to set symbols using `/define` on the compilers command line allowing you to build scripts that produce different versions of your code. For example a "full version" and a limited "demo" version.

C# does not have any pre-defined symbols, except for maybe DEBUG

If you are used to your C/C++ compiler [setting large numbers of symbols](#) so that you could identify the compiler and version you will be puzzled to discover the C# doesn't have any. So you cannot check for the .NET version from a pre-defined symbol and branch your code based on that. (The closest that you can do is to check for the execution environment at runtime, for example if you want to know [if you are running under mono.](#))

If you are compiling your code in the Visual C# / MonoDevelop environment there will be one symbol defined you can use: `DEBUG`. This is set in the project options and if you are building your code you can select if you want to build the Debug or Release version.

Unlike C and C++ there are no macros in C#

The following is not going to fly on C#. Macro's are commonly used to expand expressions while compiling C code. A common problem is that C doesn't have a boolean type. So the first thing most programmers do when starting a new project is to define their own as shown in the next little C code snippet:

[view sourceprint?](#)

```
01.#include <stdio.h>
02.
03.#define BOOL    int
04.#define TRUE    1
05.#define FALSE   0
06.#define IS_TRUE(x)  (x == TRUE)
07.
08.void main()
09.{
10.if (IS_TRUE(TRUE))
11.{
12.printf("It is TRUE!");
13.}
14.}
```

When the C preprocessor comes across this it would replace “BOOL” with int, FALSE with 0 and TRUE with 1. The compiler never saw the BOOL, or the TRUE or FALSE. But lets not hang around here — as said, this is not supported by C#

Conditional Compilation with #if / #else / #endif

Of course, if you can define symbols you need to be able to test for them as well. The #if / #else / #endif directives do just that. If you are building the debug version of your code, you can check for the DEBUG symbol:

```
#if DEBUG
Console.WriteLine(“Things are not going so well!”);
#endif
```

If you are not building the Debug version of your code — the above line is never seen by the compiler, and thus is not included in the assembly. All #if statements need to be closed by an #endif, and of course you can also use an #else statement:

```
#if DEBUG
Console.WriteLine(“Things are not going so well, some more debugging is needed!”);
#else
Console.WriteLine(“Fatal Error: Please call free support (0800) 123 123”);
#endif
```

The #*elif* directive is a short form for “#else #if #endif”, it works the same but you can save some space as you do not need to end with another #endif. If your project has a more complicated release schedule with alpha, beta and production releases you could try this:

```
#if (!RELEASE)
Console.WriteLine(“This is not a release version”);
#endif
#if (BETA)
Console.WriteLine(“Beta, for limited release only”);
#endif
#elif (ALPHA)
Console.WriteLine(“Alpha, for internal testing only”);
#endif
#else
Console.WriteLine(“Welcome to Widgets 1.0”);
#endif
```

As can be seen you can apply several operators to the symbols: ! (not), == (equality), != (inequality), && (and) and || (or).

Avoid the clutter – use conditional attributes instead

Even a small program has many potential spots where you might like to introduce a debug statement or a log function. If you have to put each of them into a separate `#if DEBUG / #endif` block they will start to take up a lot of space in your code. C# has its own slightly more elegant solution to this problem: conditional attributes. They are included in the `System.Diagnostics` namespace.

In the following program the “LogLine” function will only ever run if the “DEBUG” symbol has been set.

[view sourceprint?](#)

```
01.using System;
02.using System.Diagnostics;
03.
04.namespace MyNameSpace
05.{
06.class MainClass
07.{
08.[ Conditional("DEBUG") ]
09.public static void LogLine(string msg,string detail)
10.{
11.Console.WriteLine("Log: {0} = {1}",msg,detail);
12.}
13.
14.public static void Main(string[] args)
15.{
16.int Total = 0;
17.for(int Lp = 1; Lp < 10; Lp++)
18.{
19.LogLine("Total",Total.ToString());
20.Total = Total + Lp;
21.}
22.}
23.}
24.}
```

You can also define combinations chained together with OR to decide if the code is enabled:

```
[ Conditional("ALPHA"),Conditional("BETA") ]
public static void LogLine(string msg,string detail)
```

Conditionals work differently from the `#if/#endif` directives that in the above example the “LogLine” code will still be included in the assembly. It will never get called however. The compiler will ensure that the LogLine method isn’t called. In fact if you use a disassembler (like `monodis`) the code for Main will look like this:

[view sourceprint?](#)

```
1.public static void Main(string[] args)
2.{
3.int Total = 0;
4.for(int Lp = 1; Lp < 10; Lp++)
```

```
5.Total = Total + Lp;  
6.}
```

Using #error and #warning during compilation

C# provides the #error and #warning directives to output messages during compilation. As you can expect, if the compiler encounters an #error it will report the problem and stop compiling. The #warning directive just causes a log entry to be displayed in the build output.

```
#if !DEBUG  
#warning This code is NOT production ready  
#endif
```

Generated code builds and error reporting with #line

In regular coding there are not many uses for #line. This directive is mostly useful when you build a code generator that turns one source file into another. Microsoft uses this for example for ASP.NET. Normally if something breaks during such an intermediate step, the compiler would report the error as being in the intermediate file.

The #line directive can force the compiler to report the error as coming from a different line number, or even a completely different source file.

[view sourceprint?](#)

```
1.public static void Main(string[] args)  
2.{  
3.  
4.#line 250  
5.#if DEBUG  
6.#error This code is NOT production ready  
7.#endif
```

Even though in my original test code the warning was given at line 19 in the source file, the #line directive forces the compiler to report 250 instead.

Region Directives

The Visual Studio suite, and the newer MonoDevelop versions allow for outlining of code blocks. You click the little (+) next to the code and it expands the class or method. The #region and #endregion directives allow you to define your own outlined blocks.

[view sourceprint?](#)

```
01.#region myregion  
02.public static void Main(string[] args)  
03.{  
04.int Total = 0;  
05.for(int Lp = 1; Lp < 10; Lp++)  
06.{
```

```
07.LogLine("Total",Total.ToString());
08.Total = Total + Lp;
09.}
10.}
11.#endregion
```

C# preprocessor is fundamentally very similar to C preprocessor and the whole concept in C# has been taken from C language specification.

"The C preprocessor is a **macro processor** that is used automatically by the C compiler to transform your program before actual compilation. It is called a macro processor because it allows you to define macros, which are brief abbreviations for longer constructs."

But in C# only concept has been taken from C. But the C# compiler does not have a separate preprocessor, the directives described in this C# are processed as if there was one. Unlike C and C++ directives, you cannot use these directives to create macros.

A preprocessor directive must be the only instruction on a line. Preprocessing directives are lines in your program that start with '#'. Whitespace is allowed before and after the '#'. The '#' is followed by an identifier that is the directive name. For example, '#define' is the directive

The C# language's preprocessor directives are as follows

- #if
- #else
- #elif
- #endif
- #define
- #undef
- #warning
- #error
- #line
- #region
- #endregion

Main use of directives are

1. **Conditional compilation.** Using special preprocessing directives, you can include or exclude parts of the program according to various conditions.
2. **Line control.** If you use a program to combine or rearrange source files into an intermediate file, which is then compiled, you can use line control to inform the compiler of where each source line originally came from.

3. **Error and Warning reporting:** The directive '#error' causes the preprocessor to report a fatal error and the directive '#warning' is like the directive '#error', but causes the preprocessor to issue a warning and continue preprocessing.

Region and Unregion is new directives. It was not in C and C++ list of directives. I don't know the intention of C# developers to excludes number of directive from C and C++ list but they picked what people use atmost from list of C and C++ directive. Before telling meaning of each preprocessor directive I want to explain how to define preprocessor directive. There are two method to define directive.

1. Define in your C# program.

2. Define them at command line on compile time.

Here is example for first way

Example

```
#define TEST
using System;
public class MyClass
{
    public static void Main()
    {
        #if (TEST)
        Console.WriteLine("TEST is defined");
        #else
        Console.WriteLine("TEST is not defined");
        #endif
    }
}
```

Output

TEST is defined

In other way you can define it at command line. So program will be like this.

Example

```
using System;
public class MyClass
{
    public static void Main()
    {
        #if (TEST)
        Console.WriteLine("TEST is defined");
        #else
        Console.WriteLine("TEST is not defined");
        #endif
    }
}
```



```
}
```

At compile time user can define as below

```
csc /define:TEST MyClass.java
```

Output

TEST is defined

And if the command line will be like

```
csc MyClass.java
```

Output

TEST is not defined

Now its time to explain about various preprocessor define.

#if directive

The '#if' directive in its simplest form consists of

```
#if expression  
controlled text  
#endif /* expression */
```

The comment following the '#endif' is not required, but it is a good practice because it helps people match the '#endif' to the corresponding '#if'. Such comments should always be used, except in short conditionals that are not nested.

Above two example have shown how to use '#if' directive.

#else directive

The '#else' directive can be added to a conditional to provide alternative text to be used if the condition is false. This is what it looks like:

```
#if expression  
text-if-true  
#else /* Not expression */  
text-if-false  
#endif /* Not expression */
```

If expression is nonzero, and thus the text-if-true is active, then '#else' acts like a failing conditional and the text-if-false is ignored.

#elif directive

'#elif' stands for "else if". Like '#else', it goes in the middle of a '#if' - '#endif' pair and subdivides it; it does not require a matching '#endif' of its own. Like '#if', the '#elif' directive includes an expression to be tested.

The text following the '#elif' is processed only if the original '#if' condition failed and the '#elif' condition succeeds. More than one '#elif' can go in the same '#if'-'#endif' group. Then the text after each '#elif' is processed only if the '#elif' condition succeeds after the original '#if' and any previous '#elif' directives within it have failed. '#else' is equivalent to '#elif 1', and '#else' is allowed after any number of '#elif' directives, but '#elif' may not follow '#else'.

Example

```
#define DEBUG
#define VC_V6
using System;
public class MyClass
{
    public static void Main()
    {
        #if (DEBUG && !VC_V6)
        Console.WriteLine("DEBUG is defined");
        #elif (!DEBUG && VC_V6)
        Console.WriteLine("VC_V6 is defined");
        #elif (DEBUG && VC_V6)
        Console.WriteLine("DEBUG and VC_V6 are defined");
        #else
        Console.WriteLine("DEBUG and VC_V6 are not defined");
        #endif
    }
}
```

Output

DEBUG and VC_V6 are defined

#endif directive

#**endif** specifies the end of a conditional directive, which began with the #if directive.

#define directive

#**define** lets you define a symbol, such that, by using the symbol as the expression passed to the #if directive, the expression will evaluate to **true**

#undef directive

#**undef** lets you undefine a symbol, such that, by using the symbol as the expression in a #if directive, the expression will evaluate to **false**.

Example

```
// compile with /D:DEBUG
#undef DEBUG
using System;
public class MyClass
{
    public static void Main()
    {
        #if DEBUG
        Console.WriteLine("DEBUG is defined");
        #else
        Console.WriteLine("DEBUG is not defined");
        #endif
    }
}
```

Output

DEBUG is not defined

The '#error' and '#warning' Directives

The directive '#error' causes the preprocessor to report a fatal error. The tokens forming the rest of the line following '#error' are used as the error message.

The directive '#warning' is like the directive '#error', but causes the preprocessor to issue a warning and continue preprocessing. The tokens following '#warning' are used as the warning message.

Example 1

```
#define DEBUG
public class MyClass
{
    public static void Main()
    {
        #if DEBUG
        #error DEBUG is defined
        #endif
    }
}
```

Example 2

```
#define DEBUG
public class MyClass
{
    public static void Main()
    {
        #if DEBUG
        #warning DEBUG is defined
        #endif
    }
}
```

```
#endif  
}  
}
```

#line

#line is a directive that specifies the original line number and source file name for subsequent input in the current preprocessor input file.

Example

```
using System;  
public class MyClass  
{  
    public static void Main()  
    {  
        #line 100 "abc.sc" // change file name in the compiler output  
        int i; // error will be reported on line 101  
    }  
}
```

#region

#region lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor.

Example

```
#region MyClass definition  
public class MyClass  
{  
    public static void Main()  
    {  
    }  
}  
#endregion
```

#endregion

#endregion marks the end of a **#region** block.

Error Handling

The error handling construct in Visual Studio .NET is known as structured exception handling. The constructs used may be new to Visual Basic users, but should be familiar to users of C++ or Java.

Structured exception handling is straightforward to implement, and the same concepts are applicable to either VB.NET or C#. Throughout this section, example code will be shown in both languages.

VB .NET allows backward compatibility by also providing unstructured exception handling, via the familiar On Error GoTo statement and Err object, although this model is not discussed in this section.

Exceptions

Exceptions are used to handle error conditions in Visual Studio .NET. They provide information about the error condition.

An exception is an instance of a class which inherits from the System.Exception base class. Many different types of exception class are provided by the .NET Framework, and it is also possible to create your own exception classes. Each type extends the basic functionality of the System.Exception class by allowing further access to information about the specific type of error that has occurred.

An instance of an Exception class is created and thrown when the .NET Framework encounters an error condition. You can deal with exceptions by using the Try, Catch Finally construct.

Try, Catch, Finally

This construct allows you to catch errors that are thrown within your code. An example of this construct is shown below. An attempt is made to rotate an envelope, which throws an error.

C#

```
try
{
    IEnvelope env = new EnvelopeClass();
    env.PutCoords(0D, 0D, 10D, 10D);
    ITransform2D trans = (ITransform2D) env;
    trans.Rotate(env.LowerLeft, 1D);
}
catch (System.Exception ex)
{
    MessageBox.Show("Error: " + ex.Message);
}
```

```
finally
{
    // Perform any tidy up code.
}
```

VB.NET

```
Try
    Dim env As IEnvelope = New EnvelopeClass()
    env.PutCoords(0D, 0D, 10D, 10D)
    Dim trans As ITransform2D = env
    trans.Rotate(env.LowerLeft, 1D)
Catch ex As System.Exception
    MessageBox.Show("Error: " + ex.Message)
Finally
    ' Perform any tidy up code.
End Try
```

The Try block is placed around the code which may fail. If an error is thrown within the Try block, the point of execution will switch to the first Catch block.

The Catch block handles a thrown error. A Catch block is executed when the Type of a thrown error matches the Type of error specified by the Catch block. You can have more than one Catch block to handle different kinds of errors. The code shown below checks first if the exception thrown is a DivideByZeroException.

C#

```
...
catch (DivideByZeroException divEx)
{
    // Perform divide by zero error handling.
}
catch (System.Exception ex)
{
    // Perform general error handling.
}
...
```

VB.NET

```
...
Catch divEx As DivideByZeroException
    // Perform divide by zero error handling.
Catch ex As System.Exception
    // Perform general error handling.
...
```

If you do have more than one Catch block, note that the more specific exception Types should precede the general System.Exception, which will always succeed the type check.

The Finally block is always executed, either after the Try block completes, or after a Catch block, if an error was thrown. The Finally block should therefore contain code which must always be executed, for example to clean up resources like file handles or database connections.

If you do not have any cleanup code, you do not need to include a Finally block.

Code without exception handling

If a line of code not contained in a Try block throws an error, the .NET runtime searches for a Catch block in the calling function, continuing up the call stack until a Catch block is found.

If no Catch block is specified in the call stack at all, the exact outcome may depend on the location of the executed code and the configuration of the .NET runtime. It is therefore advisable to at least include a Try, Catch, Finally construct for all entry points to a program.

Error from COM components

The structured exception handling model differs from the HRESULT model used by COM. C++ developers could easily ignore a error condition in an HRESULT if they wished. In Visual Basic 6 however, an error condition in an HRESULT would populate the Err object and raise an error.

The .NET runtime's handling of errors from COM components is somewhat similar to the way COM errors were handled at VB 6. If a .NET program calls a function in a COM component (through the COM interop services) and an error condition is returned as the HRESULT, the HRESULT is used to populate an instance of the COMException class. This is then thrown by the .NET runtime, where you can handle it in the usual way, by using a Try, Catch Finally block.

It is advisable therefore to enclose all code that may raise an error in a COM component within a Try block with a corresponding Catch block to catch a COMException. Below is the first example rewritten to check for an error from a COM component.

C#

```
try
{
    IEnvelope env = new EnvelopeClass();
    env.PutCoords(0D, 0D, 10D, 10D);
    ITransform2D trans = (ITransform2D) env;
    trans.Rotate(env.LowerLeft, 1D);
}
catch (COMException COMex)
{
    if (COMex.ErrorCode == -2147220984)
```

```

        MessageBox.Show("You cannot rotate an Envelope");
    else
        MessageBox.Show
            ("Error " + COMex.ErrorCode.ToString() + ": " +
COMex.Message);
    }
catch (System.Exception ex)
{
    MessageBox.Show("Error: " + ex.Message);
}
...

```

VB.NET

```

Try
    Dim env As IEnvelope = New EnvelopeClass()
    env.PutCoords(0D, 0D, 10D, 10D)
    Dim trans As ITransform2D = env
    trans.Rotate(env.LowerLeft, 1D)
Catch COMex As COMException
    If (COMex.ErrorCode = -2147220984) Then
        MessageBox.Show("You cannot rotate an Envelope")
    Else
        MessageBox.Show _
            ("Error " + COMex.ErrorCode.ToString() + ": " + COMex.Message)
    End If
Catch ex As System.Exception
    MessageBox.Show("Error: " + ex.Message)
...

```

The COMException class belongs to the System.Runtime.InteropServices namespace. It provides access to the value of the original HRESULT via the ErrorCode property, which you can test, to find out which error condition occurred.

Throwing errors and the exception hierarchy

If you are coding a user interface, you may wish to attempt to correct the error condition in code and try the call again. Alternatively you may wish to report the error to the user to let them decide which course of action to take; here you can make use of the Message property of the Exception class to identify the problem.

However, if you are writing a function that is only called from other code, you may wish to deal with an error by creating a specific error condition and propagating this error to the caller. You can do exactly this by using the Throw keyword.

To simply throw the existing error to the caller function, write your error handler simply by using the Throw keyword, as shown below.

C#

```

catch (System.Exception ex)

```



```

{
    throw;
}
...

```

VB.NET

```

Catch ex As System.Exception
    Throw
...

```

If you wish to propagate a different or more specific error back to the caller, you should create a new instance of an Exception class, populate it appropriately, and throw this exception back to the caller. The example shown below uses the ApplicationException constructor to set the Message property.

C#

```

catch (System.Exception ex)
{
    throw new ApplicationException("You had an error in your
application");
}
...

```

VB.NET

```

Catch ex As System.Exception
    Throw New ApplicationException _
        ("You had an error in your application")
...

```

If you do this however, the original exception is lost. In order to allow complete error information to be propagated, the Exception class includes the InnerException property. This property should be set to equal the caught exception, before the new exception is thrown. This creates an error hierarchy. Again, the example shown below uses the ApplicationException constructor to set the InnerException and Message properties.

C#

```

catch (System.Exception ex)
{
    System.ApplicationException appEx =
        new ApplicationException("You had an error in your application",
ex);
    throw appEx;
}
...

```

VB.NET

```

Catch ex As System.Exception
    Dim appEx As System.ApplicationException = _
        New ApplicationException("You had an error in your application",
ex)
    Throw appEx
...

```

In this way, the function that eventually deals with the error condition can access all the information about the cause of the condition and its context.

If you throw an error, the current function's Finally clause will be executed before control is returned to the calling function.

Writing your error handler

The best approach to handling an error will depend on exactly what error is thrown, and in what context. You may find it useful to review the practices described in the MSDN topic [Best Practices for Handling Exceptions](#).

Exception handling is an in built mechanism in .NET framework to detect and handle run time errors. The .NET framework contains lots of standard exceptions. The exceptions are anomalies that occur during the execution of a program. They can be because of user, logic or system errors. If a user (programmer) do not provide a mechanism to handle these anomalies, the .NET run time environment provide a default mechanism, which terminates the program execution.

C# provides three keywords try, catch and finally to do exception handling. The try encloses the statements that might throw an exception whereas catch handles an exception if one exists. The finally can be used for doing any clean up process.

The general form try-catch-finally in C# is shown below

```

try
{
// Statement which can cause an exception.
}
catch(Type x)
{
// Statements for handling the exception
}
finally
{
//Any cleanup code
}

```

If any exception occurs inside the try block, the control transfers to the appropriate catch block and later to the finally block.

But in C#, both catch and finally blocks are optional. The try block can exist either with one or more catch blocks or a finally block or with both catch and finally blocks.

If there is no exception occurred inside the try block, the control directly transfers to finally block. We can say that the statements inside the finally block is executed always. Note that it is an error to transfer control out of a finally block by using break, continue, return or goto.

In C#, exceptions are nothing but objects of the type Exception. The Exception is the ultimate base class for any exceptions in C#. The C# itself provides couple of standard exceptions. Or even the user can create their own exception classes, provided that this should inherit from either Exception class or one of the standard derived classes of Exception class like DivideByZeroException or ArgumentException etc.

Uncaught Exceptions

The following program will compile but will show an error during execution. The division by zero is a runtime anomaly and program terminates with an error message. Any uncaught exceptions in the current context propagate to a higher context and looks for an appropriate catch block to handle it. If it can't find any suitable catch blocks, the default mechanism of the .NET runtime will terminate the execution of the entire program.

```
//C#: Exception Handling
//Author: rajeshvs@msn.com
using System;
class MyClient
{
    public static void Main()
    {
        int x = 0;
        int div = 100/x;
        Console.WriteLine(div);
    }
}
```

The modified form of the above program with exception handling mechanism is as follows. Here we are using the object of the standard exception class DivideByZeroException to handle the exception caused by division by zero.

```
//C#: Exception Handling
using System;
class MyClient
{
    public static void Main()
    {
        int x = 0;
        int div = 0;
        try
```

```

{
div = 100/x;
Console.WriteLine("This line in not executed");
}
catch(DivideByZeroException de)
{
Console.WriteLine("Exception occurred");
}
Console.WriteLine("Result is {0}",div);
}
}

```

In the above case the program do not terminate unexpectedly. Instead the program control passes from the point where exception occurred inside the try block to the catch blocks. If it finds any suitable catch block, executes the statements inside that catch and continues with the normal execution of the program statements. If a finally block is present, the code inside the finally block will get also be executed.

```

//C#: Exception Handling
using System;
class MyClient
{
public static void Main()
{
int x = 0;
int div = 0;
try
{
div = 100/x;
Console.WriteLine("Not executed line");
}
catch(DivideByZeroException de)
{
Console.WriteLine("Exception occurred");
}
finally
{
Console.WriteLine("Finally Block");
}
Console.WriteLine("Result is {0}",div);
}
}

```

Remember that in C#, the catch block is optional. The following program is perfectly legal in C#.

```

//C#: Exception Handling
using System;
class MyClient
{
public static void Main()
{

```

```

int x = 0;
int div = 0;
try
{
div = 100/x;
Console.WriteLine("Not executed line");
}
finally
{
Console.WriteLine("Finally Block");
}
Console.WriteLine("Result is {0}",div);
}
}

```

But in this case, since there is no exception handling catch block, the execution will get terminated. But before the termination of the program statements inside the finally block will get executed. In C#, a try block must be followed by either a catch or finally block.

Multiple Catch Blocks

A try block can throw multiple exceptions, which can handle by using multiple catch blocks. Remember that more specialized catch block should come before a generalized one. Otherwise the compiler will show a compilation error.

```

//C#: Exception Handling: Multiple catch
using System;
class MyClient
{
public static void Main()
{
int x = 0;
int div = 0;
try
{
div = 100/x;
Console.WriteLine("Not executed line");
}
catch(DivideByZeroException de)
{
Console.WriteLine("DivideByZeroException" );
}
catch(Exception ee)
{
Console.WriteLine("Exception" );
}
finally
{
Console.WriteLine("Finally Block");
}
Console.WriteLine("Result is {0}",div);
}
}

```

```
}
```

Catching all Exceptions

By providing a catch block without a brackets or arguments, we can catch all exceptions occurred inside a try block. Even we can use a catch block with an Exception type parameter to catch all exceptions happened inside the try block since in C#, all exceptions are directly or indirectly inherited from the Exception class.

```
//C#: Exception Handling: Handling all exceptions
using System;
class MyClient
{
    public static void Main()
    {
        int x = 0;
        int div = 0;
        try
        {
            div = 100/x;
            Console.WriteLine("Not executed line");
        }
        catch
        {
            Console.WriteLine("oException" );
        }
        Console.WriteLine("Result is {0}",div);
    }
}
```

The following program handles all exception with Exception object.

```
//C#: Exception Handling: Handling all exceptions
using System;
class MyClient
{
    public static void Main()
    {
        int x = 0;
        int div = 0;
        try
        {
            div = 100/x;
            Console.WriteLine("Not executed line");
        }
        catch(Exception e)
        {
            Console.WriteLine("oException" );
        }
        Console.WriteLine("Result is {0}",div);
    }
}
```

Throwing an Exception

In C#, it is possible to throw an exception programmatically. The 'throw' keyword is used for this purpose. The general form of throwing an exception is as follows.

```
throw exception_obj;
```

For example the following statement throw an ArgumentException explicitly.

```
throw new ArgumentException("Exception");
```

```
//C#: Exception Handling:
using System;
class MyClient
{
    public static void Main()
    {
        try
        {
            throw new DivideByZeroException("Invalid Division");
        }
        catch(DivideByZeroException e)
        {
            Console.WriteLine("Exception" );
        }
        Console.WriteLine("LAST STATEMENT");
    }
}
```

Re-throwing an Exception

The exceptions, which we caught inside a catch block, can re-throw to a higher context by using the keyword throw inside the catch block. The following program shows how to do this.

```
//C#: Exception Handling: Handling all exceptions
using System;
class MyClass
{
    public void Method()
    {
        try
        {
            int x = 0;
            int sum = 100/x;
        }
        catch(DivideByZeroException e)
        {
            throw;
        }
    }
}
class MyClient
```

```

{
public static void Main()
{
MyClass mc = new MyClass();
try
{
mc.Method();
}
catch(Exception e)
{
Console.WriteLine("Exception caught here" );
}
Console.WriteLine("LAST STATEMENT");
}
}

```

Standard Exceptions

There are two types of exceptions: exceptions generated by an executing program and exceptions generated by the common language runtime. `System.Exception` is the base class for all exceptions in C#. Several exception classes inherit from this class including `ApplicationException` and `SystemException`. These two classes form the basis for most other runtime exceptions. Other exceptions that derive directly from `System.Exception` include `IOException`, `WebException` etc.

The common language runtime throws `SystemException`. The `ApplicationException` is thrown by a user program rather than the runtime. The `SystemException` includes the `ExecutionEngineException`, `StackOverflowException` etc. It is not recommended that we catch `SystemExceptions` nor is it good programming practice to throw `SystemExceptions` in our applications.

- `System.OutOfMemoryException`
- `System.NullReferenceException`
- `System.InvalidCastException`
- `System.ArrayTypeMismatchException`
- `System.IndexOutOfRangeException`
- `System.ArithmeticException`
- `System.DivideByZeroException`
- `System.OverflowException`

User-defined Exceptions

In C#, it is possible to create our own exception class. But `Exception` must be the ultimate base class for all exceptions in C#. So the user-defined exception classes must inherit from either `Exception` class or one of its standard derived classes.

```

//C#: Exception Handling: User defined exceptions
using System;
class MyException : Exception
{
public MyException(string str)
{
Console.WriteLine("User defined exception");
}
}

```



```
}  
}  
class MyClient  
{  
public static void Main()  
{  
try  
{  
throw new MyException("RAJESH");  
}  
catch(Exception e)  
{  
Console.WriteLine("Exception caught here" + e.ToString());  
}  
Console.WriteLine("LAST STATEMENT");  
}  
}
```

13

Multithreading

Basic Threading in C#

Threading is a great way to make your application smoother. A single thread in a C# application, is an independent execution path that can run simultaneously with the main application thread. Of course, C# supports **multithreading**. And to use the threading namespace, you can directly call it from `System.Threading`, or import it:

```
1using System.Threading;
```

Without this, a basic HTTP request makes your application unavailable in a span of time. And if you are using a progress bar, you won't see it updating (since the application is unresponsive at this state).

Single Thread

The most basic way to implement this, is by using the `Thread` class. An example:

```
1Thread tMain = new Thread(new ThreadStart(someMethod));  
2tMain.Start();
```

Wherein `someMethod` is a void function that contains the code you want to execute. With this, the thread will execute in parallel with the main thread. The only drawback, is you cannot modify controls created in the main thread like you normally use to. This will result into a Cross-thread operation not valid: Control accessed from a thread other than the thread it was created on exception. The solution is pretty simple:

```
1// Won't Work  
2textBox1.Text = "Hello";  
3  
4// Will Work  
5this.Invoke(new MethodInvoker(delegate { textBox1.Text = "Hello"; }));
```

This approach functions just the same as a `BackgroundWorker` instance.

Powertip: You can disable all the controls in your form painlessly without setting the `Enabled` property of each control (imagine having 20+ controls in your form).

```
1 private void controlsEnableToggle(bool val)
2 {
3     foreach (Control c in this.Controls)
4     {
5         c.Enabled = val;
6     }
7 }
```

Multiple Threads

Dealing with multiple repetitive tasks with a single line of execution is painful, especially in large numbers. C# supports multithreading. This means, we can initiate as many threads as we like, and it will do the job. So how to do that? A simple for-loop.

```
1 for (int i = 0; i < 10; i++) {
2     Thread tMain = new Thread(new ThreadStart(someMethod));
3     tMain.Start();
4 }
```

The above code works, but what if we need to supply parameters to our method?

```
01 // .NET 2
02 for (int i = 0; i < 10; i++) {
03     Thread tMain = new Thread(new
04 ParameterizedThreadStart(someMethod));
05     tMain.Start(param);
06 }
07 // .NET 3.5 & 4
08 for (int i = 0; i < 10; i++) {
09     Thread tMain = new Thread(unused => someMethod(param));
10     tMain.Start();
11 }
```

Pausing and Stopping Threads

Whenever you close a form, the main thread will stop, but the threads you created are not. Thus, the process is still alive. To stop this, you can just go to the task manager, and end the process from there. But that's not user-friendly.

You cannot actually stop or pause threads immediately. You can never tell what the thread is doing, and terminating them abnormally may cause side effects. For this, `ManualResetEvent` is used.

```
1ManualResetEvent pauseEvent = new ManualResetEvent(true);
2ManualResetEvent stopEvent = new ManualResetEvent(false);
```

The bool values passed to the constructor indicates whether the initial state of the instance is signaled or not. This is pretty easy to use.

```
01
02ManualResetEvent pauseEvent = new ManualResetEvent(true);
03ManualResetEvent stopEvent = new ManualResetEvent(false);
04private void someMethod(string param) {
05    // Stops the thread
06    if (stopEvent.WaitOne(0))
07        return;
08
09    // Pauses the thread
10    pauseEvent.WaitOne(Timeout.Infinite);
11
12    // Your code
13}
14
```

To change the signaled state of these events, we will use `Reset` and `Set` methods. Here's how to use them in relation to the above code:

```
01
02// Stops the threads
03private void Form1_FormClosing(object sender, FormClosingEventArgs e)
04{
05    stopEvent.Set();
06}
07// Pauses the threads
08private void Button1_Click(object sender, EventArgs e)
09{
10    pauseEvent.Reset();
11}
12// Resumes the threads
13private void Button2_Click(object sender, EventArgs e)
14{
15    pauseEvent.Set();
16}
17
```

Create New Thread [C#]

This example shows how to create a new **thread in .NET Framework**. First, create a new [ThreadStart](#) delegate. The delegate points to a method that will be executed by the new thread. Pass this delegate as a parameter when creating a new [Thread](#) instance. Finally, call the [Thread.Start](#) method to run your method (in this case `WorkThreadFunction`) on background.

```
[C#]
using System.Threading;

Thread thread = new Thread(new ThreadStart(WorkThreadFunction));
thread.Start();
```

The `WorkThreadFunction` could be defined as follows.

```
[C#]
public void WorkThreadFunction()
{
    try
    {
        // do any background work
    }
    catch (Exception ex)
    {
        // log errors
    }
}
```

Introduction to Threads In C#

Threads: Threads are often called lightweight processes. However they are not processes. A Thread is a small set of executable instructions, which can be used to isolate a task from a process. Multiple threads are efficient way to obtain parallelism of hardware and give interactive user interaction to your applications.

C# Thread: .NET Framework has thread-associated classes in `System.Threading` namespace. The following steps demonstrate how to create a thread in C#.

Step 1. Create a `System.Threading.Thread` object.

Creating an object to `System.Threading.Thread` creates a managed thread in .Net environment. The `Thread` class has only one constructor, which takes a `ThreadStart` delegate as parameter. The `ThreadStart` delegate is wrap around the callback method, which will be called when we start the thread.

Step 2: Create the call back function

This method will be a starting point for our new thread. It may be an instance function of a class or a static function. Incase of instance function, we should create an object of the class, before we create the `ThreadStart` delegate. For static functions we can directly use the function name to instantiate the delegate. The callback function should have void as both return type and parameter. Because the `ThreadStart` delegate function is declared like this. (For more information on delegate see MSDN for “Delegates”).

Step 3: Starting the Thread.

We can start the newly created thread using the `Thread`'s `Start` method. This is an asynchronous method, which requests the operating system to start the current thread.

For Example:

 Collapse | [Copy Code](#)

```
// This is the Call back function for thread.

Public static void MyCallbackFunction()

{
    while (true)

        {
            System.Console.WriteLine(" Hey!, My Thread Function Running");
            .....
        }
}

public static void Main(String []args)

{

    // Create an object for Thread

    Thread MyThread = new Thread(new ThreadStart
    (MyCallbackFunction));

    MyThread.Start()
```

```
.....  
}
```

Killing a Thread:

We can kill a thread by calling the `Abort` method of the thread. Calling the `Abort` method causes the current thread to exit by throwing the `ThreadAbortException`.

☒Collapse | [Copy Code](#)
`MyThread.Abort();`

Suspend and Resuming Thread:

We can suspend the execution of a thread and once again start its execution from another thread using the Thread object's `Suspend` and `Resume` methods.

☒Collapse | [Copy Code](#)
`MyThread.Suspend() // causes suspend the Thread Execution.`
`MyThread.Resume() // causes the suspended Thread to resume its execution.`

Thread State:

A Thread can be in one the following state.

Thread State	Description
Unstarted	Thread is Created within the common language run time but not Started still.
Running	After a Thread calls Start method
WaitSleepJoin	After a Thread calls its wait or Sleep or Join method.
Suspended	Thread Responds to a Suspend method call.
Stopped	The Thread is Stopped, either normally or Aborted.

We can check the current state of a thread using the Thread's `ThreadState` property.

Thread Priority:

The Thread class's `ThreadPriority` property is used to set the priority of the Thread. A Thread may have one of the following values as its Priority: `Lowest`, `BelowNormal`, `Normal`, `AboveNormal`, `Highest`. The default property of a thread is `Normal`.

Synchronizing Threads in Multithreaded application in .Net – C#

The Problem “Concurrency”

When you build multithreaded application, your program needs to ensure that shared data should be protected from against the possibility of multiple threads engagement with its value. What's gonna happen if multiple threads were accessing the data at the same point? CLR can suspend your any thread for a while who's going to update the value or is in the middle of updating the value and same time a thread comes to read that value which is not completely updated, that thread is reading an uncompleted/unstable data.

To illustrate the problem of concurrency let write some line of code

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("----Synchronnization of Threads-----");
        Console.WriteLine("Main Thread {0}",
Thread.CurrentThread.ManagedThreadId);
        Printer p = new Printer();

        Thread[] threads = new Thread[5];

        //Queue 5 threads
        for (int i = 0; i < 5; i++)
        {
            threads[i] = new Thread(new ThreadStart(p.PrintNumbersNonSync));
        }
        foreach (Thread t in threads)
        {
            t.Start();
        }

        Console.ReadLine();
    }
}

class Printer
{
    public void PrintNumbersNonSync()
    {
        Console.WriteLine(" ");
        Console.WriteLine("Executing Thread {0}",
Thread.CurrentThread.ManagedThreadId);
        for (int i = 1; i <= 10; i++)
        {
            Console.Write(i + " ");
        }
    }
}
```


}

Run this program multiple times and watch the output:

Output1:

```
-----Synchronnization of Threads-----  
Main Thread 10  
  
Executing Thread 12  
  
Executing Thread 11  
1 2 1 2 3 4 5 6 7 8 9 10  
3 4 5 6 7 8 9 10 Executing Thread 13  
1 2 3 4 5 6 7 8 9 10  
Executing Thread 14  
1 2 3 4 5 6 7 8 9 10  
Executing Thread 15  
1 2 3 4 5 6 7 8 9 10
```

Output2:

```
-----Synchronnization of Threads-----  
Main Thread 10  
  
Executing Thread 11  
1 2 3 4 5 6 7 8  
Executing Thread 12  
1 9 10 2 3 4 5 6 7 8 9 10  
Executing Thread 13  
1 2 3 4 5 6 7 8 9 10  
Executing Thread 14  
1 2  
Executing Thread 15  
1 2 3 4 5 6 7 8 9 10 3 4 5 6 7 8 9 10 _
```

your output could be different from these.

As you can see all the output would be different as many time you run the program. What happening here is that all the threads are sharing the same object of Printer class and trying to execute the same function at the same time so every time the shared data is being updated in a random pattern which is an unstable state.

Synchronization of threads Solution to the problem of concurrency

Use the locks whenever there's a Shared section. C# provides Lock keyword that can be used to lock the section that is being accessed by multiple threads. This is the very basic technique to avoid the instability in multithreaded environment while programming with c#.

The Lock keyword requires you to specify the token (an object reference) that must be acquired by a thread to enter within the lock scope. In case of private method you can lock it down by passing the reference of current type using "this" keyword.

For e.g.

```
public void PrintNumbersSynchronized()  
{  
    //Synchronization thread  
    lock (this)  
    {  
        Console.WriteLine(" ");  
        Console.WriteLine("Executing Thread {0}",  
Thread.CurrentThread.ManagedThreadId);  
    }  
}
```

```

        for (int i = 1; i <= 10; i++)
        {
            Console.Write(i + " ");
        }
    }
}

```

However if you are locking down a region of code within a public member, it is safer (and best practice) to declare a private object member variable to server as the lock token:

```

class Printer
{
    // Synchronization token
    private Object ThreadLock = new object();

    public void PrintNumbersSynchronized()
    {
        //Synchronization thread
        lock (ThreadLock)
        {
            Console.WriteLine(" ");
            Console.WriteLine("Executing Thread {0}",
Thread.CurrentThread.ManagedThreadId);
            for (int i = 1; i <= 10; i++)
            {
                Console.Write(i + " ");
            }
        }
    }
}

```

Output:

```

-----Synchronnization of Threads-----
Main Thread 10

Executing Thread 12
1 2 3 4 5 6 7 8 9 10
Executing Thread 13
1 2 3 4 5 6 7 8 9 10
Executing Thread 14
1 2 3 4 5 6 7 8 9 10
Executing Thread 15
1 2 3 4 5 6 7 8 9 10
Executing Thread 16
1 2 3 4 5 6 7 8 9 10 _

```

Other methods to perform the synchronization

Monitor is class in System.Threading namespace that also provides the same functionality. Actually lock is just a shorthand notation of Monitor class. Once compiler processed a lock it actually resolves to the following:

```

public void PrintNumbersSync()
{
    Monitor.Enter(ThreadLock);
    try
    {
        Console.WriteLine(" ");
    }
}

```

```

        Console.WriteLine("Executing Thread {0}",
Thread.CurrentThread.ManagedThreadId);
        for (int i = 1; i <= 10; i++)
        {
            Console.Write(i + " ");
        }
    }
    finally
    {
        Monitor.Exit(ThreadLock);
    }
}

```

Here little more code you can see having the try..finally block. Other than Enter() and Exit() methods, Monitor class also provides the methods like Monitor.Pulse() and PulseAll() to inform the waiting threads that its completed.

Simple operations using the *Interlocked* Type

To perform some basic operation it's not necessary that you write the block using the lock or Monitor. System.Threading namespace provides an interesting class that can help you out.

Now you can replace this

```

lock (ThreadLock)
{
    intVal++;
}

```

By

```
int newVal = Interlocked.Increment(ref intVal);
```

It returns the new values of updated variable as well as update the referenced value at the same time.

Similarly,

Add()

Exchange() //for swapping

CompareExchange() // Compare a value and then exchange

Equals()

Decrement()

Read()

Are some basic operations you can perform in a multithreaded environment to make your threadsafe using the Interlocked class.

Synchronization using the [Synchronization] attribute

The **Synchronization** attribute is a member of System.Runtime.Remoting.Contexts namespace. In essence, this class-level attribute effectively lock downs all instance members code of the object for the thread safety. Additionally you have to derive your class from **ContextBoundObject** to keep your object with in the contextual boundaries. Here's the complete code:

```

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

```

```

        Console.WriteLine("----Synchronnization of Threads-----");
        Console.WriteLine("Main Thread {0}",
Thread.CurrentThread.ManagedThreadId);
        Printer p = new Printer();

        Thread[] threads = new Thread[5];

        //Queue 5 threads
        for (int i = 0; i < 5; i++)
        {
            threads[i] = new Thread(new ThreadStart(p.PrintNumbersNonSync));
        }
        foreach (Thread t in threads)
        {
            t.Start();
        }

        Console.ReadLine();
    }
}

[Synchronization]
class Printer : ContextBoundObject
{
    public void PrintNumbersNonSync()
    {
        Console.WriteLine(" ");
        Console.WriteLine("Executing Thread {0}",
Thread.CurrentThread.ManagedThreadId);
        for (int i = 1; i <= 10; i++)
        {
            Console.Write(i + " ");
        }
    }
}
}

```

Output:

```

-----Synchronnization of Threads-----
Main Thread 10
Executing Thread 12
1 2 3 4 5 6 7 8 9 10
Executing Thread 13
1 2 3 4 5 6 7 8 9 10
Executing Thread 14
1 2 3 4 5 6 7 8 9 10
Executing Thread 15
1 2 3 4 5 6 7 8 9 10
Executing Thread 16
1 2 3 4 5 6 7 8 9 10 _

```

This appraoch is a lazy way to write thread safe code because CLR can lock non-thread sensitive data and that could be an victim of OverLocking. So please choose this approach wisely and carefully.

Happy Threading...!!

Introduction

Overview

Multithreading or free-threading is the ability of an operating system to concurrently run programs that have been divided into subcomponents, or threads.

Technically, multithreaded programming requires a multitasking/multithreading operating system, such as GNU/Linux, Windows NT/2000 or OS/2; capable of running many programs concurrently, and of course, programs have to be written in a special way in order to take advantage of these multitasking operating systems which appear to function as multiple processors. In reality, the user's sense of time is much slower than the processing speed of a computer, and multitasking appears to be simultaneous, even though only one task at a time can use a computer processing cycle.

Objective

The objective of this document is:

- A brief Introduction to Threading
- Features of Threading
- Threading Advantages

Features and Benefits of Threads

Mutually exclusive tasks, such as gathering user input and background processing can be managed with the use of threads. Threads can also be used as a convenient way to structure a program that performs several similar or identical tasks concurrently.

One of the advantages of using the threads is that you can have multiple activities happening simultaneously. Another advantage is that a developer can make use of threads to achieve faster computations by doing two different computations in two threads instead of serially one after the other.

Threading Concepts in C#

In .NET, threads run in AppDomains. An AppDomain is a runtime representation of a logical process within a physical process. And a thread is the basic unit to which the OS allocates processor time. To start with, each AppDomain is started with a single thread. But it is capable of creating other threads from the single thread and from any created thread as well.

How do they work

A multitasking operation system divides the available processor time among the processes and threads that need it. A thread is executed in the given time slice, and then it is suspended and execution starts for next thread/process in the queue. When the OS switches from one thread to another, it saves thread context for preempted thread and loads the thread context for the thread to execute.

The length of time slice that is allocated for a thread depends on the OS, the processor, as also on the priority of the task itself.

Working with threads

In .NET framework, `System.Threading` namespace provides classes and interfaces that enable multi-threaded programming. This namespace provides:

- `ThreadPool` class for managing group of threads,
- `Timer` class to enable calling of delegates after a certain amount of time,
- A `Mutex` class for synchronizing mutually exclusive threads, along with classes for scheduling the threads, sending wait notifications and deadlock resolutions.

Information on this namespace is available in the help documentations in the Framework SDK.

Defining and Calling threads

To get a feel of how Threading works, run the below code:

☒Collapse | [Copy Code](#)

```
using System;
using System.Threading;

public class ServerClass
{
    // The method that will be called when the thread is started.
    public void Instance Method()
    {
        Console.WriteLine("You are in InstanceMethod.Running on Thread A");
        Console.WriteLine("Thread A Going to Sleep Zzzzzzzz");

        // Pause for a moment to provide a delay to make threads more
        // apparent.
        Thread.Sleep(3000);
        Console.WriteLine("You are Back in InstanceMethod.Running on Thread A");
    }

    public static void StaticMethod()
    {

```

```

        Console.WriteLine("You are in StaticMethod. Running on Thread
B.");
        // Pause for a moment to provide a delay to make threads more
        // apparent.
        Console.WriteLine("Thread B Going to Sleep Zzzzzzzz");

        Thread.Sleep(5000);
        Console.WriteLine("You are back in static method. Running on
Thread B");
    }
}

public class Simple
{
    public static int Main(String[] args)
    {
        Console.WriteLine ("Thread Simple Sample");
        ServerClass serverObject = new ServerClass();
        // Create the thread object, passing in the
        // serverObject.InstanceMethod method using a ThreadStart
        delegate.
        Thread InstanceCaller = new
            Thread(new ThreadStart(serverObject.InstanceMethod));

        // Start the thread.
        InstanceCaller.Start();

        Console.WriteLine("The Main() thread calls this " +
            "after starting the new InstanceCaller thread.");

        // Create the thread object, passing in the
        // serverObject.StaticMethod method using a ThreadStart
        delegate.
        Thread StaticCaller = new Thread(new
            ThreadStart(ServerClass.StaticMethod));
        // Start the thread.
        StaticCaller.Start();
        Console.WriteLine("The Main () thread calls this " +
            "after starting the new StaticCaller threads.");
        return 0;
    }
}

```

If the code in this example is compiled and executed, you would notice how processor time is allocated between the two method calls. If not for threading, you would have to wait till the first method slept for 3000 secs for the next method to be called. Try disabling threading in the above code and notice how they work. Nevertheless, execution time for both would be the same.

An important property of this class (which is also settable) is `Priority`.

Scheduling Threads

Every thread has a thread priority assigned to it. Threads created within the common language runtime are initially assigned the priority of `ThreadPriority.Normal`. Threads created outside the runtime retain the priority they had before they entered the managed environment. You can get or set the priority of any thread with the `Thread.Priority` property.

Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the operating system. The details of the scheduling algorithm used to determine the order in which threads are executed varies with each operating system. Under some operating systems, the thread with the highest priority (of those threads that can be executed) is always scheduled to run first. If multiple threads with the same priority are available, the scheduler cycles through the threads at that priority, giving each thread a fixed time slice in which to execute. As long as a thread with a higher priority is available to run, lower priority threads do not get to execute. When there are no more run able threads at a given priority, the scheduler moves to the next lower priority and schedules the threads at that priority for execution. If a higher priority thread becomes run able, the lower priority thread is preempted and the higher priority thread is allowed to execute once again. On top of all that, the operating system can also adjust thread priorities dynamically as an application's user interface is moved between foreground and background. Other operating systems might choose to use a different scheduling algorithm.

Pausing and Resuming threads

After you have started a thread, you often want to pause that thread for a fixed period of time. Calling [Thread.Sleep](#) causes the current thread to immediately block for the number of milliseconds you pass to `Sleep`, yielding the remainder of its time slice to another thread. One thread cannot call `Sleep` on another thread. Calling `Thread.Sleep(Timeout.Infinite)` causes a thread to sleep until it is interrupted by another thread that calls [Thread.Interrupt](#) or is aborted by [Thread.Abort](#).

Thread Safety

When we are working in a multi threaded environment, we need to maintain that no thread leaves the object in an invalid state when it gets suspended. Thread safety basically means the members of an object always maintain a valid state when used concurrently by multiple threads.

There are multiple ways of achieving this – The `Mutex` class or the `Monitor` classes of the Framework enable this, and more information on both is available in the Framework SDK documentation. What we are going to look at here is the use of locks.

You put a lock on a block of code – which means that that block has to be executed at one go and that at any given time, only one thread could be executing that block.

The syntax for the `lock` would be as follows:

☒Collapse | [Copy Code](#)

```
using System;
using System.Threading;

//define the namespace, class etc.

...
public somemethod(...)
{
    ...
    lock(this)
    {
        Console.WriteLine("Inside the lock now");
        ...
    }
}
```

Synchronization Primitives

1. Simple blocking : `Sleep`, `Join`, and `Task.Wait` are simple blocking methods. These mechanisms wait until another thread to finish or period of time to elapse.
2. Locking : The standard exclusive locking constructs are `lock (Monitor.Enter/Monitor.Exit)`, `Mutex`, and `Spinlock`. The nonexclusive locking constructs are semaphore and the reader/writer locks.
3. Signaling : There are two commonly used signaling devices: Event wait handlers and `Monitor's Wait/Pulse` methods.
4. Nonblocking synchronization constructs : C# provide the following nonblocking constructs: `Thread.MemoryBarrier`, `Thread.VolatileWrite` and the `InterLocked` class.

Locking

Lock :

lock ensures that one thread does not enter a critical section of code while another thread is in the critical section. If another thread attempts to enter a locked code, it will wait, block, until the object is released. Using `lock` keyword only one thread can access section of code at a time and once it finishes then only another thread in a queue will be accessed.

Syntax of **lock** keyword :

```
Object lockObj= new Object();
lock (lockObj)
{
    // code section
}
```

Lock calls Enter at the beginning of the block and Exit at the end of the block.

DO's and DONT's :

- * Avoid locking on **public** types.
- * Avoid locking on instances which is not in your code control.
- * Better to lock on **private** objects
- * Lock on **private static** object variable to protect data common to all instances.

Monitor :

Monitor class also works same as lock in which it provides a mechanism that synchronizes access to objects. But it uses The **Enter** method allows one and only one thread to proceed into the following statements; all other threads are blocked until the executing thread calls **Exit**. It Monitor is unbound, which means it can be called directly from any context. This class is under **System.Threading** namespace.

Code snippet shows Monitor class:

```
Object lockObj= new Object();
```

```
Monitor.Enter(lockObj);
try
{
    DoSomething();
}
finally
{
    Monitor.Exit(lockObj);
}
```

Mutex:

A mutex is similar to a monitor; it prevents the simultaneous execution of a block of code by more than one thread at a time. Unlike monitors, however, a mutex can be used to synchronize threads across processes. It represents by the **mutex** class.

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

namespace ConsoleApplication11
{
    class SThread
```

```

{
    public void childThread()
    {
        for (int i = 0; i < 6; i++)
        {
            Console.WriteLine("generated thread {0}", i);
            Thread.Sleep(1000);
        }
        Console.WriteLine("exiting the generated thread");
    }
}
public class Thread3
{
    static void Main(string[] args)
    {
        SThread st = new SThread();
        ThreadStart ts = new ThreadStart(st.childThread);
        Thread t = new Thread(ts);
        t.Start();
        for (int i = 0; i < 6; i++)
        {
            Console.WriteLine("Main Thread {0}", i);
            Thread.Sleep(1000);
        }
        Console.WriteLine("exiting the main Thread");
    }
}

```

```

file:///C:/Documents and Settings/NEERJA/Local Settings/Application Data/Temporary Proje...
Main Thread 0
generated thread 0
Main Thread 1
generated thread 1
Main Thread 2
generated thread 2
Main Thread 3
generated thread 3
Main Thread 4
generated thread 4
Main Thread 5
generated thread 5
exiting the main Thread
exiting the generated thread

```

```

namespace ConsoleApplication11
{
    class SThread
    {
        string name;
        Thread t;
        public SThread(string s){
            name = s;
            t = new Thread(new ThreadStart(this.sThread));
        }
    }
}

```

```

        t.Name = name;
        t.Start();
    }
    public void sThread()
    {
        for (int i = 0; i < 6; i++)
        {
            Console.WriteLine("name of thread is {0} and the value
is {1}", name, i);
            Thread.Sleep(1000);
        }
        Console.WriteLine("Exiting the {0} Thread", name);
    }
}
public class Thread3
{
    static void Main(string[] args)
    {
        new SThread("first");
        new SThread("second");
        Console.WriteLine("Exiting the main Thread");
        Console.ReadLine();
    }
}
}

```



```

file:///C:/Documents and Settings/NEERJA/Local Settings/Application Data/Temporary Proje...
name of thread is first and the value is 0
Exiting the main Thread
name of thread is second and the value is 0
name of thread is first and the value is 1
name of thread is second and the value is 1
name of thread is first and the value is 2
name of thread is second and the value is 2
name of thread is first and the value is 3
name of thread is second and the value is 3
name of thread is first and the value is 4
name of thread is second and the value is 4
name of thread is first and the value is 5
name of thread is second and the value is 5
Exiting the first Thread
Exiting the second Thread

```

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace ConsoleApplication11
{
    class SThread
    {


```

```

        public void sthread()
        {
            for (int i = 0; i < 6; i++)
            {
                Console.WriteLine("name of thread is {0} and the value
is {1}", Thread.CurrentThread.Name,i);
                Thread.Sleep(1000);
            }
            Console.WriteLine("Exiting the {0}
Thread",Thread.CurrentThread.Name);
        }
    }
    public class Thread3
    {
        static void Main(string[] args)
        {
            SThread st1 = new SThread();
            Thread t = new Thread(new ThreadStart(st1.sthread));
            t.Name = "First";
            t.Start();

            SThread st2 = new SThread();
            Thread t1 = new Thread(new ThreadStart(st1.sthread));
            t1.Name = "Second";
            t1.Start();
            Console.WriteLine("Is the first thread alive {0}",
t.IsAlive);
            Console.WriteLine("Is the second thread alive {0}",
t1.IsAlive);
            Console.WriteLine("Waiting for the threads to complete");
            t.Join();
            t1.Join();
            Console.WriteLine("Is the first thread alive {0}",
t.IsAlive);
            Console.WriteLine("Is the first thread alive {0}",
t1.IsAlive);
            Console.ReadLine();
        }
    }
}

```



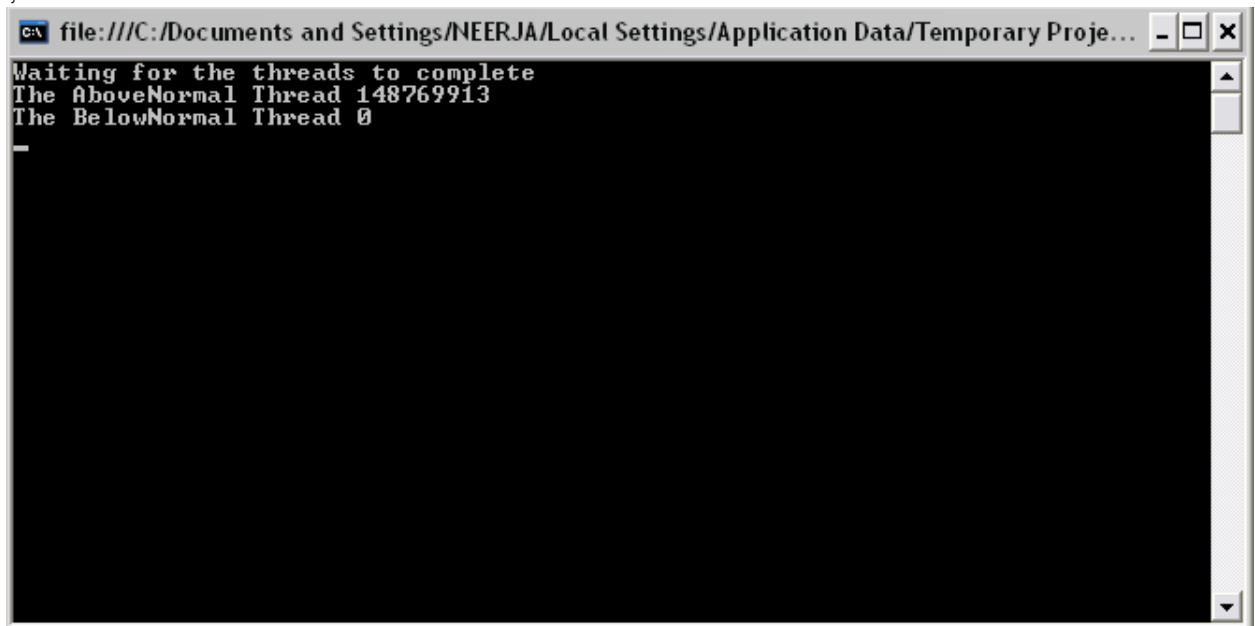
```
file:///C:/Documents and Settings/NEERJA/Local Settings/Application Data/Temporary Proje...
name of thread is Second and the value is 0
Is the first thread alive True
Is the second thread alive True
Waiting for the threads to complete
name of thread is First and the value is 0
name of thread is Second and the value is 1
name of thread is First and the value is 1
name of thread is Second and the value is 2
name of thread is First and the value is 2
name of thread is Second and the value is 3
name of thread is First and the value is 3
name of thread is Second and the value is 4
name of thread is First and the value is 4
name of thread is Second and the value is 5
name of thread is First and the value is 5
Exiting the Second Thread
Exiting the First Thread
Is the first thread alive False
Is the first thread alive False
```

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

namespace ConsoleApplication11
{
    class SThread
    {
        public int count=0;
        bool doit=true;
        public void sthread()
        {
            while (doit)
            {
                count++;
            }
        }
        public void stop(){
            doit=false;
        }
    }
    public class Thread3
    {
        static void Main(string[] args)
        {
            SThread st1 = new SThread();
            Thread t = new Thread(new ThreadStart(st1.sthread));
            t.Name = "First";
            t.Priority = ThreadPriority.AboveNormal;
            t.Start();
        }
    }
}
```

```
        SThread st2 = new SThread();
        Thread t1 = new Thread(new ThreadStart(st1.sthread));
        t1.Name = "Second";
        t1.Priority = ThreadPriority.BelowNormal;
        t1.Start();

        Thread.Sleep(1000);
        st1.stop();
        st2.stop();
        Console.WriteLine("Waiting for the threads to complete");
        t.Join();
        t1.Join();
        Console.WriteLine("The AboveNormal Thread {0}", st1.count);
        Console.WriteLine("The BelowNormal Thread {0}", st2.count);
        Console.ReadLine();
    }
}
```



```
file:///C:/Documents and Settings/NEERJA/Local Settings/Application Data/Temporary Proje...
Waiting for the threads to complete
The AboveNormal Thread 148769913
The BelowNormal Thread 0
```

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

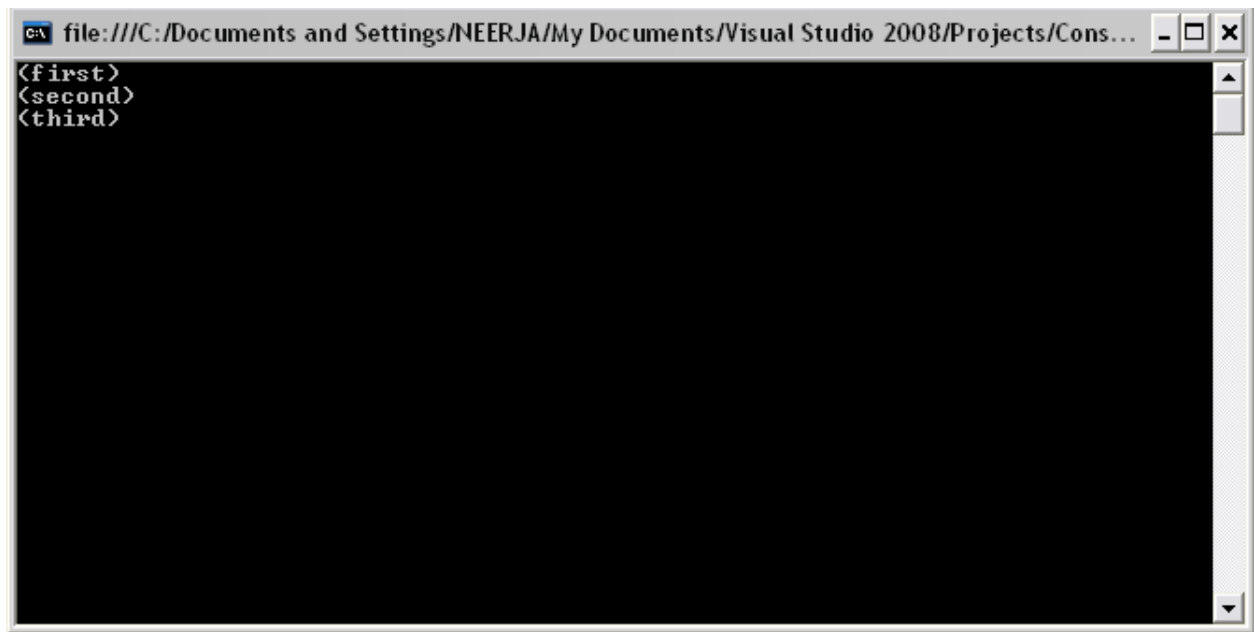
namespace ConsoleApplication11
{
    class CallTo
    {
        public string s;
        public void call(string s1)
        {
            s = s1;
            Console.Write("{0}", s);
            Thread.Sleep(1000);
            Console.WriteLine("");
        }
    }
    class CallFrom
    {
        string str;
        CallTo callto;
        public Thread t;
        public CallFrom(CallTo ct, string s)
        {
            callto = ct;
            str = s;
            t = new Thread(new ThreadStart(this.callit));
            t.Start();
        }
        public void callit()
        {
            callto.call(str);
        }
    }
    public class Thread3
    {
        static void Main(string[] args)
        {
            CallTo ct = new CallTo();
            CallFrom cf1 = new CallFrom(ct, "first");
            CallFrom cf2 = new CallFrom(ct, "second");
            CallFrom cf3 = new CallFrom(ct, "third");
            cf1.t.Join();
            cf2.t.Join();
            cf3.t.Join();
            Console.ReadLine();
        }
    }
}

```




```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading;  
  
namespace ConsoleApplication11  
{  
    class CallTo  
    {  
        public string s;  
        public void call(string s1)  
        {  
            Monitor.Enter(this);  
            s = s1;  
            Console.Write("{0}", s);  
            Thread.Sleep(1000);  
            Console.WriteLine("");  
            Monitor.Exit(this);  
        }  
    }  
    class CallFrom  
    {  
        string str;  
        CallTo callto;  
        public Thread t;  
        public CallFrom(CallTo ct, string s)  
        {  
            callto = ct;  
            str = s;  
        }  
    }  
}
```

```
        t = new Thread(new ThreadStart(this.callit));
        t.Start();
    }
    public void callit()
    {
        callto.call(str);
    }
}
public class Thread3
{
    static void Main(string[] args)
    {
        CallTo ct = new CallTo();
        CallFrom cf1 = new CallFrom(ct, "first");
        CallFrom cf2 = new CallFrom(ct, "second");
        CallFrom cf3 = new CallFrom(ct, "third");
        cf1.t.Join();
        cf2.t.Join();
        cf3.t.Join();
        Console.ReadLine();
    }
}
```



```
file:///C:/Documents and Settings/NEERJA/My Documents/Visual Studio 2008/Projects/Cons...
<first>
<second>
<third>
```


BLOCK- 4

U n i t

14

Namespace

Interface:

Interfaces describe a group of related functionalities that can belong to any class or struct. Interfaces can consist of methods, properties, events, indexers, or any combination of those four member types. An **Interface** is a *reference type* and it contains only *abstract members*, so sometime it also called *pure abstract class*. An interface contains only declaration for its members. Any implementation must be placed in class that inherited them. An interface can't contain constants, data fields, constructors, destructors and static members. All the member declarations inside interface are implicitly public.

To implement an interface member, the corresponding member on the class must be public, non-static, and have the same name and signature as the interface member. Properties and indexers on a class can define extra accessors for a property or indexer defined on an interface.

Classes and structs can inherit from more than one interface. An interface can itself inherit from multiple interfaces.

Defining an interface:

interface IDemoInterface

```
{  
    void MethodDemo();  
}
```

We define an interface by using keyword interface. Above an interface name is IDemoInterface. A common naming convention is to prefix all interface names with a capital "I".

Example:

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

```
namespace DemoInterface  
{
```

```
    interface IDemoInterface // defining an interface  
    {  
        void MethodDemo();  
    }
```

```
        class MyClass : IDemoInterface // implementing an interface in the  
class  
    {
```

```

static void Main(string[] args)
{
    MyClass mco = new MyClass ();
    mco.MethodDemo();
}

public void MethodDemo()
{
    Console.WriteLine("An example of Interface.");
    Console.ReadLine();
}
}

```

OUTPUT:

An example of Interface.

Explanation:

In above example we define an interface named IDemoInterface which contains a method named MethodDemo().

Now, a class named MyClass inherits the interface IDemoInterface using : symbol.

```
class MyClass : IDemoInterface
```

The class MyClass implement the method MethodDemo() which is defined by the interface IDemoInterface. Then we create an object of class MyClass named *mco* and called the method MehtodDemo().

So, the above description defines the actual story of an interface.

Multiple Inheritance using C# interfaces:

Multiple inheritance is not allowed in C#. So, this problem can solve by using interface. This can be done using child class that inherits from any number of c# interfaces.

```

using System;
using System.Collections.Generic;
using System.Text;

namespace DemoInterface
{
    interface IParentInterafce
    {
        void MethodParentDemo();
    }
    interface IDemoInterface
    {
        void MethodDemo();
    }

    class MyClass : IParentInterafce, IDemoInterface
    {
        public static void Main(string[] args)
        {
            MyClass mco = new MyClass ();
            mco.MethodDemo();
            mco.MethodParentDemo();
        }

        public void MethodDemo()
        {

```

```

        Console.WriteLine("Implementing IDemoInterface");
    }
    public void MethodParentDemo()
    {
        Console.WriteLine("Implementing IParentInterafce");
        Console.ReadLine();
    }
}
}

```

OUTPUT:

```

Implementing IDemoInterface
Implementing IParentInterafce

```

Explanation:

In above example we have defined two interfaces named IParentInterafce and IDemoInterface. Then we have inherited both interfaces in class named MyClass.

So, by doing this we can resolve the problem of multiple inheritance.

An interface can inherit other interface as below:

```

interface IParentInterafce
{
    void MethodParentDemo();
}
interface IDemoInterface :IParentInterafce
{
    void MethodDemo();
}

```

Implementation of both interface will same as in above exaple.

```

//using namespace

```

```

using maths;

```

```

using System;

```

```

class test

```

```

{

```

```

    public static void Main(string[] args)
    {

```

```

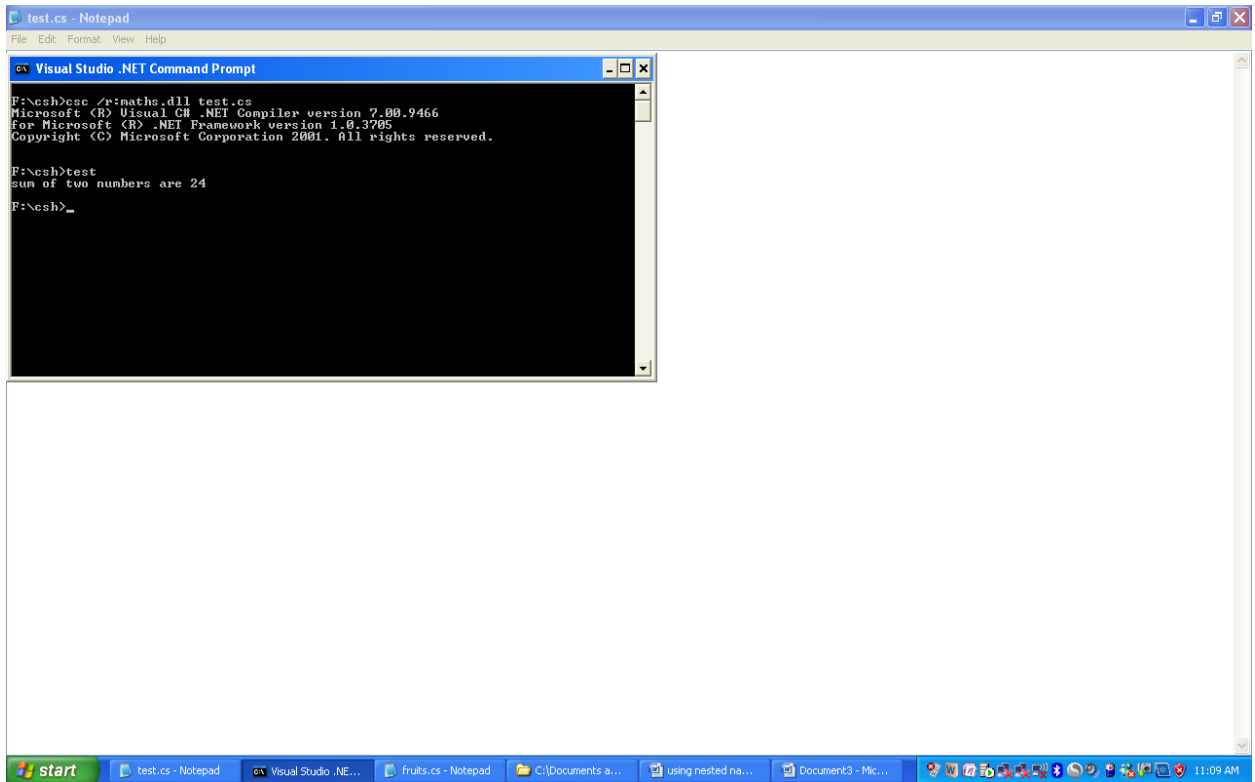
        add a=new add();
        a.sum();

```

```

    }
}

```



15

Graphics

Persistent Graphics

An important point to note before proceeding with this chapter is that simply creating a Graphics Object for a component and then drawing on that component does not create persistent graphics. In fact what will happen is that as soon as the window is minimized or obscured by another window the graphics will be erased.

For this reason, steps need to be taken to ensure that any graphics are persistent. Two mechanisms are available for achieving this. One is to repeatedly perform the drawing in the *Paint()* event handler of the control (which is triggered whenever the component needs to be redrawn), or to perform the drawing on a bitmap image in memory and then transfer that image to the component whenever the *Paint()* event is triggered. We will look at redrawing the graphics in the *Paint()* event in this chapter and Using Bitmaps for Persistent Graphics in C# in the next chapter.

Creating a Graphics Object

The first step in this tutorial is to create a new Visual Studio project called CSharpGraphics. With the new project created select the Form in the design area and click on the lightning bolt at the top of the *Properties* panel to list the events available for the Form. Double click the *Paint* event to display the code editing page.

Graphics Objects are created by calling the *CreateGraphics()* method of the component on which the drawing is to be performed. For example, a Graphics Object can be created on our Form called Form1 by calling *CreateGraphics()* method as follows in the *Paint()* method:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    System.Drawing.Graphics graphicsObj;

    graphicsObj = this.CreateGraphics();
}
```



```
}
```

Now that we have a Graphic Object we need a Pen with which to draw.

Creating a Pen In C#

A Graphics Object is of little use without a Pen object with which to draw (much as a sheet of paper is no good without a pen or pencil). A Pen object may be quite easily created as follows:

```
Pen variable_name = new Pen (color, width);
```

where *variable_name* is the name to be assigned to the Pen object, *color* is the color of the pen and *width* is the width of the lines to be drawn by the pen.

For example, we can create red pen that is 5 pixels wide as follows:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    System.Drawing.Graphics graphicsObj;

    graphicsObj = this.CreateGraphics();

    Pen myPen = new Pen(System.Drawing.Color.Red, 5);
}
```

Once a Pen object has been created other properties may be changed. For example, the `DashStyle` property can be modified to change the style of line (i.e `Dash`, `DashDot`, `DashDotDot`, `Dot`, `Solid` or `Custom`). Properties such as the color and width may similarly be changed after a Pen has been created:

```
myPen.DashStyle =
System.Drawing.Drawing2D.DashStyle.DashDotDot;

myPen.Color = System.Drawing.Color.RoyalBlue;

myPen.Width = 3;
```

Now that we have a `Paint()` event handler, a Graphics Object and Pen we can now begin to draw.

Drawing Lines in C#

Lines are drawn in C# using the `DrawLine()` method of the Graphics Object. This method takes a pre-instantiated Pen object and two sets of x and y co-ordinates (the start and end points of the line) as arguments. For example, to draw a line from co-ordinates (20, 20) to (200, 210) on our sample form:

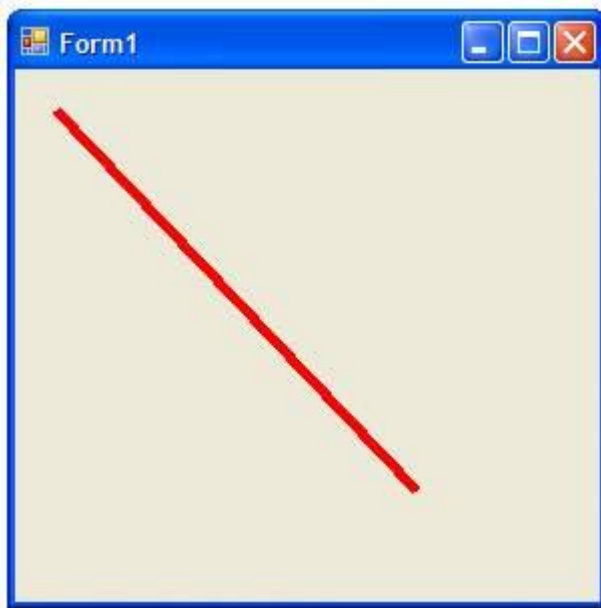
```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    System.Drawing.Graphics graphicsObj;

    graphicsObj = this.CreateGraphics();

    Pen myPen = new Pen(System.Drawing.Color.Red, 5);

    graphicsObj.DrawLine(myPen, 20, 20, 200, 210);
}
```

The above code, when compiled and executed will result in the form appearing as follows:



Drawing Squares and Rectangles in C#

For the purposes of drawing rectangles and squares in C# the GraphicsObject provides the *DrawRectangle()* method. There are two ways to use the *DrawRectangle()* method. One is to pass through a *Rectangle* object and Pen and the other is to create an instance of a Rectangle object and pass that through along with the Pen. We will begin by looking at drawing a rectangle without a pre-created Rectangle object. The syntax for this is:

```
graphicsobj.DrawRectangle(pen, x, y, width, height);
```

The alternative is to pass through a Rectangle object in place of the co-ordinates and dimensions. The syntax for creating a Rectangle object in C# is as follows:

```
Rectangle rectangleObj = new Rectangle (x, y, width, height);
```

Once a Rectangle object has been instantiated the syntax to call *DrawRectangle()* is as follows:

graphicsobj.DrawRectangle(pen, x, y, rectangleobj);

The following example creates a Rectangle which is then used as an argument to *DrawRectangle()*:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    System.Drawing.Graphics graphicsObj;

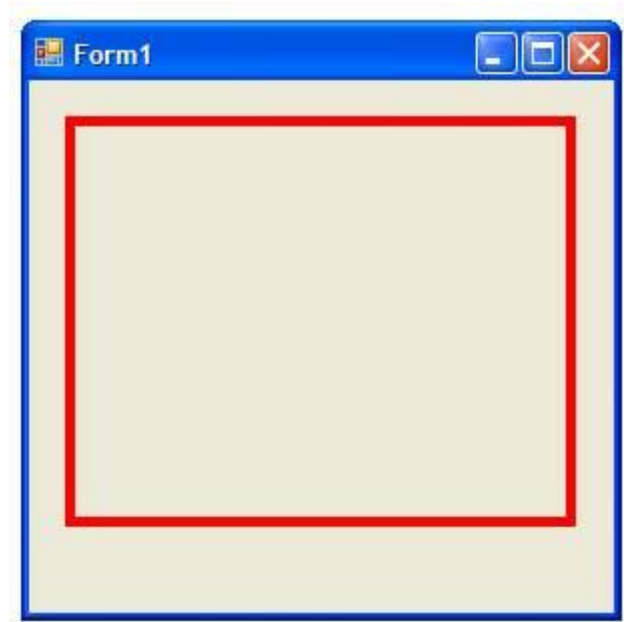
    graphicsObj = this.CreateGraphics();

    Pen myPen = new Pen(System.Drawing.Color.Red, 5);

    Rectangle myRectangle = new Rectangle(20, 20, 250, 200);

    graphicsObj.DrawRectangle(myPen, myRectangle);
}
```

When an application containing the above code is compiled and executed the following graphics will appear in the form:



If multiple rectangles of different shapes need to be drawn it is not necessary to create a new Rectangle object for each call to the *DrawRectangle()* method. Instead the shape of an existing Rectangle object may be altered by calling the *Inflate()* method of the Rectangle class. This method accepts two arguments, the amount by which the width is to be changed and the amount by which the height is to be changed. If a dimension is to be left unchanged 0 should be passed through as the change value.

To reduce a dimension pass through the negative amount by which the dimension is to be changed:

```
Rectangle myRectangle = new Rectangle(20, 20, 250, 200);

myRectangle.Inflate(10, -20); Increase width by 10. Reduce height
my 20
```

Drawing Ellipses and Circles in C#

Ellipses and circles are drawn in C# using the *DrawEllipse()* method of the *GraphicsObject* class. The size of the shape to be drawn is defined by specifying a rectangle into which the shape must fit. As with the *DrawRectangle()* method, there are two ways to use the *DrawEllipse()* method. One is to pass through a *Rectangle* object and *Pen* and the other is to create an instance of a *Rectangle* object and pass that through along with the *Pen*.

To draw an ellipse without first creating a *Rectangle* object use the following syntax:

```
graphicsObj.DrawEllipse(pen, x, y, width, height);
```

The alternative is to pass through a *Rectangle* object in place of the co-ordinates and dimensions. The syntax for creating a *Rectangle* object in C# is as follows:

```
Rectangle rectangleObj = new Rectangle (x, y, width, height);
```

Once a *Rectangle* object has been instantiated the syntax to call *DrawRectangle()* is as follows:

```
graphicsObj.DrawEllipse(pen, x, y, rectangleobj);
```

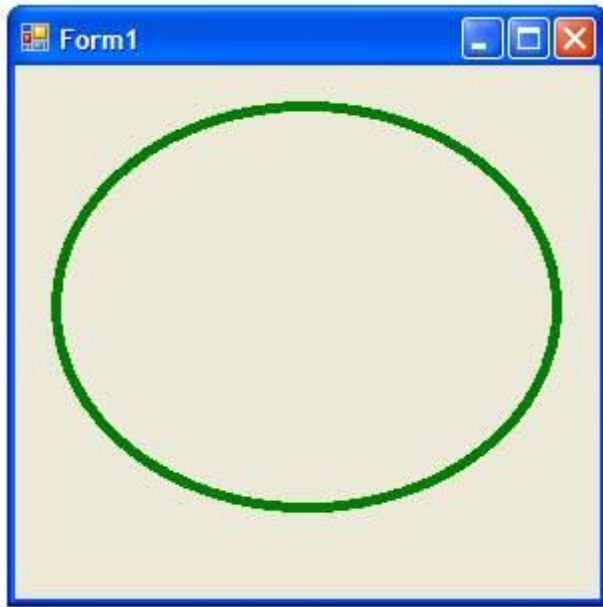
The following example creates a *Rectangle* which is then used as an argument to *DrawEllipse()*:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    System.Drawing.Graphics graphicsObj;

    graphicsObj = this.CreateGraphics();

    Pen myPen = new Pen(System.Drawing.Color.Green, 5);
    Rectangle myRectangle = new Rectangle(20, 20, 250, 200);
    graphicsObj.DrawEllipse(myPen, myRectangle);
}
```

When compiled and executed the above code creates the following graphics output on the form:



Drawing Text with C#

Text is drawn onto a Graphics Object using the *DrawText()* method. The syntax for this method is as follows:

```
graphicsobj.DrawString(string, font, brush, x, y);
```

The *string* argument specifies the text to be drawn. Font defines the font to be used to display the text and requires the creation of a *Font* object. The *brush* object is similar to the *Pen* object used to draw shapes with the exception that it specifies a fill pattern. Finally, the x and y values specify the top left hand corner of the text.

In order to create a Font object a font size, font family and font style may be specified. For example to create a Helvetica, 40 point Italic font:

```
Font myFont = new System.Drawing.Font("Helvetica", 40,  
FontStyle.Italic);
```

A brush object is created by specifying by calling the appropriate constructor for the brush type and specifying a color:

```
Brush myBrush = new SolidBrush(System.Drawing.Color.Red);
```

Having created the necessary objects we can incorporate these into our example C# application to draw some text:

```
private void Form1_Paint(object sender, PaintEventArgs e)  
{  
    System.Drawing.Graphics graphicsObj;
```

```
graphicsObj = this.CreateGraphics();

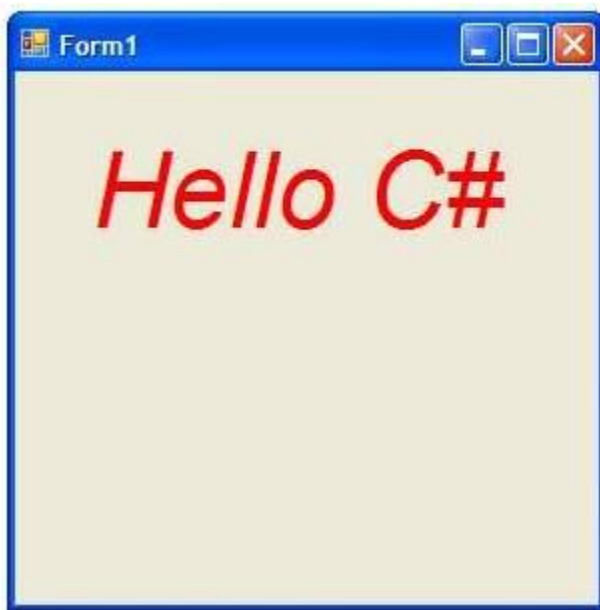
Font myFont = new System.Drawing.Font("Helvetica", 40,
FontStyle.Italic);

Brush myBrush = new SolidBrush(System.Drawing.Color.Red);

graphicsObj.DrawString("Hello C#", myFont, myBrush, 30,
30);

}
```

The above code, when compiled and run, will output the following text onto the form:



All drawing in GDI+ takes place upon the **Graphics** object and there are several contexts in which you'll find one. During a Paint cycle the **Graphics** object will be provided in a **PaintEventArgs** object that is handed to your code in the **OnPaint** and **OnPaintBackground** methods of a Windows Forms control. This same event argument is passed to handlers that service the **Paint** event raised by the **OnPaint** methods. When printing, the **PrintPageEventArgs** provided in a **PrintPage** event will contain a **Graphics** object for the printer and you can obtain a **Graphics** object for certain types of image so that you can paint directly on an image in memory as if it were the screen.

Obtaining the Graphics object.

When you're writing programs that place graphics on screen the graphics object will be handed to you wrapped up in a **PaintEventArgs** object. There are two ways in which your code can get hold of the Graphics object. You can override the protected OnPaint or OnPaintBackground methods or you can add a handler to the Paint event. In all these cases the Graphics object is passed in a PaintEventArgs object. Listing 1 shows the various methods.

Listing 1

```
protected override void OnPaint(PaintEventArgs e)
{
    //get the Graphics object from the PaintEventArgs
    Graphics g=e.Graphics;
    g.DrawLine(...);

    //or use it directly
    e.Graphics.DrawLine(...);

    //Remember to call the base class or the Paint event won't fire
    base.OnPaint (e);
}

//This is the Paint event handler
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
    //get the Graphics object from the PaintEventArgs
    Graphics g=e.Graphics;
    g.DrawLine(...);

    //or use it directly
    e.Graphics.DrawLine(...);
}
```

You can see in Listing 1 that you can get a reference to the **Graphics** object and save that for use in the code or use it directly from the **PaintEventArgs**. Remember *not* to save the **Graphics** object outside of the scope of the method.

What to do with it when you have it.

When your code has access to a Graphics object there are a number of things you can do. They fall into five general categories.

- **Stroke a shape.** Shapes such as rectangles, ellipses and lines are drawn using a **Pen** object. The pen can have different thickness or colour and have many other attributes which you will see in another article.

- **Fill a shape.** Shapes can be filled with a Brush object. Brushes have many complex settings in GDI+ but they all basically fill an area with colour.
- **Draw a string.** Text can be placed on the Graphics surface using the DrawString method.
- **Draw an image.** Images can be drawn in any scale using one of the DrawImage methods.
- **Modify the Graphics object.** There are many methods that change the way the Graphics object performs. You can change the quality of graphics and have high-quality graphics at the expense of speed. You can change the way the graphics are output to create rotation, zooming or distortion effects.

To demonstrate the use of the Graphics object, the following listing shows the paint handler from a form that demonstrates stroking and filling of shapes. Figure 1 shows this simple application at work.

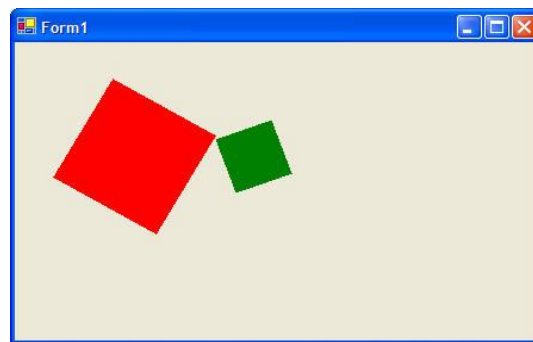


Figure 1. Stroking and filling.

```
protected override void OnPaint(PaintEventArgs e)
{
    //Get the Graphics object
    Graphics g=e.Graphics;

    //Draw a line
    g.DrawLine(Pens.Red,10,5,110,15);

    //Draw an ellipse
    g.DrawEllipse(Pens.Blue,10,20,110,45);

    //Draw a rectangle
    g.DrawRectangle(Pens.Green,10,70,110,45);

    //Fill an ellipse
    g.FillEllipse(Brushes.Blue,130,20,110,45);

    //Fill a rectangle
    g.FillRectangle(Brushes.Green,130,70,110,45);
}
```



```
base.OnPaint (e);  
}
```

Before you can draw lines and shapes, render text, or display and manipulate images with GDI+, you need to create a [Graphics](#) object. The [Graphics](#) object represents a GDI+ drawing surface, and is the object that is used to create graphical images.

There are two steps in working with graphics:

1. Creating a [Graphics](#) object.
2. Using the [Graphics](#) object to draw lines and shapes, render text, or display and manipulate images.

Creating a Graphics Object

A graphics object can be created in a variety of ways.

To create a graphics object

- Receive a reference to a graphics object as part of the [PaintEventArgs](#) in the [Paint](#) event of a form or control. This is usually how you obtain a reference to a graphics object when creating painting code for a control. Similarly, you can also obtain a graphics object as a property of the [PrintPageEventArgs](#) when handling the [PrintPage](#) event for a [PrintDocument](#).

-or-

- Call the [CreateGraphics](#) method of a control or form to obtain a reference to a [Graphics](#) object that represents the drawing surface of that control or form. Use this method if you want to draw on a form or control that already exists.

-or-

- Create a [Graphics](#) object from any object that inherits from [Image](#). This approach is useful when you want to alter an already existing image.

The following sections give details about each of these processes.

PaintEventArgs in the Paint Event Handler

When programming the [PaintEventHandler](#) for controls or the [PrintPage](#) for a [PrintDocument](#), a graphics object is provided as one of the properties of [PaintEventArgs](#) or [PrintPageEventArgs](#).

To obtain a reference to a Graphics object from the PaintEventArgs in the Paint event

1. Declare the [Graphics](#) object.
2. Assign the variable to refer to the [Graphics](#) object passed as part of the [PaintEventArgs](#).
3. Insert code to paint the form or control.

The following example shows how to reference a [Graphics](#) object from the [PaintEventArgs](#) in the [Paint](#) event:

C#

[C++](#)

[VB](#)

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs pe)
{
    // Declares the Graphics object and sets it to the Graphics
    object
    // supplied in the PaintEventArgs.
    Graphics g = pe.Graphics;
    // Insert code to paint the form here.
}
```

CreateGraphics Method

You can also use the [CreateGraphics](#) method of a control or form to obtain a reference to a [Graphics](#) object that represents the drawing surface of that control or form.

To create a Graphics object with the CreateGraphics method

- Call the [CreateGraphics](#) method of the form or control upon which you want to render graphics.

VB

```
Dim g as Graphics
' Sets g to a Graphics object representing the drawing surface of
the
' control or form g is a member of.
g = Me.CreateGraphics
```

C#

```
Graphics g;  
// Sets g to a graphics object representing the drawing surface  
of the  
// control or form g is a member of.  
g = this.CreateGraphics();
```

C++

```
Graphics ^ g;  
// Sets g to a graphics object representing the drawing surface  
of the  
// control or form g is a member of.  
g = this->CreateGraphics();
```

Create from an Image Object

Additionally, you can create a graphics object from any object that derives from the [Image](#) class.

To create a Graphics object from an Image

- Call the [Graphics.FromImage](#) method, supplying the name of the Image variable from which you want to create a [Graphics](#) object.

The following example shows how to use a [Bitmap](#) object:

C#

[C++](#)

[VB](#)

```
Bitmap myBitmap = new Bitmap(@"C:\Documents and  
Settings\Joe\Pics\myPic.bmp");  
Graphics g = Graphics.FromImage(myBitmap);
```

Note

You can only create [Graphics](#) objects from nonindexed .bmp files, such as 16-bit, 24-bit, and 32-bit .bmp files. Each pixel of nonindexed .bmp files holds a color, in contrast to pixels of indexed .bmp files, which hold an index to a color table.

•

Drawing and Manipulating Shapes and Images

After it is created, a [Graphics](#) object may be used to draw lines and shapes, render text, or display and manipulate images. The principal objects that are used with the [Graphics](#) object are:

- The [Pen](#) class—Used for drawing lines, outlining shapes, or rendering other geometric representations.
- The [Brush](#) class—Used for filling areas of graphics, such as filled shapes, images, or text.
- The [Font](#) class—Provides a description of what shapes to use when rendering text.
- The [Color](#) structure—Represents the different colors to display.

To use the Graphics object you have created

- Work with the appropriate object listed above to draw what you need.

For more information, see the following topics:

To render	See
Lines	How to: Draw a Line on a Windows Form
Shapes	How to: Draw an Outlined Shape
Text	How to: Draw Text on a Windows Form
Images	How to: Render Images with GDI+

Like Java, C# provides us with a rich set of classes, methods and events for developing applications with graphical capabilities. Since there is not much theory involved, we can straight away jump to an interesting example (Listing - 1), which prints "Welcome to C#" on a form. Relevant explanations are shown as comments:

Listing - 1

```
using System;
using System.Windows.Forms;
using System.Drawing;
public class Hello:Form
{
    public Hello()
    {
        this.Paint += new PaintEventHandler(f1_paint);
    }
    private void f1_paint(object sender,PaintEventArgs e)
    {
```

```

Graphics g = e.Graphics;
g.DrawString("Hello C#",new Font("Verdana",20),
new SolidBrush(Color.Tomato),40,40);
g.DrawRectangle(new Pen(Color.Pink,3),20,20,150,100);
}
public static void Main()
{
Application.Run(new Hello());
}
// End of class
}

```

The method DrawString() takes four arguments as shown in the above example. Every method in the Graphics class have to be accessed by creating an object of that class. You can easily update the above program to render other graphical shapes like Rectangle, Ellipse etc. All you have to do is to apply the relevant methods appropriately.

Changing the Unit of Measurement

As you may know the default Graphics unit is Pixel. By applying the PageUnit property, you can change the unit of measurement to Inch, Millimeter etc as shown below:

```

Graphics g = e.Graphics;
g.PageUnit = GraphicsUnit.Inch

```

Working with ColorDialog Box

If you have ever done Visual Basic Programming, you should be aware of predefined dialog boxes like ColorDialog, FontDialog etc. In C#, you or your user can choose a color by applying the ColorDialog class appropriately. Firstly you have to create an object of ColorDialog class as shown below:

```

ColorDialog cd = new ColorDialog();

```

Using the above object call ShowDialog() method to display the color dialog box. Finally invoke the Color property and apply it appropriately as shown in Listing - 2:

Listing - 2

```

using System;
using System.Drawing;
using System.Windows.Forms;
public class Clr:Form
{
Button b1 = new Button();
TextBox tb = new TextBox();
ColorDialog clg = new ColorDialog();
public Clr()
{
b1.Click += new EventHandler(b1_click);
b1.Text = "OK";
}
}

```

```

tb.Location = new Point(50,50);
this.Controls.Add(b1);
this.Controls.Add(tb);
}
public void b1_click(object sender, EventArgs e)
{
    clg.ShowDialog();
    tb.BackColor = clg.Color;
}
public static void Main()
{
    Application.Run(new Clr());
}
// End of class
}

```

Here the background color of the form will change as you select a color from the dialog box.

Working with FontDialog Box

You can easily create a Font selection dialog box by following the same steps as in the previous listing and by affecting some minor changes. Listing - 3 shown below examines the usage of this useful tool:

Listing - 3

```

using System;
using System.Drawing;
using System.Windows.Forms;
public class Fonts:Form
{
    Button b1 = new Button();
    TextBox tb = new TextBox();
    FontDialog flg = new FontDialog();
    public Fonts()
    {
        b1.Click += new EventHandler(b1_click);
        b1.Text = "OK";
        tb.Location = new Point(50,50);
        this.Controls.Add(b1);
        this.Controls.Add(tb);
    }
    public void b1_click(object sender, EventArgs e)
    {
        clg.ShowDialog();
        tb.FontName = flg.Font;
    }
    public static void Main()
    {
        Application.Run(new Fonts());
    }
// End of class
}

```

```
}
```

Using System.Drawing.Drawing2D Namespace

The System.Drawing.Drawing2D namespace provides advanced techniques for manipulating Pen and Brush objects. For example, you can change the look and feel of lines by applying the values of DashStyle Enumerator (like Dash, DashDot etc).

Also by making use of various Brush classes like SolidBrush, HatchStyleBrush etc you can modify the appearance of filled shapes. For instance, a rectangle can be filled with Vertical and Horizontal lines. As already examined a normal shape (Using DrawXXX() method) accepts a Pen class argument besides the floating point values while a filled shape (Using FillXXX() methods) accepts a Brush class argument.

Working with Pen objects

Listing - 4 given below examines the usage of various DrawXXX() methods by making use of some properties in the System.Drawing.Drawing2D namespace:

Listing - 4

```
using System;
using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Drawing2D;
public class Drawgra:Form
{
    public Drawgra()
    {
        this.Text = "Illustrating DrawXXX() methods";
        this.Size = new Size(450,400);
        this.Paint += new PaintEventHandler(Draw_Graphics);
    }
    public void Draw_Graphics(object sender,PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        Pen penline = new Pen(Color.Red,5);
        Pen penellipse = new Pen(Color.Blue,5);
        Pen penpie = new Pen(Color.Tomato,3);
        Pen penpolygon = new Pen(Color.Maroon,4);
        /*DashStyle Enumeration values are Dash,
        DashDot,DashDotDot,Dot,Solid etc*/
        penline.DashStyle = DashStyle.Dash;
        g.DrawLine(penline,50,50,100,200);
        //Draws an Ellipse
        penellipse.DashStyle = DashStyle.DashDotDot;
        g.DrawEllipse(penellipse,15,15,50,50);
        //Draws a Pie
        penpie.DashStyle = DashStyle.Dot;
        g.DrawPie(penpie,90,80,140,40,120,100);
        //Draws a Polygon
        g.DrawPolygon(penpolygon,new Point[]{
            new Point(30,140),
```

```

new Point(270,250),
new Point(110,240),
new Point(200,170),
new Point(70,350),
new Point(50,200)});
}
public static void Main()
{
Application.Run(new Drawgra());
}
// End of class
}

```

Working with Brush objects

Brush class is used to fill the shapes with a given color, pattern or Image. There are four types of Brushes like SolidBrush (Default Brush), HatchStyleBrush, GradientBrush and TexturedBrush. The listings given below show the usage of each of these brushes by applying them in a FillXXX() method:

Using SolidBrush

Listing - 5

```

using System;
using System.Windows.Forms;
using System.Drawing;
public class Solidbru:Form
{
public Solidbru()
{
this.Text = "Using Solid Brushes";
this.Paint += new PaintEventHandler(Fill_Graph);
}
public void Fill_Graph(object sender,PaintEventArgs e)
{
Graphics g = e.Graphics;
//Creates a SolidBrush and fills the rectangle
SolidBrush sb = new SolidBrush(Color.Pink);
g.FillRectangle(sb,50,50,150,150);
}
public static void Main()
{
Application.Run(new Solidbru());
}
// End of class
}

```

Using HatchBrush

Listing - 6

```

using System;

```



```

using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Drawing2D;
public class Hatchbru:Form
{
    public Hatchbru()
    {
        this.Text = "Using Solid Brushes";
        this.Paint += new PaintEventHandler(Fill_Graph);
    }
    public void Fill_Graph(object sender,PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        //Creates a Hatch Style,Brush and fills the rectangle
        /*Various HatchStyle values are DiagonalCross,ForwardDiagonal,
        Horizontal, Vertical, Solid etc. */
        HatchStyle hs = HatchStyle.Cross;
        HatchBrush sb = new HatchBrush(hs,Color.Blue,Color.Red);
        g.FillRectangle(sb,50,50,150,150);
    }
    public static void Main()
    {
        Application.Run(new Hatchbru());
    }
    // End of class
}

```

Using GradientBrush

Listing - 7

```

using System;
using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Drawing2D;
public class Texturedbru:Form
{
    Brush bgbrush;
    public Texturedbru()
    {
        Image bgimage = new Bitmap("dotnet.gif");
        bgbrush = new TextureBrush(bgimage);
        this.Paint+=new PaintEventHandler(Text_bru);
    }
    public void Text_bru(object sender,PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        g.FillEllipse(bgbrush,50,50,500,300);
    }
    public static void Main()
    {
        Application.Run(new Texturedbru());
    }
}

```

```
// End of class  
}
```

Working with Images

You can easily insert images by following the procedure given below

1) Create an object of Bitmap class as shown below:

```
Image img = new Bitmap("image1.bmp");
```

2) Apply the above object in DrawImage() method

```
g.DrawImage(img,20,20,100,90);
```

Conclusion

In this article, I've examined two core namespaces System.Drawing and System.Drawing.Drawing2D by showing the usage of various methods and properties with the help of numerous listings.

The very important topic of displaying text is left until this late in the chapter because drawing text to the screen is (in general) more complex than drawing simple graphics. Although displaying a line or two of text when you're not that bothered about the appearance is extremely easy - it takes one single call to the Graphics.DrawString() method; if you are trying to display a document that has a fair amount of text in it, you rapidly find that things become a lot more complex. This is for two reasons:

- If you're concerned about getting the appearance just right, you must understand fonts. Whereas shape drawing requires brushes and pens as helper objects, the process of drawing text requires fonts as helper objects. And understanding fonts is not a trivial task.
- Text needs to be very carefully laid out in the window. Users generally expect words to follow naturally from one word to another and to be lined up with clear spaces in between. Doing that is harder than you might think. For starters, you don't usually know in advance how much space on the screen a word is going to take up. That has to be calculated (using the Graphics.MeasureString() method). Also, the space a word occupies on the screen affects where in the document every subsequent word is placed. If your application does any line wrapping, it'll need to assess word sizes carefully before deciding where to place the line break. The next time you run Microsoft Word, look carefully at the way Word is continually repositioning text as you do your work; there's a lot of complex processing going on there. Chances are that any GDI+ application you work on won't be nearly as complex as Word. However, if you need to display any text, many of the same considerations apply.

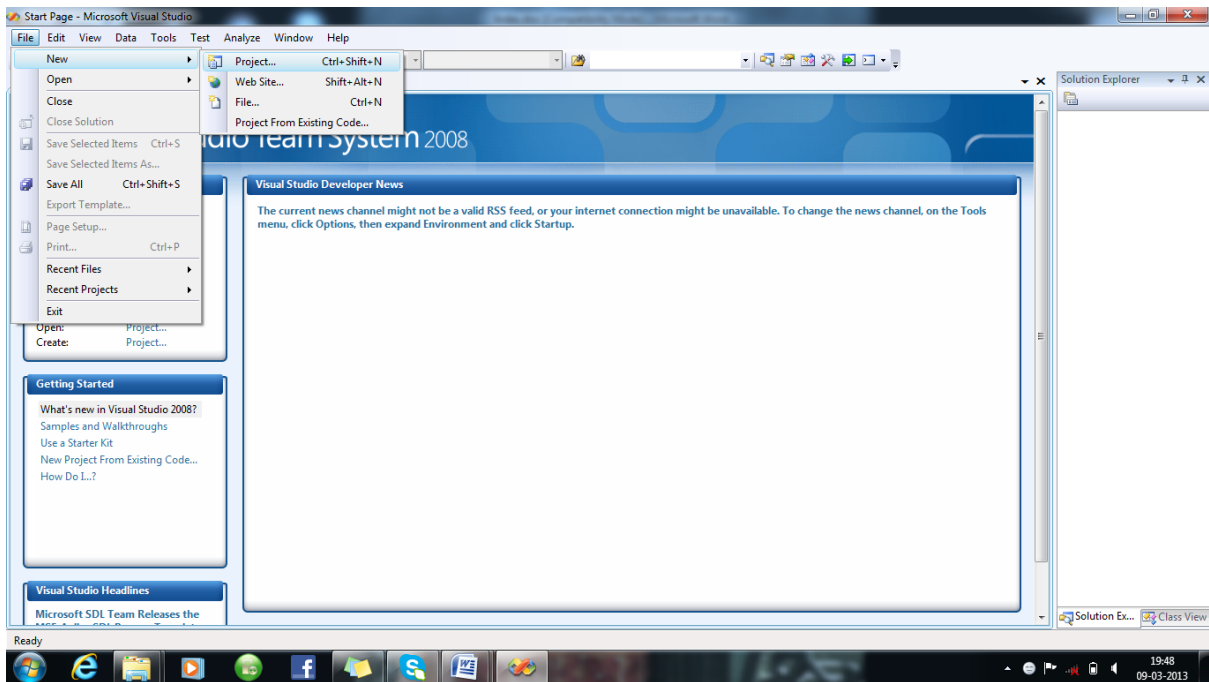
In short, high-quality text processing is tricky to get right. However, putting a line of text on the screen, assuming that you know the font and where you want it to go, is actually very simple. Therefore, the next section presents a quick example that shows how to

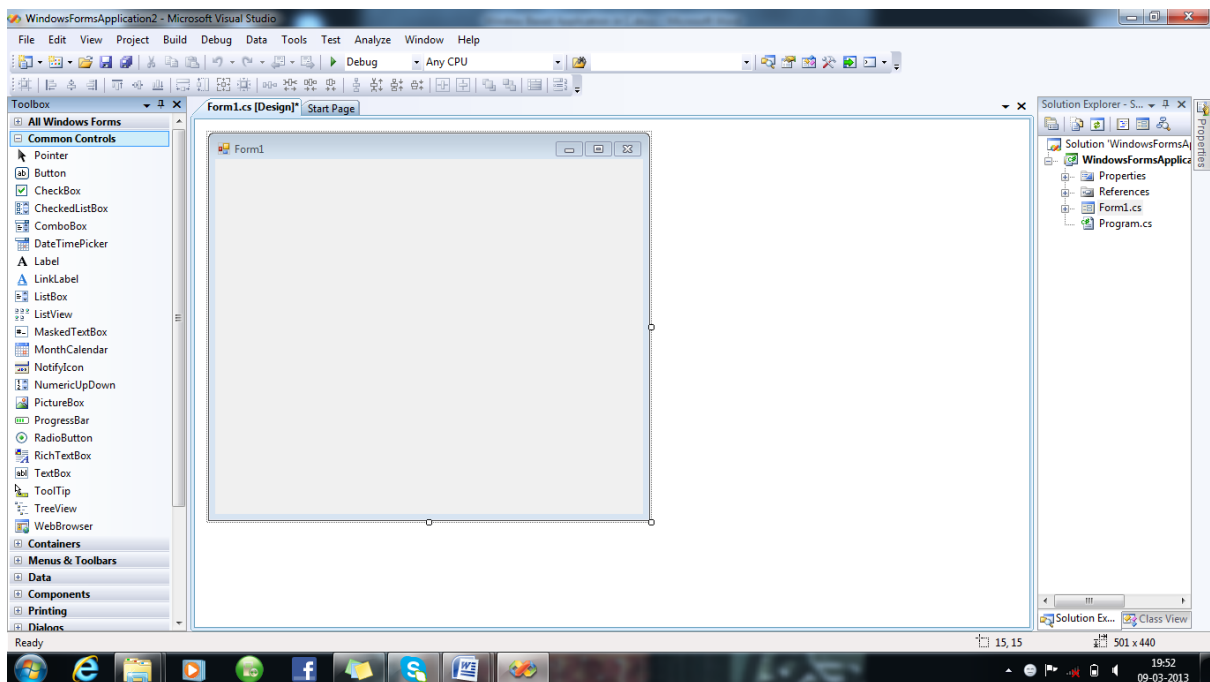
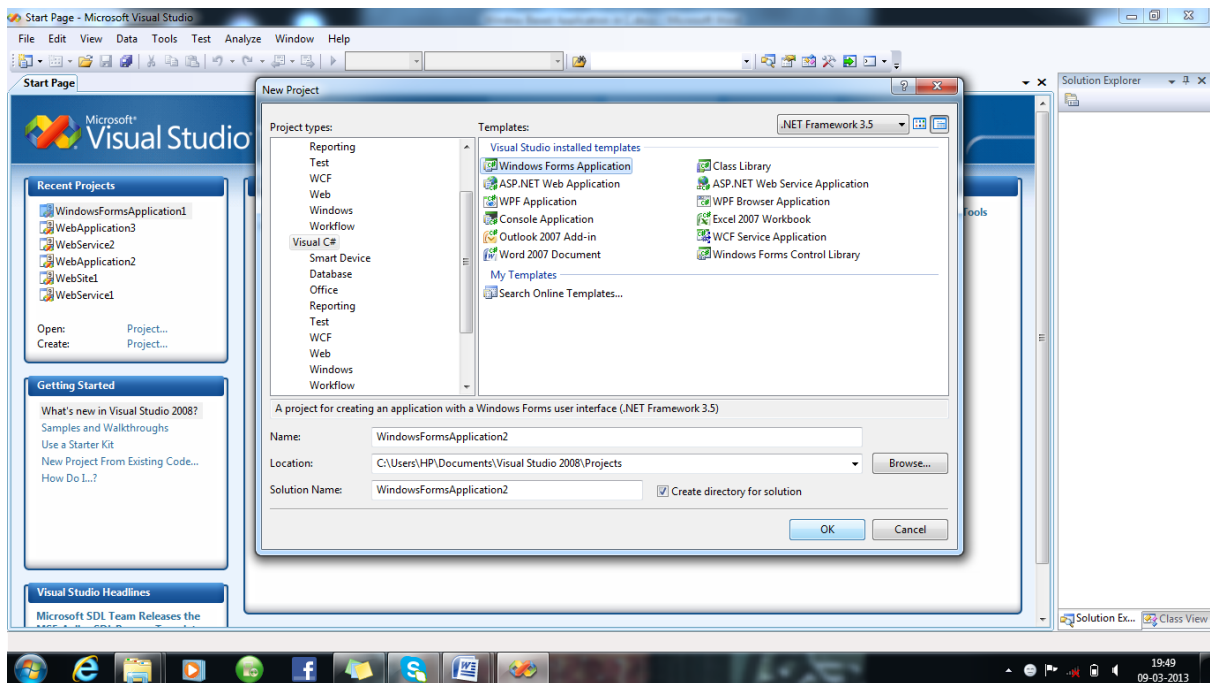
display some text, followed by a short review of the principles of fonts and font families and a more realistic (and involved) text-processing example, CapsEditor.

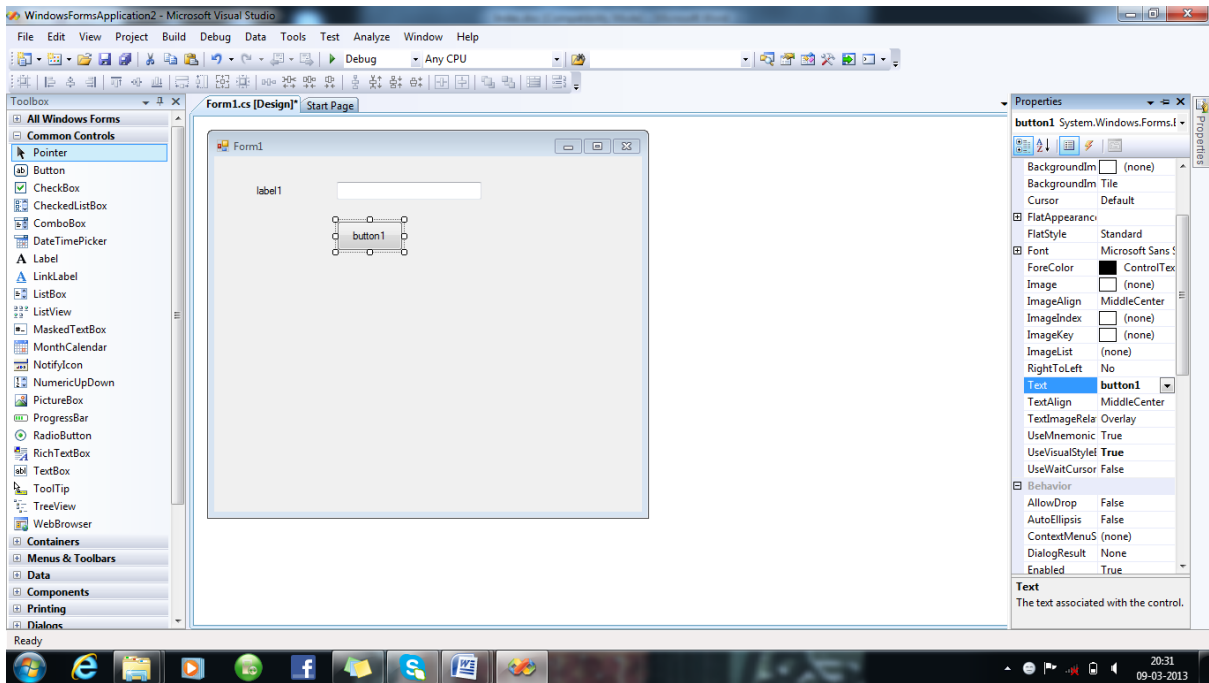
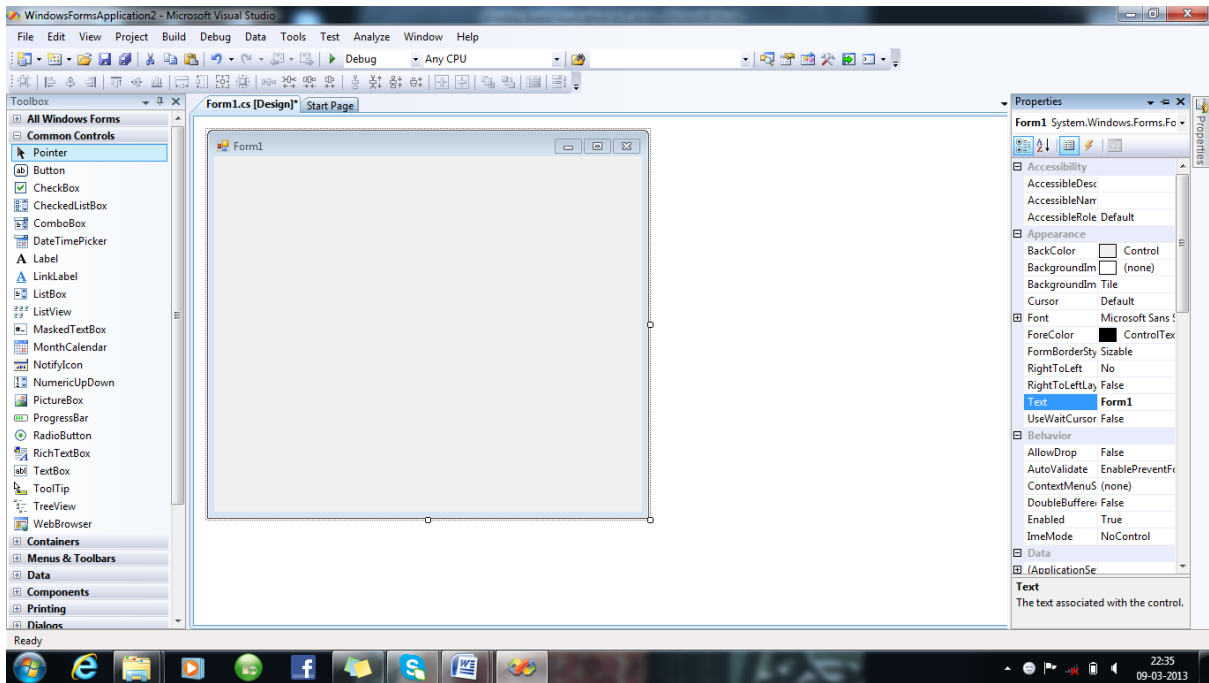
16

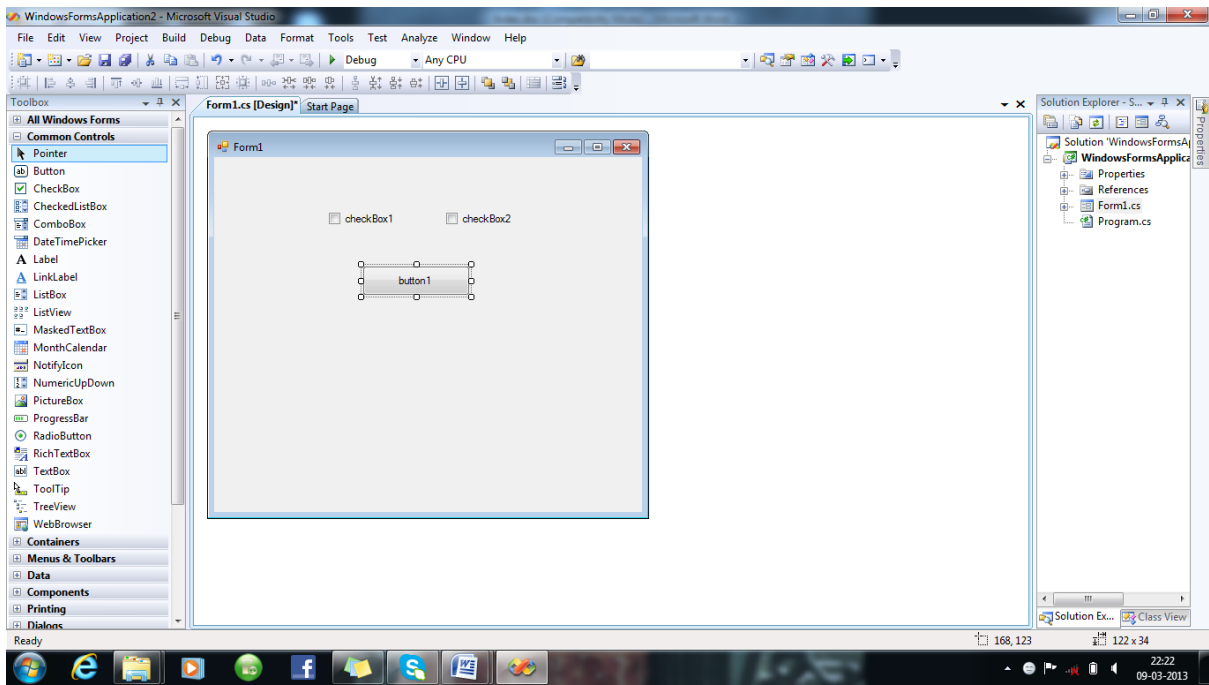
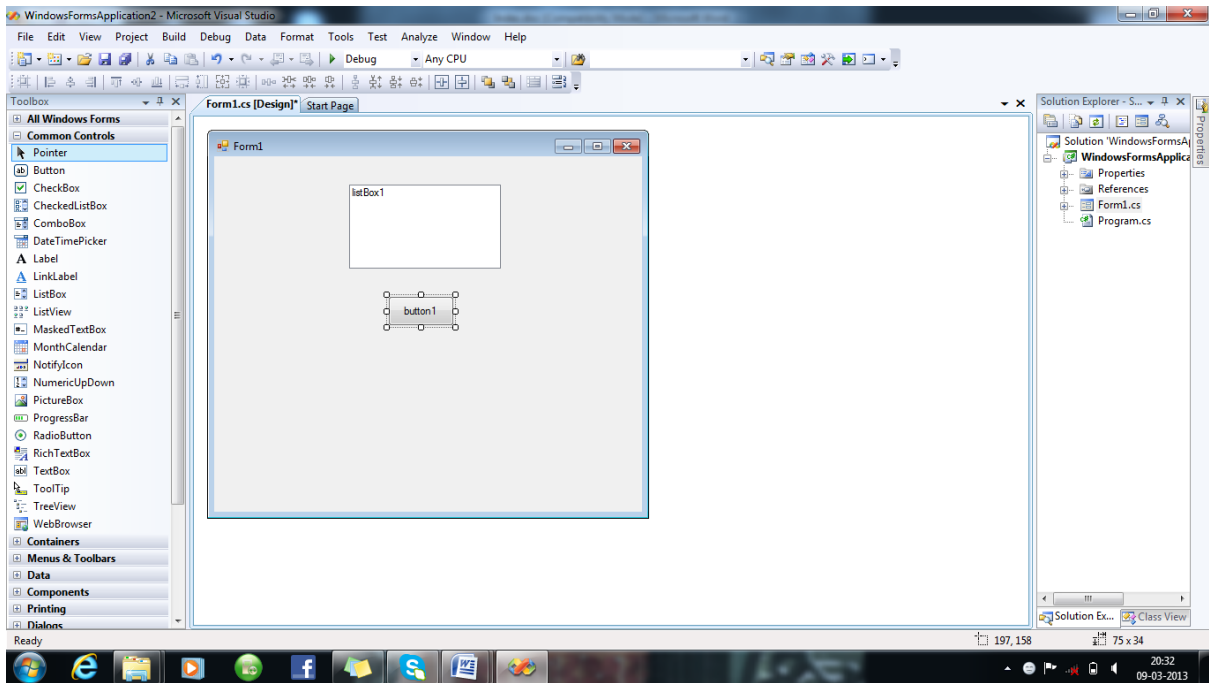
Window Based Application in C#

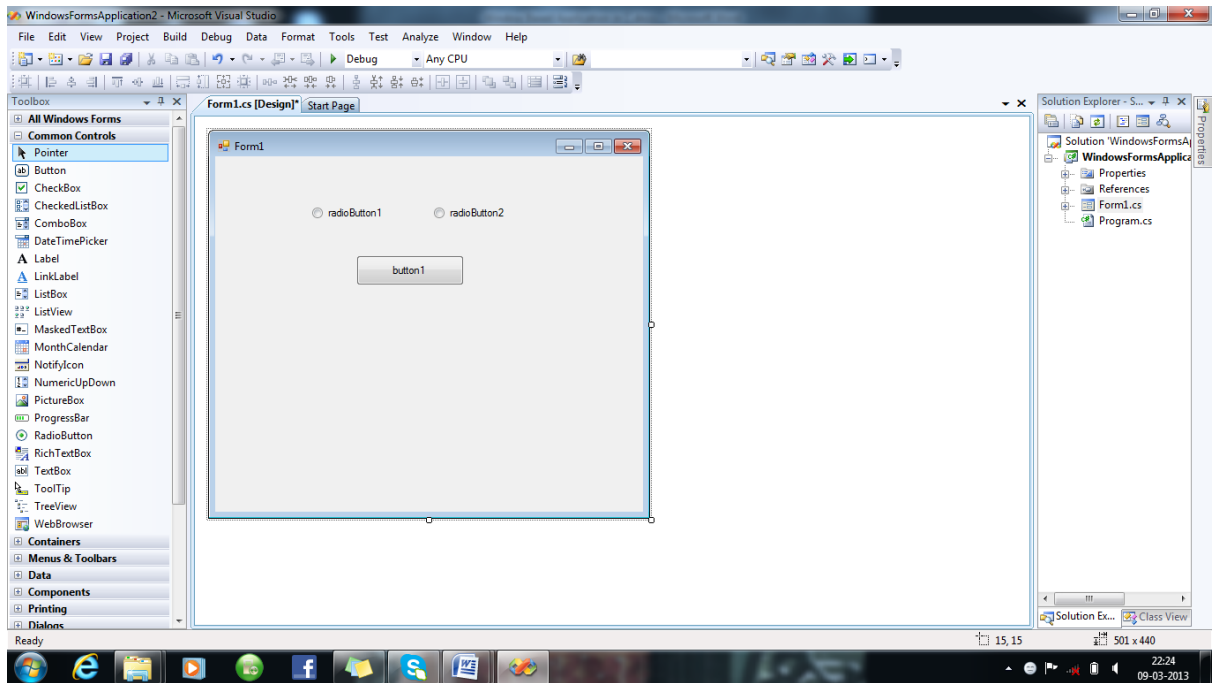
Window Based Application in C#:

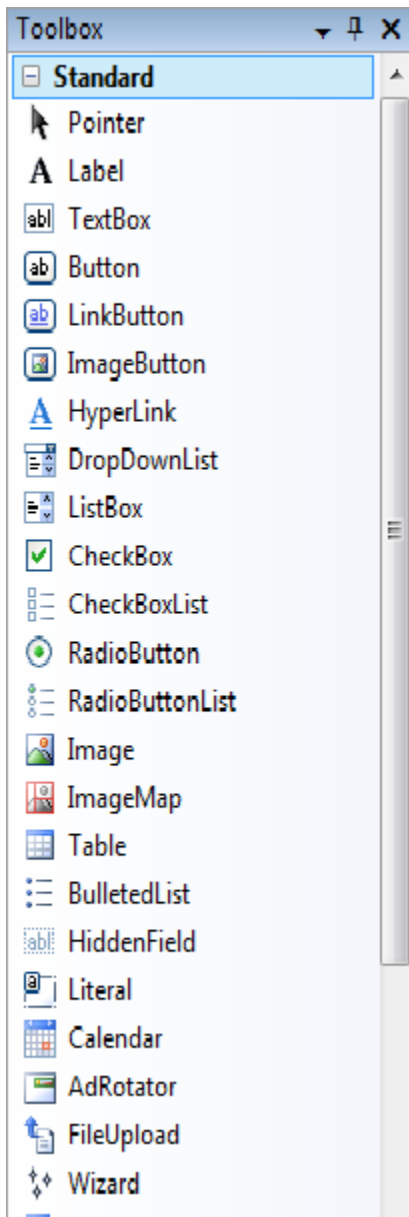












Properties

Form1 System.Windows.Forms.Fo

Accessibility

AccessibleDesc	
AccessibleName	
AccessibleRole	Default

Appearance

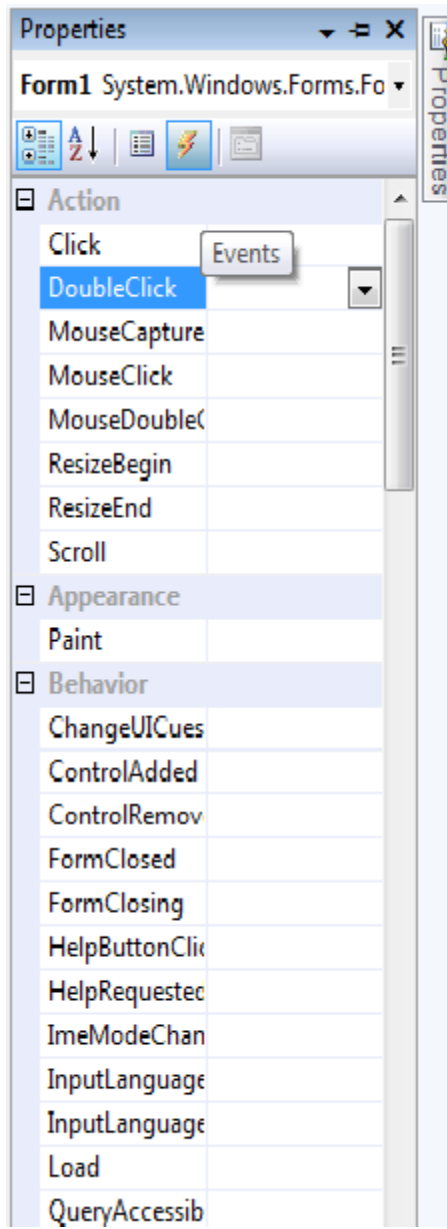
BackColor	<input type="color"/> Control
BackgroundImage	<input type="color"/> (none)
BackgroundImageTile	Tile
Cursor	Default

Font

Font	Microsoft Sans S
ForeColor	<input type="color"/> ControlText
FormBorderStyle	Sizable
RightToLeft	No
RightToLeftLayout	False
Text	Form1
UseWaitCursor	False

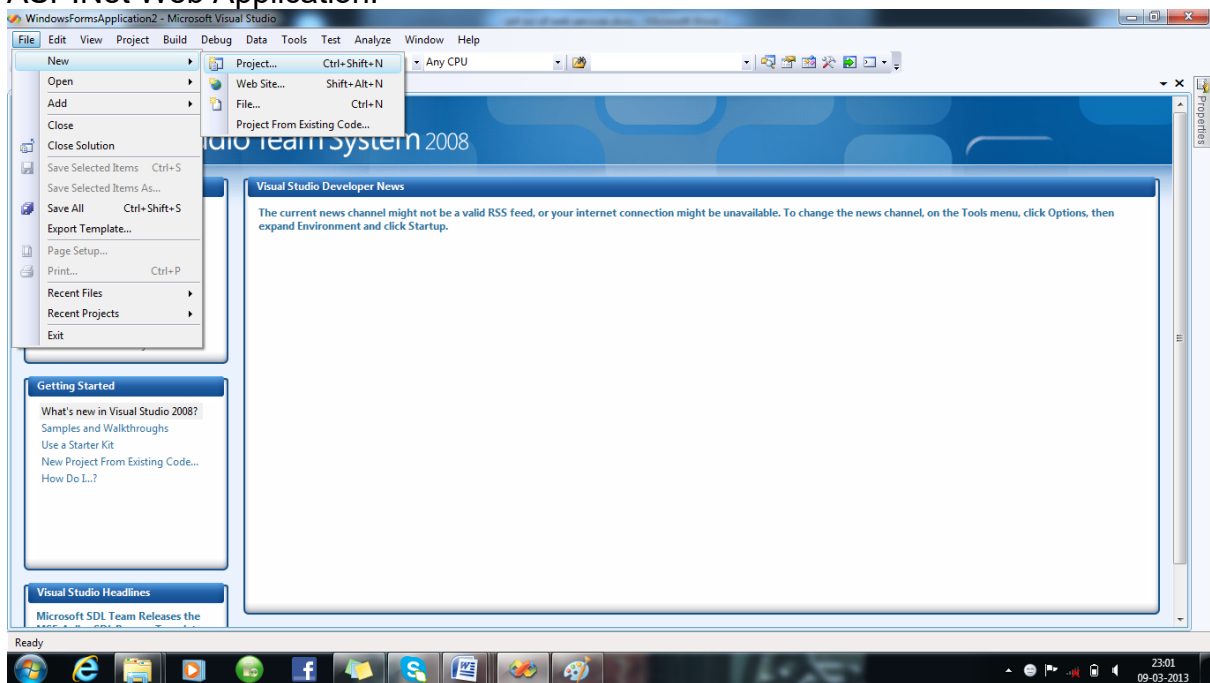
Behavior

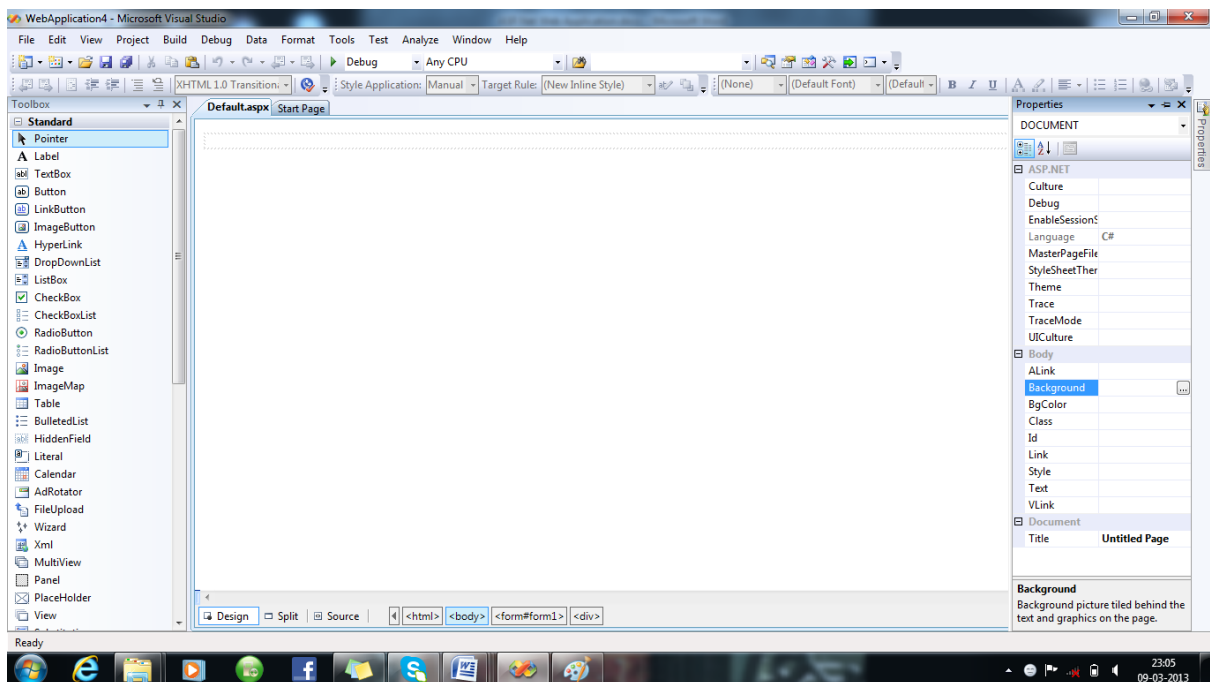
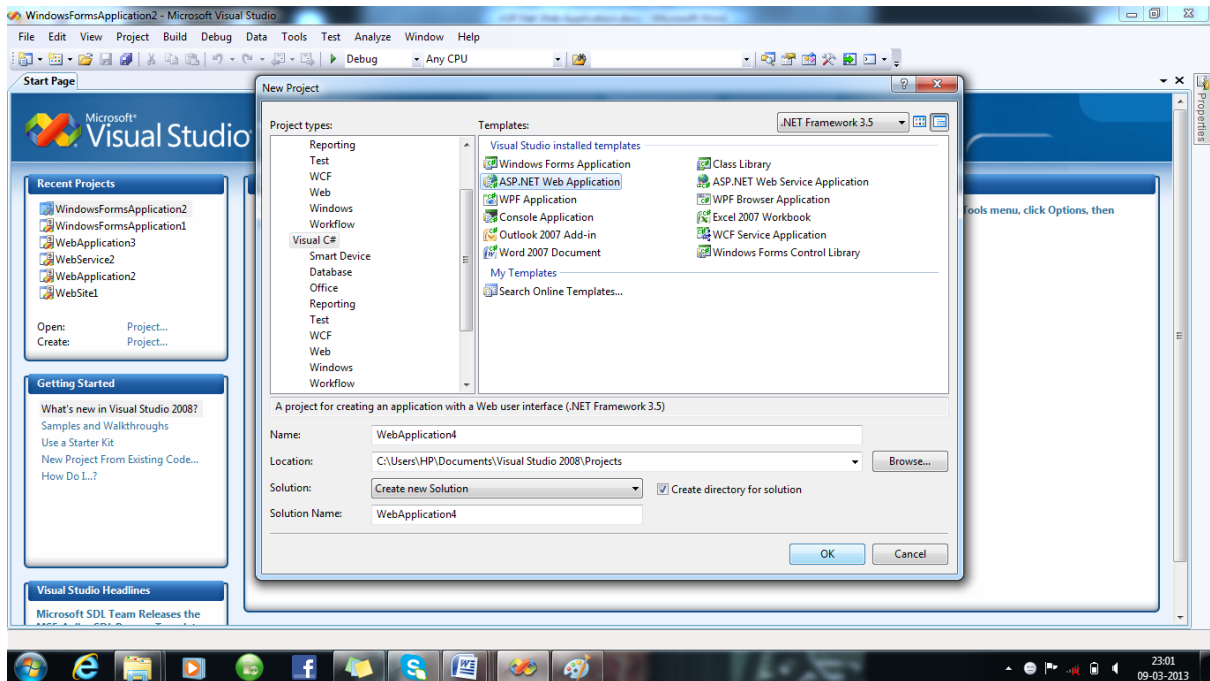
AllowDrop	False
AutoValidate	EnablePreventFo
ContextMenu	(none)
DoubleBuffering	False



Web Based Application in C#

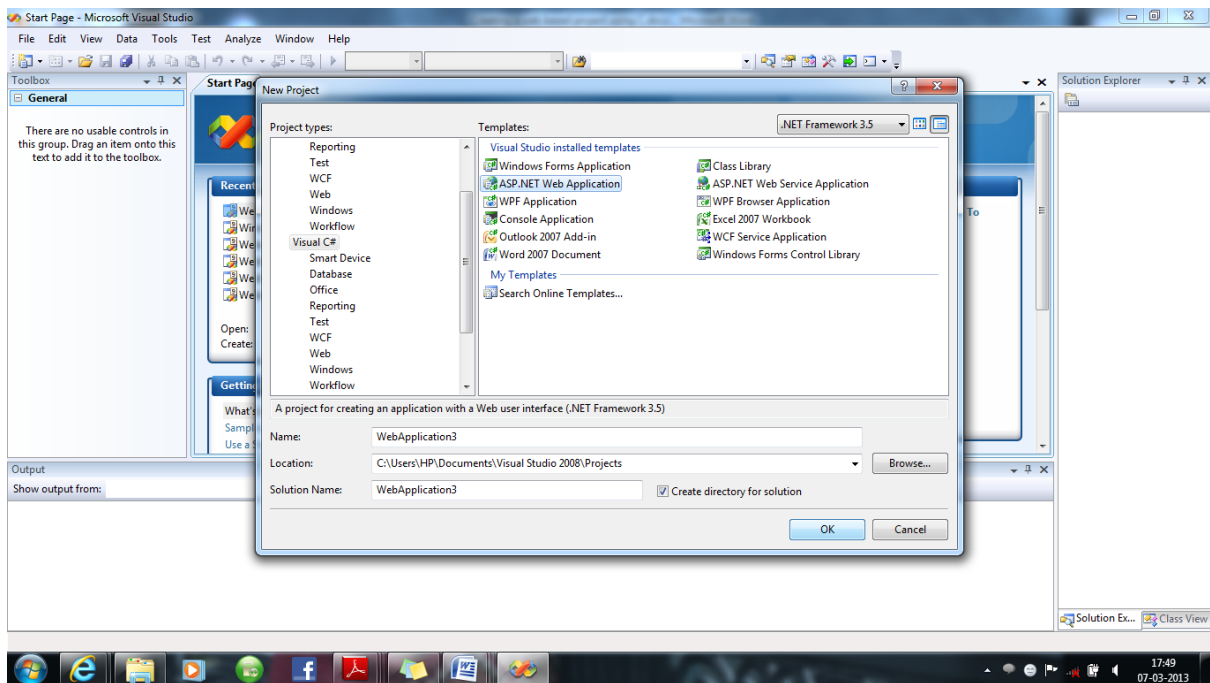
ASP.Net Web Application:



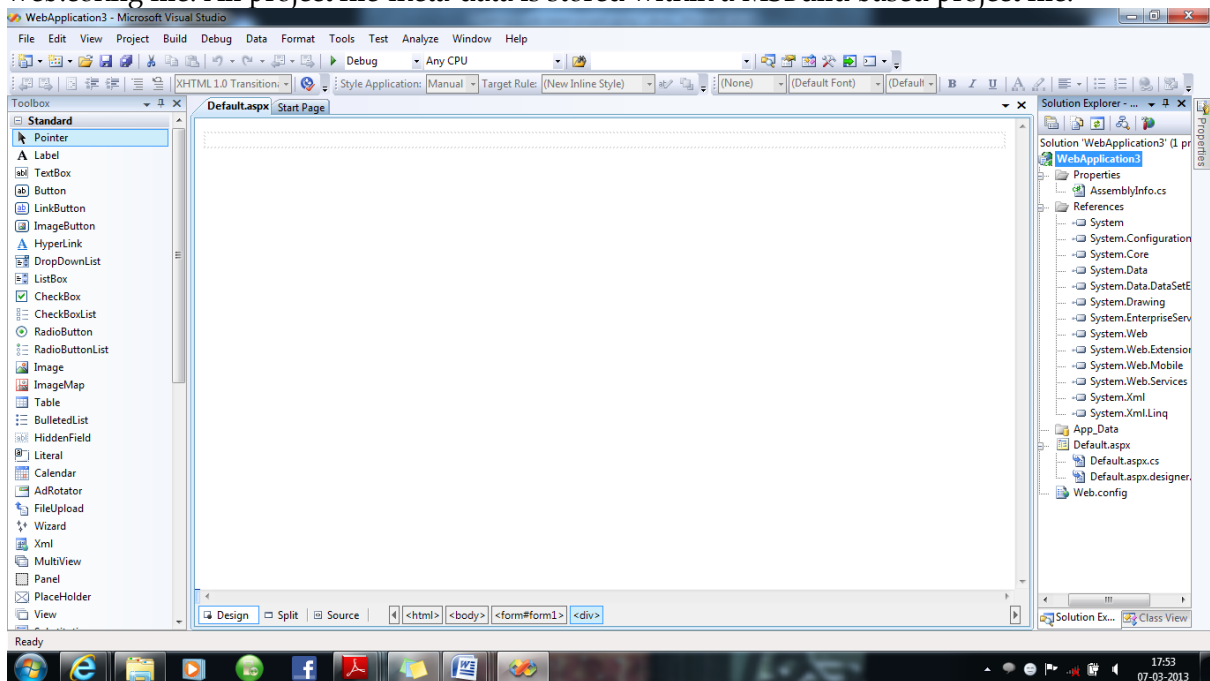


Creating a web based project using C#:

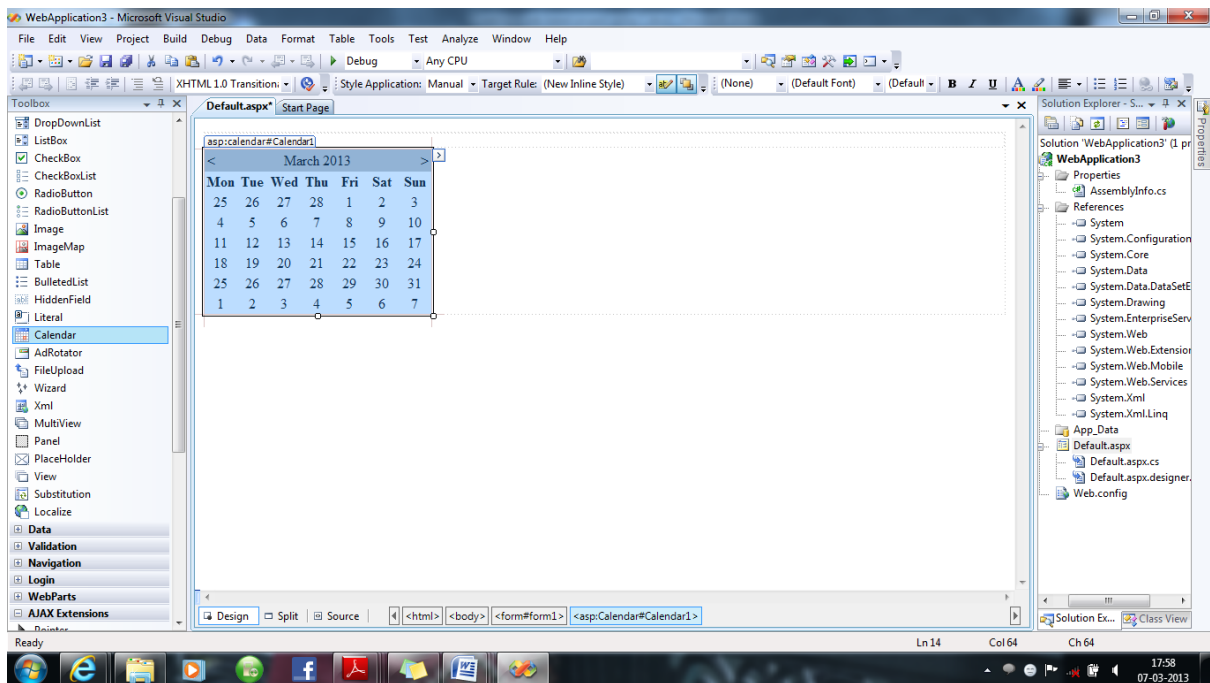
Select File->New->Project within the Visual Studio. Then name your application and click ok. This will bring you the New Project dialog box. Click on the "Visual C#" node in the tree-view on the left hand side of the dialog box and choose the "ASP.NET Web Application" icon:



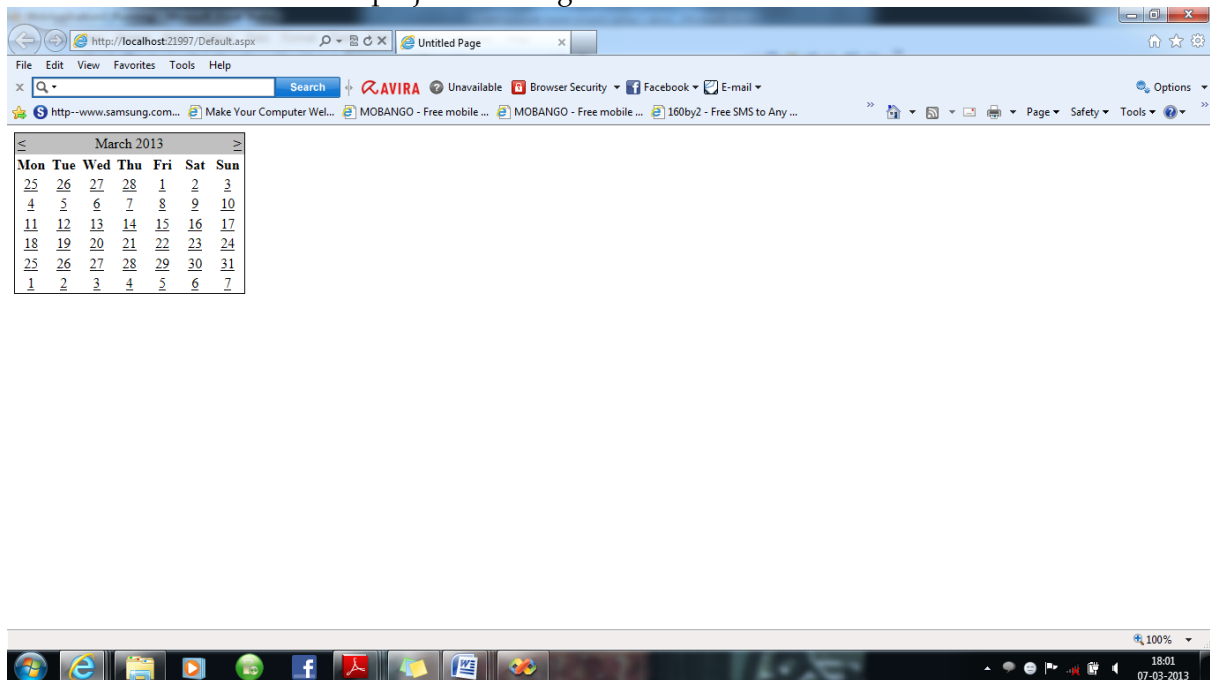
Visual Studio will then create and open a new web project within the solution explorer. By default it will have a single page (Default.aspx), an AssemblyInfo.cs file, as well as a web.config file. All project file-meta-data is stored within a MSBuild based project file.



Double click on the Default.aspx page in the solution explorer to open and edit the page. Add a calendar from ToolBox by dragging and dropping on the design view.



Press F5 to build and run the project in debug mode.

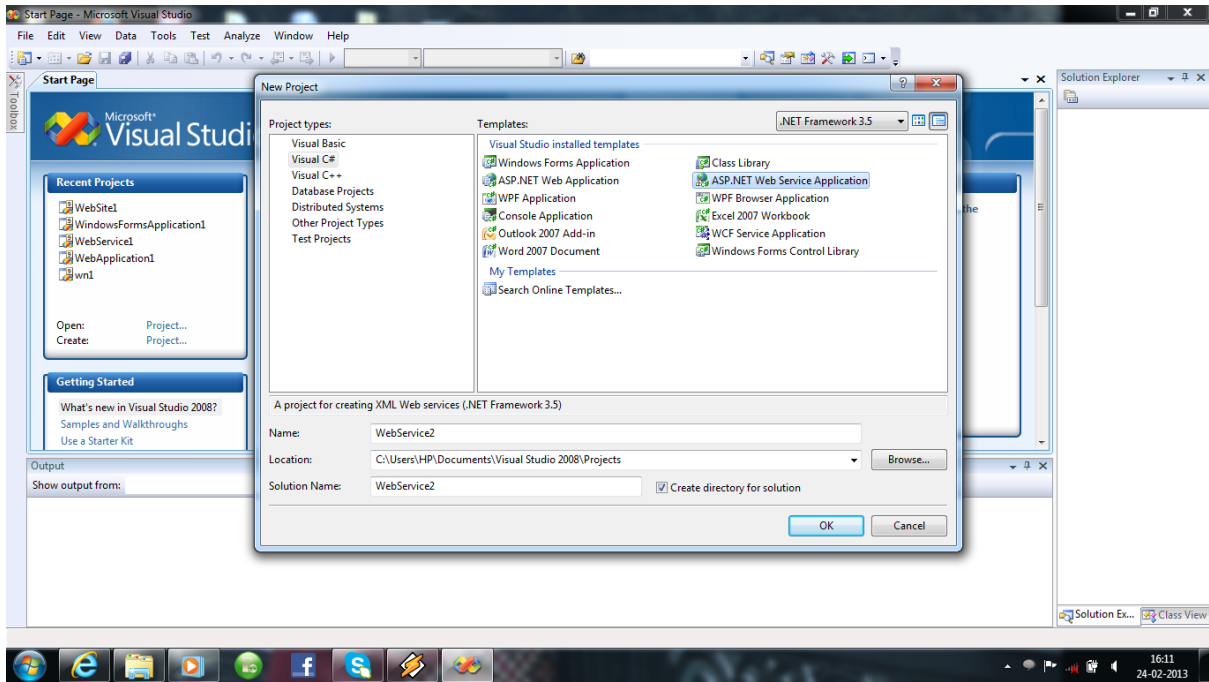


You can end the debug session by closing the browser window, or by choosing the Debug->Stop Debugging (Shift-F5) menu item.

The steps for creating a web service.

1. Open the Microsoft Visual Studio start page.
2. Go to File menu and select New then click Project.
3. Select Visual C# from Project Types and select ASP.NET Web Service Application.
4. Enter the name for your service.

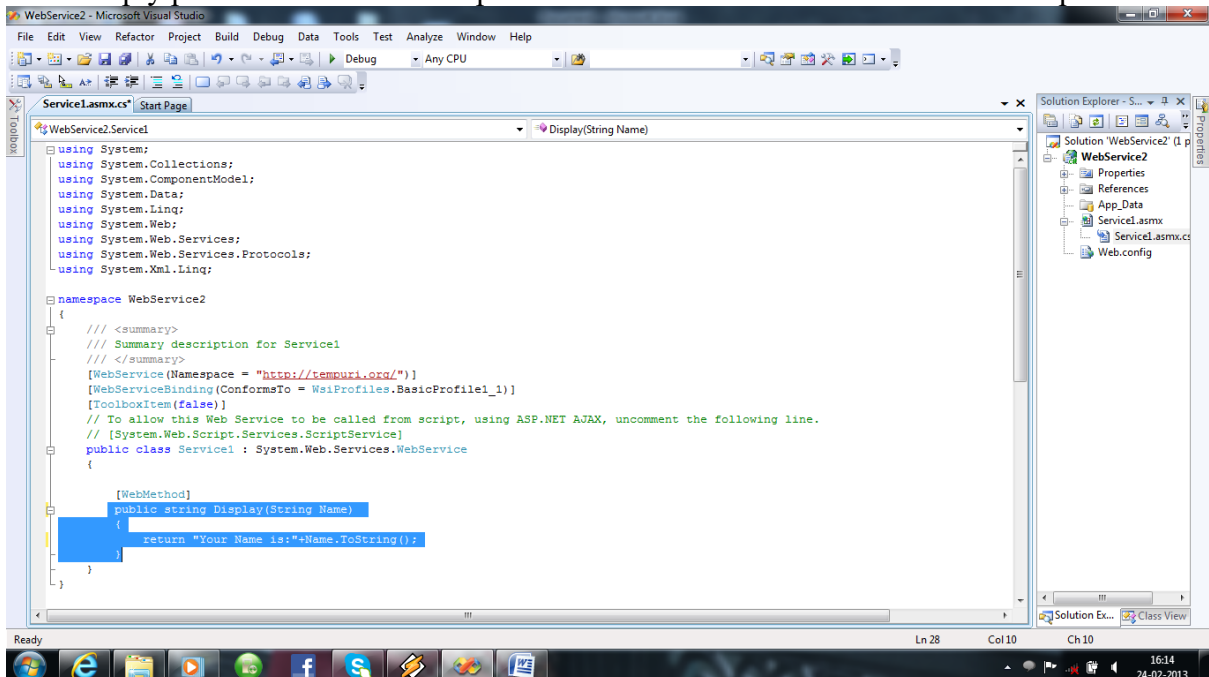
5. Click Ok.



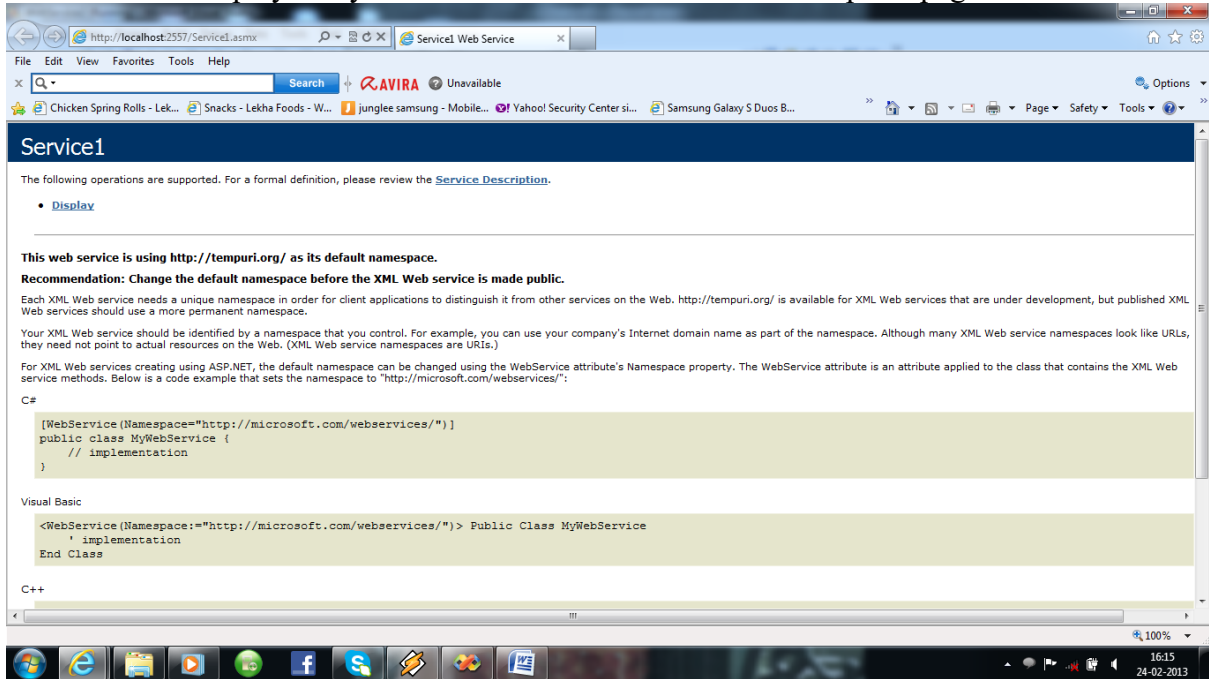
After clicking ok automatically the coding page will be open.

```
[WebMethod]
public string Display(String Name)
{
    return "Your Name is:"+Name.ToString();
}
```

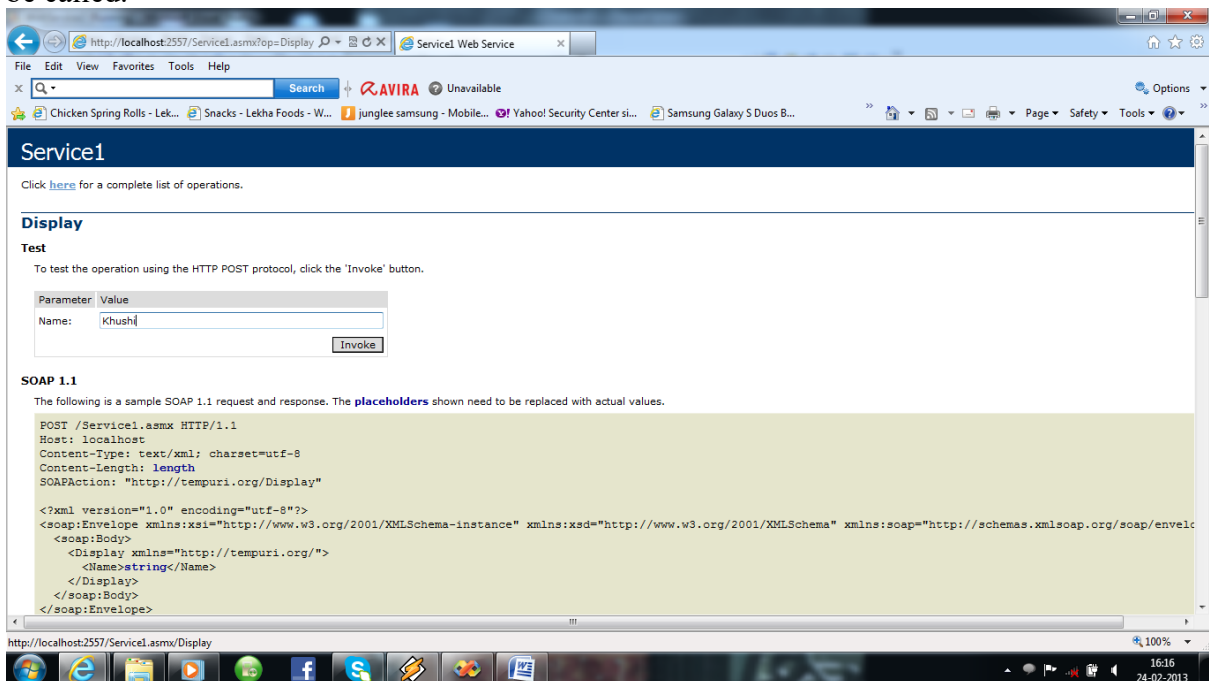
Now simply press F5 to run this sample. It will take few minutes to build and compile.



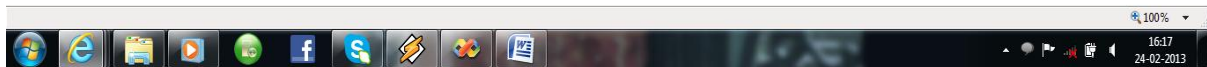
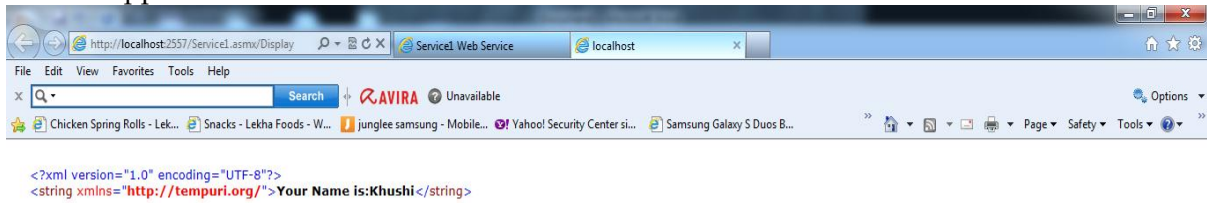
In this example we have created only one web method, Display.
Click the link Display and you will be taken to the services description page.



Now to test the Display web method, simply click the Invoke button and our method will be called.



Recalling our method returns the string “Your Name is:Khushi”; the result is returned in an XML wrapper.



Note that the XML node reflects the datatype of the method’s return value,string. This XML message is received and converted to the string “You Name is:Khushi”.This means that any variable (of type string) in our code can be assigned to the result of our Web method.

18

Files and Database programming

18.1 INTRODUCTION

The File and the Directory classes, which we have used in the previous couple of chapters, are great for direct file and directory manipulation. However, sometimes we wish to get information on them instead, and once again, the System.IO namespace comes to our rescue: The FileInfo and DirectoryInfo classes.

18.1 .1 DATA:

A data file is a computer file which stores data to use by a computer application or system. It generally does not refer to files that contain instructions or code to be executed (typically called program files), or to files which define the operation or structure of an application or system (which include configuration files, directory files, etc.); but specifically to information used as input, or written as output by some other software program.

18.1 .2 DATABASE:

A database is essentially an electronic means of storing data in an organized manner that is a collection of coherent and meaning full data. Data can be anything that a business or individual needs to keep track of and that, prior to computers, could have only been tracked on one or more paper documents. Once stored, data in the database can be retrieved, processed, and displayed by programs as information to the reader. The actual structure that a database uses to store data can take one of many different forms, each which offers certain advantages when that information is to be retrieved or updated. In the next section, we will look at how storing the database in a flat file structure differs from a relational database structure, and the advantages and disadvantages that each of those presents.

we can access the data using access a database using Structured Query Language (SQL), which is a standard language supported by most database software including SQL Server, Access, and Oracle.

18.2: FILES AND DIRECTORIES

18.2 .1 FILES:

Files are the most basic form of database – all of the information is stored in a single file. A flat file includes a field for every item of information that you need to store. While they are easy to create and can be useful in certain situations, flat files are not very efficient. They can be quite wasteful of storagespace, containing a lot of duplicated information, especially in a complex system where multiple files hold connected information. This can make information harder to maintain and retrieve. If you have worked with spreadsheets before, then you have already worked with one of the most common examples of a flat file database.

18.2 .2 DIRECTORIES:

A directory is just a file that contains other files (or directories).
A directory or folder is nothing more than a location on a disk used for storing information about files.

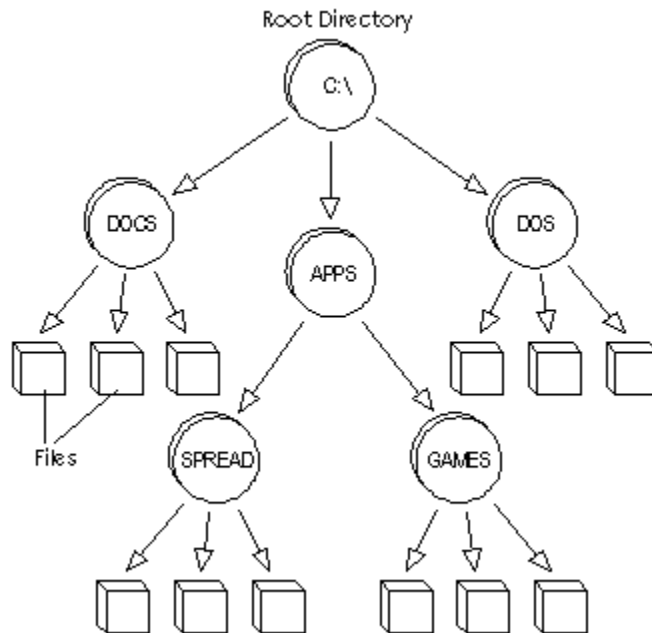
The part of the hard disk where you are authorized to save data is called your home directory. Normally all the data you want will be saved in files and directories in your home directory.

18.2 .3 ORGANIZING A FOLDER AND FILES IN TO A HIERARCHICAL STRUCTURE:

An organizational unit, or container, used to organize folders and files into a hierarchical structure. Directories contain bookkeeping information about files that are, figuratively speaking, beneath them in the hierarchy. You can think of a

directory as a file cabinet that contains folders that contain files. Many graphical user interfaces use the term folder instead of directory.

Computer manuals often describe directories and file structures in terms of an inverted tree. The files and directories at any level are contained in the directory above them. To access a file, you may need to specify the names of all the directories above it. You do this by specifying a path.



The topmost directory in any file is called the root directory. A directory that is below another directory is called a subdirectory. A directory above a subdirectory is called the parent directory. Under DOS and Windows, the root directory is a back slash (/).

18.3 CREATING A FILE :

The File class in the .NET Framework class library provides static methods for creating, reading, copying, moving, and deleting files. In this article, we will see how to create a text file using different options available in .NET.

18.3.1 WE CAN CREATE A FILE IN FOUR DIFFERENT FOLLOWING METHODS :

- File.Create
- File.CreateText
- FileInfo.Create
- FileInfo.CreateText

You could use:

```
using (File.Create(filename)) ;
```

That looks slightly odd, mind you. You could use braces instead:

```
using (File.Create(filename)) {}
```

Or just call Dispose directly:

```
File.Create(filename).Dispose();
```

Either way, if you're going to use this in more than one place you should probably consider wrapping it in a helper method, e.g.

```
public static void CreateEmptyFile(string filename)
{
    File.Create(filename).Dispose();
}
```

18.3.2 CREATING A TEXT FILE:

EXP:

```
public class CreateFileOrFolder
{
    static void Main()
    {
        // Specify a name for your top-level folder.
        string folderName = @"c:\Top-Level Folder";

        // To create a string that specifies the path to a subfolder under
        your
        // top-level folder, add a name for the subfolder to folderName.
        string pathString = System.IO.Path.Combine(folderName,
        "SubFolder");

        // You can write out the path name directly instead of using the
        Combine
        // method. Combine just makes the process easier.
        string pathString2 = @"c:\Top-Level Folder\SubFolder2";

        // You can extend the depth of your path if you want to.
        //pathString = System.IO.Path.Combine(pathString, "SubSubFolder");

        // Create the subfolder. You can verify in File Explorer that you
        have this
```

```

// structure in the C: drive.
//   Local Disk (C:)
//       Top-Level Folder
//           SubFolder
System.IO.Directory.CreateDirectory(pathString);

// Create a file name for the file you want to create.
string fileName = System.IO.Path.GetRandomFileName();

// This example uses a random string for the name, but you also
can specify
// a particular name.
//string fileName = "MyNewFile.txt";

// Use Combine again to add the file name to the path.
pathString = System.IO.Path.Combine(pathString, fileName);

// Verify the path that you have constructed.
Console.WriteLine("Path to my file: {0}\n", pathString);

// Check that the file doesn't already exist. If it doesn't exist,
create
// the file and write integers 0 - 99 to it.
// DANGER: System.IO.File.Create will overwrite the file if it
already exists.
// This could happen even with random file names, although it is
unlikely.
if (!System.IO.File.Exists(pathString))
{
    using (System.IO.FileStream fs =
System.IO.File.Create(pathString))
    {
        for (byte i = 0; i < 100; i++)
        {
            fs.WriteByte(i);
        }
    }
}
else
{
    Console.WriteLine("File \"{0}\" already exists.", fileName);
    return;
}

// Read and display the data from your file.
try
{
    byte[] readBuffer = System.IO.File.ReadAllBytes(pathString);

```

```

        foreach (byte b in readBuffer)
        {
            Console.Write(b + " ");
        }
        Console.WriteLine();
    }
    catch (System.IO.IOException e)
    {
        Console.WriteLine(e.Message);
    }

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
// Sample output:

// Path to my file: c:\Top-Level Folder\SubFolder\ttxvauxe.vv0

//0 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 41 42 43 44 45 46 47 48 49 50 51 52
53 54 55 56
// 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78
79 80 81 82 8
//3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
}

```

Use [Directory.Create](#) to create the directories:

```
Directory.Create("c:\mytempFolder\driveC\New Folder");
```

```

string targetPath = @"C:\mytempFolder\";
string path = @"D:\New Folder\a.txt";
char driveLetter = path[0];
string filePath = path.SubString(3);
string newFilePath = Path.Combine(targetPath,
Path.Combine(String.Format("drive{0}", driveLetter.ToString()),
filePath));

```

18.4 Deleting a File:

```
using System.IO;
```

```
class Program
```

```
{
```



```

static void Main()
{
    // 1.
    // Call Delete wrapper method.
    TryToDelete("Word.doc");
}

/// <summary>
/// Wrap the Delete method with an exception handler.
/// </summary>
static bool TryToDelete(string f)
{
    try
    {
        // A.
        // Try to delete the file.
        File.Delete(f);
        return true;
    }
    catch (IOException)
    {
        // B.
        // We could not delete the file.
        return false;
    }
}
}

```

Output

The file is deleted, or nothing happens.

18.5 DATABASE:

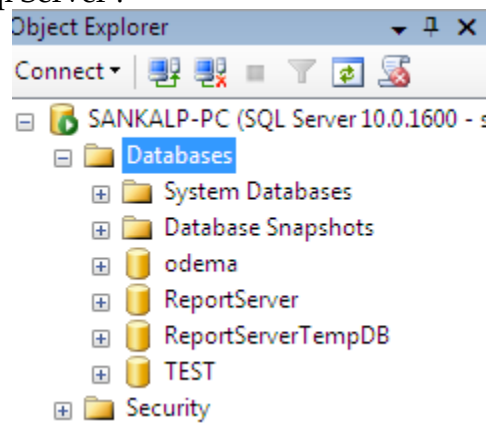
A **Database** is an organized collection of information that is divided into tables. Each table is further divided into rows and columns; these columns store the actual information. You access a database using Structured Query Language (SQL), which is a standard language supported by most database software including SQL Server, Access, and Oracle.

In C# .Net we may use SQL SERVER database for ADO.NET Database access,
Exp: How to create SQL Server database tables,

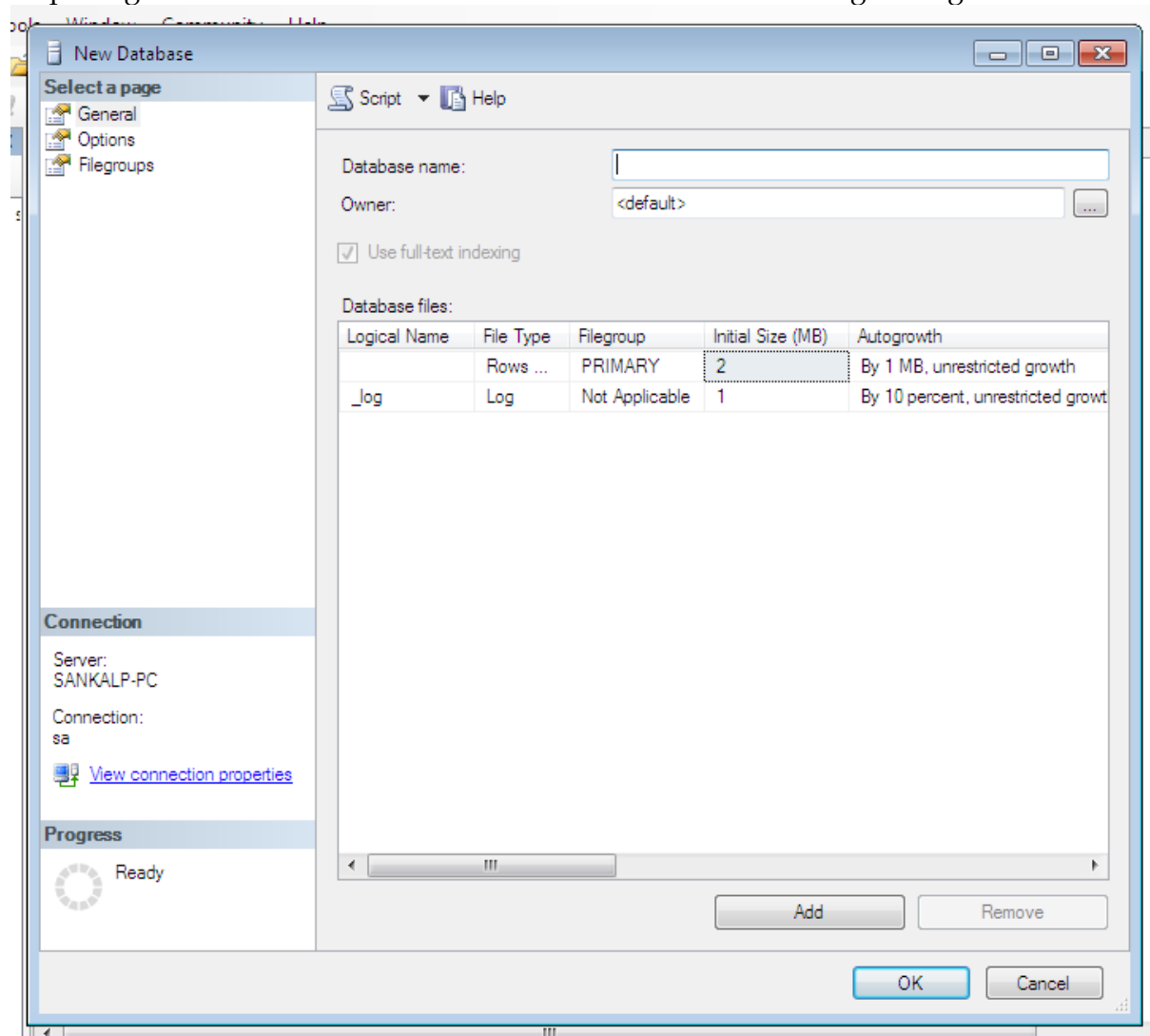
Step1: In Sql Server Query Window ;

```
CREATE DATABASE TEST
```

Select database in Sql Server :

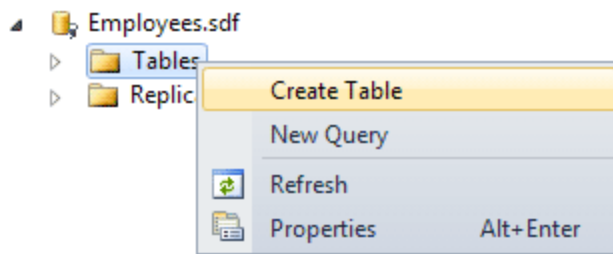


Step2: Right Click to database and create a new database like the given fig:



Step3:

```
CREATE TABLE [dbo].[Employee] (
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [EmpCode] [varchar](50) NULL,
    [EmpName] [varchar](50) NULL,
    [FathersName] [varchar](50) NULL,
    [Address] [varchar](50) NULL,
    [City] [varchar](50) NULL,
    [State] [varchar](50) NULL,
)
```



New Table

View: **General** Refresh Help

Name:

Column Name	Data Type	Length	Allow Nulls	Unique	Primary Key

Delete

Default Value

Identity	False
Identity Increment	
Identity Seed	
Is RowGuid	False
Precision	
Scale	

Default Value
Default value for this column.

Connection: Microsoft SQL Server Compact
Employees.sdf
[View connection properties](#)

OK Cancel

18.5.1 COMPONENTS OF DATABASE:

An Access database consists of several different components. Each component listed is called an *object*.

Listed below are the names and descriptions of the different objects you can use in Access. This tutorial will focus on the first two objects: tables and queries.

Tables: tables are where the actual data is defined and entered. Tables consist of records (rows) and fields (columns).

Queries: queries are basically questions about the data in a database. A query consists of specifications indicating which fields, records, and summaries you want to see from a database. Queries allow you to extract data based on the criteria you define.

Forms: forms are designed to ease the data entry process. For example, you can create a data entry form that looks exactly like a paper form. People generally prefer to enter data into a well-designed form, rather than a table.

Reports: when you want to print records from your database, design a report. Access even has a wizard to help produce mailing labels.

Pages: a data access page is a special type of Web page designed for viewing and working with data from the Internet or an intranet. This data is stored in a Microsoft Access database or a Microsoft SQL Server database.

Macros: a macro is a set of one or more actions that each performs a particular operation, such as opening a form or printing a report. Macros can help you automate common tasks. For example, you can run a macro that prints a report when a user clicks a command button.

Modules: a module is a collection of Visual Basic for Applications declarations and procedures that are stored together as a unit.

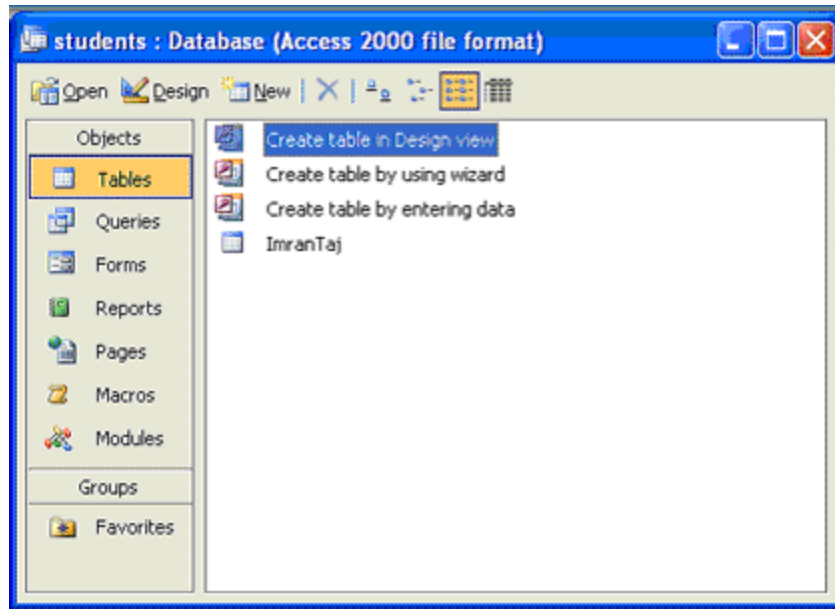
18.5.2 ADO.NET COMPONENTS:

- SqlConnection
- SqlDataAdapter
- DataSet
- DataTable
- DataRow, DataColumn collections
- SqlDataReader
- SqlCommand

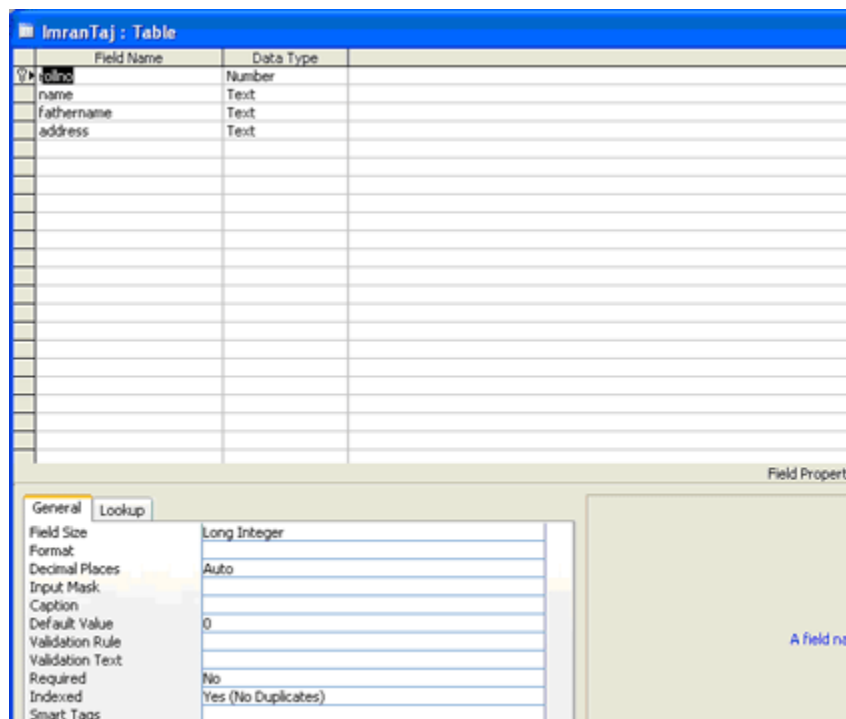
18.6 MS –ACCESS:

We can connect C# Database with Microsoft Access database also;
These are very simple steps to create and connect an Access database in C#.

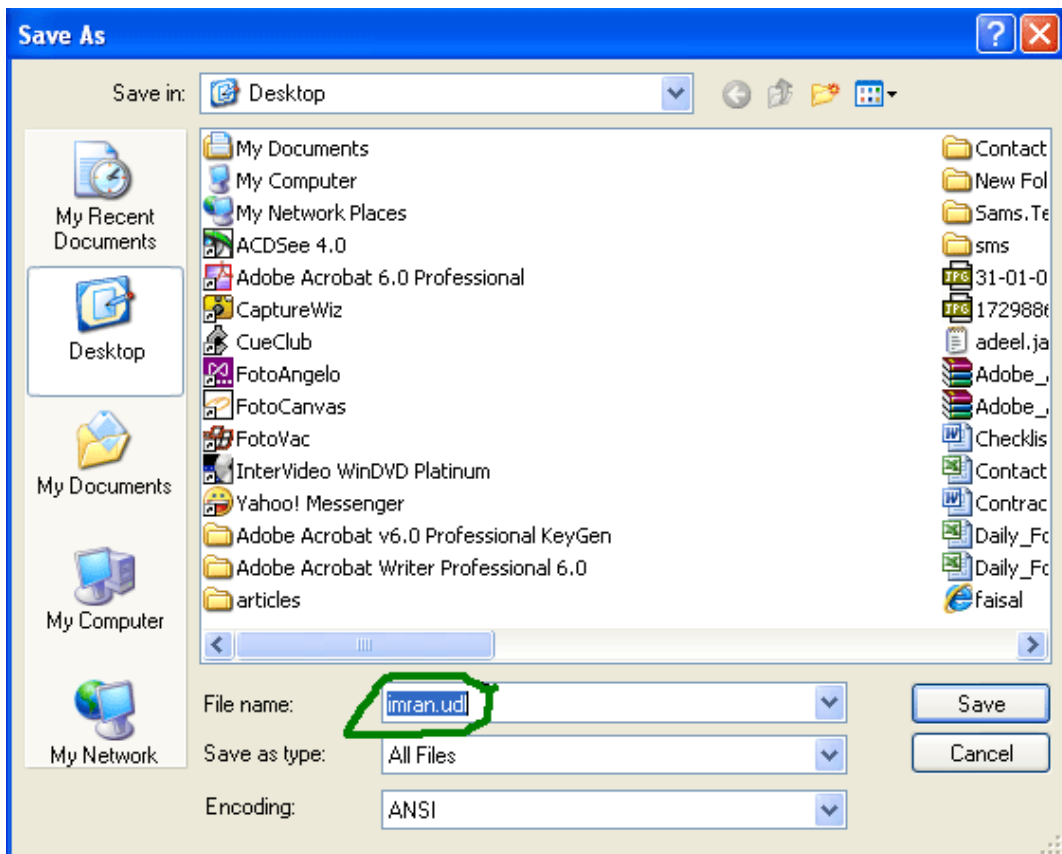
Step1: Create Access Database (eg: student)



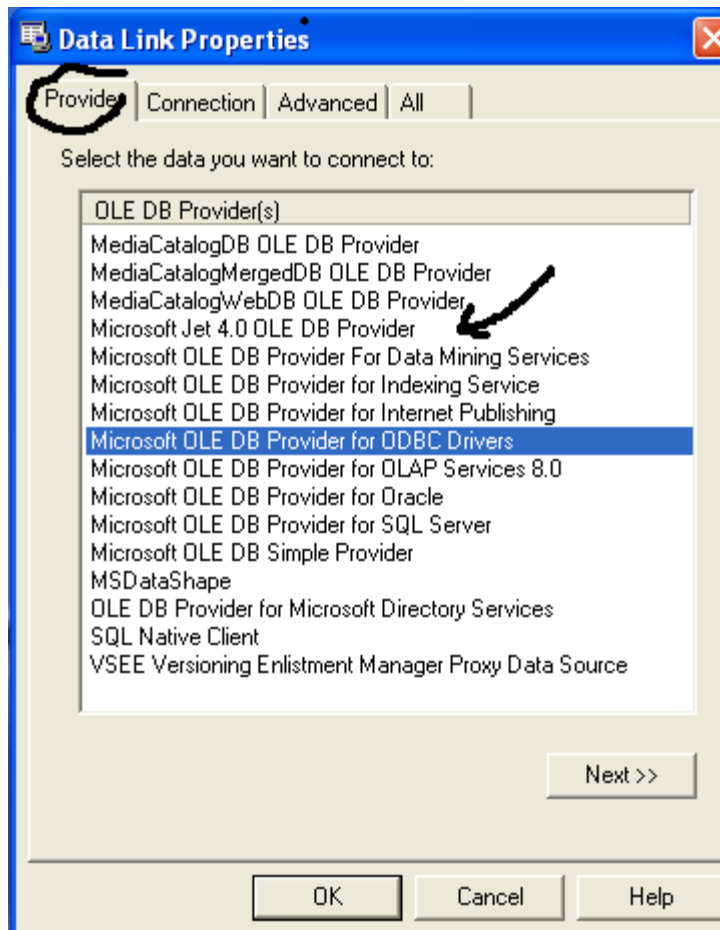
In design View we can create the column and column fields described below:



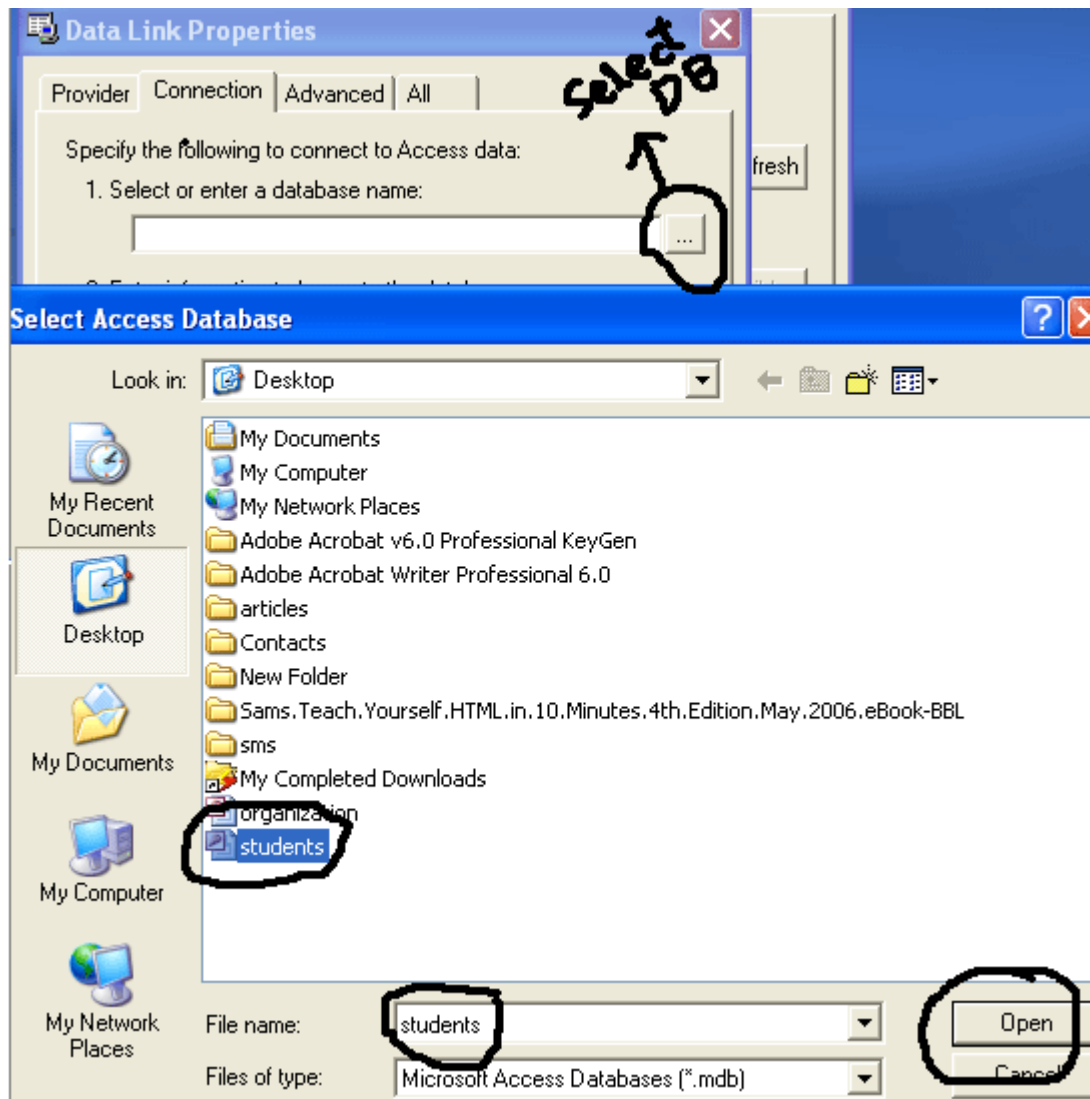
Now open you notepad and click on save As button. Name then Imran.hdl. Change save type "ALL FILES".



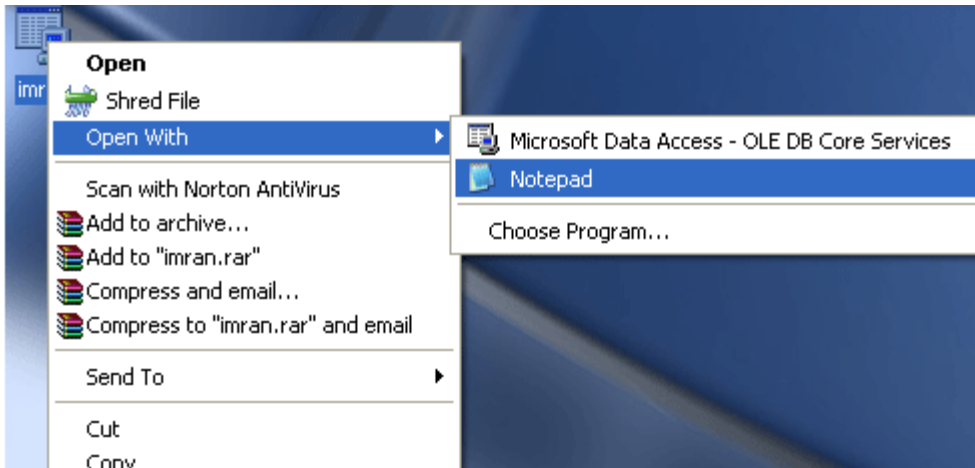
Now double click on imran.udl file. A wizard will start like this:



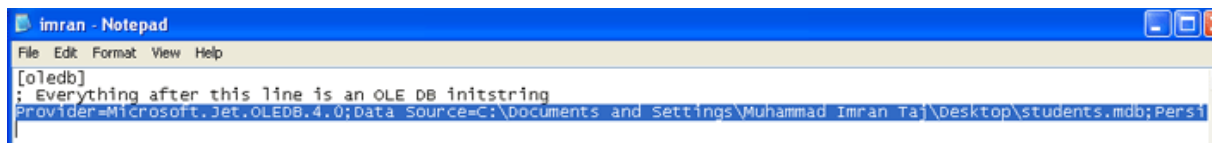
Click Provider TAB, select Microsoft Jet 4.0 OLE DB (denoted by black arrow) then click next. Now click "Select or enter a database name" and select the desire database then click open.



- Now click on test connection and click OK
- Now edit this UDL file with note pad and copy link as shown below,



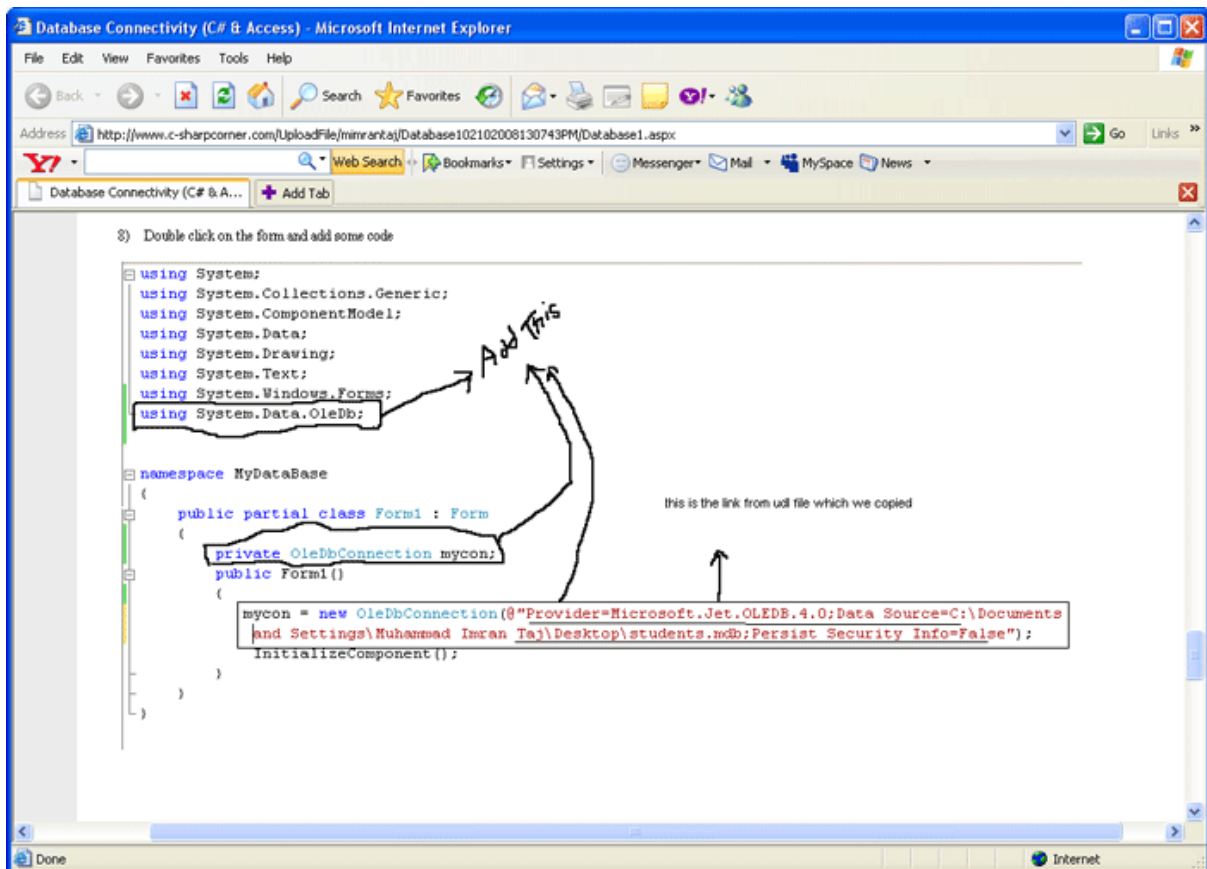
now open notepad:



Now Open Visual Studio editor and create Text boxes and buttons like:

A screenshot of a Visual Studio Windows Form titled 'Form1'. The form has a light beige background. It contains four text boxes arranged vertically, each with a label to its left: 'Roll No', 'Name', 'Father Name', and 'Address'. Below the text boxes is a button labeled 'Insert'.

- Double click on the form and add some code



```

cmd.CommandText = "insert into student values(" + this.textBox1.Text
+ "," + this.textBox2.Text + "," + this.textBox3.Text + "," + this.textBox4.Text
+ ")";
int temp = cmd.ExecuteNonQuery();
if (temp > 0)
{
    MessageBox.Show("Record Added");
}
else
{
    MessageBox.Show("Record not Added");
}
mycon.Close();

```

18.7 Inserting Data in MS-Access :

With the help of C# code we can insert data in Access database as described below;

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

namespace Project2
{
    public partial class Form1 : Form1
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Button1_Click(object sender, EventArgs e)
        {
            //what should i write here so that it store in details.mdb database.
        }

    }
}
```

now write the following code in button_click Event;

Write the following code in Button_Click event :

```
string constr=@"Provider=Microsoft.Jet.OLEDB.4.0; Data Source=d:\details.mdb";
string cmdstr="insert into info(name,age)values(?,?)";

OleDbConnection con=new OleDbConnection(constr);
OleDbCommand com=new OleDbCommand(cmdstr,con);
```

```
con.Open();
com.Parameters.AddWithValue("?",textBox1.Text);
com.Parameters.AddWithValue("?",int.Parse(textBox3.Text));
com.ExecuteNonQuery();
con.Close();
```

And Import the following namespaces :

```
using System.Data.OleDb;
```

on the other hand in configuration section the data would be inserted like below description;

App.config

```
<?xml version="1.0" encoding="utf-8" ?>

<configuration>

<appSettings>

    <add key="dsn" value="Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=..\EMP.mdb" />

</appSettings>

</configuration>
```

And in click event we may write;

```
string ConnectionString = System.Configuration.ConfigurationSettings.AppSettings["dsn"];
```

18.8 Deleting Data in MS-Access :

In click event we may write the following code for deleting a particular data from MS Access;

```
try
{
```

```

        OleDbConnection con = new
OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data Source=
|DataDirectory|/gdsg.mdb");

        OleDbCommand cmd = new OleDbCommand("DELETE FROM Table1
WHERE ID="+TextBox3.Text+"", con);

        con.Open();
        cmd.CommandType = CommandType.Text;
        OleDbDataAdapter da = new OleDbDataAdapter(cmd);

        cmd.ExecuteNonQuery();
        Response.Write("deleted succesfully");
    }
    catch (Exception ex)
    {
        Response.Write(ex.ToString());
    }
}

```

18.9 Updating Data in MS-Access:

```

try
{
    OleDbConnection con = new
OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data Source=
|DataDirectory|/gdsg.mdb");
    OleDbCommand cmd = new OleDbCommand("UPDATE Table1
SET designation='" + TextBox5.Text + "' WHERE ID=" +
TextBox4.Text + "'", con);

    con.Open();
    cmd.CommandType = CommandType.Text;

    OleDbDataAdapter da = new OleDbDataAdapter(cmd);
    cmd.CommandType = CommandType.Text;
    cmd.ExecuteNonQuery();
    Response.Write("updated successfully");
}
catch (Exception ex)
{
    Response.Write(ex.ToString());
}
}

```

18.10 Retrieving Data in MS-Access :

For retrieving a particular data or whole data from the database of Access we may write the select query like statement in access;

```
try
{
    OleDbConnection con = new
OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data Source=
|DataDirectory|/gdsg.mdb");
    OleDbCommand cmd = new OleDbCommand("select * from
Table1", con);
    cmd.CommandType = CommandType.Text;
    OleDbDataAdapter da = new OleDbDataAdapter(cmd);
    DataSet ds = new DataSet();
    da.Fill(ds, "Temptbl");
    GridView1.DataSource = ds;
    GridView1.DataBind();
}
catch (Exception ex)
{
    Response.Write(ex.ToString());
}
```

18.11 SQL (STRUCTURED QUERY LANGUAGE):

SQL is structured Query Language which is a computer language for storing, manipulating and retrieving data stored in relational database.

SQL is the standard language for Relation Database System. All relational database management systems like MySQL, MS Access, Oracle, Sybase, Informix, postgres and SQL Server uses SQL as standard database language.

Also they are using different dialects, Such as:

- MS SQL Server using T-SQL,
- Oracle using PL/SQL,
- MS Access version of SQL is called JET SQL (native format)etc

We need to write database program in c#:

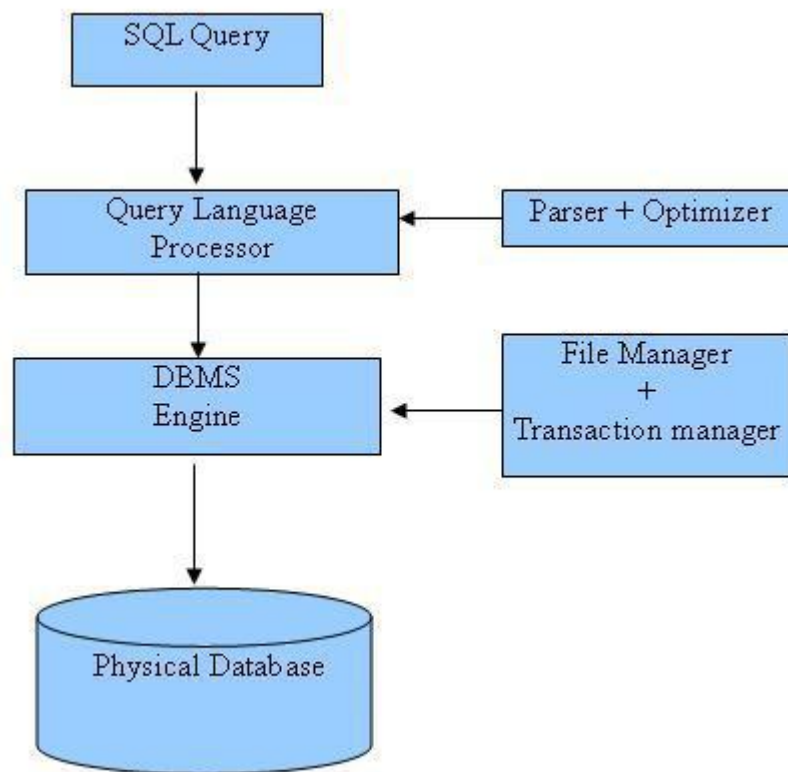
- Queries
- INSERT statements
- UPDATE statements
- DELETE statements
- T-SQL data types

18.11.1 SQL PROCESS:

When you are executing an SQL command for any RDBMS, the system determines the best way to carry out your request and SQL engine figures out how to interpret the task.

There are various components included in the process. These components are Query Dispatcher, Optimization engines, Classic Query Engine and SQL query engine etc. Classic query engine handles all non-SQL queries but SQL query engine won't handle logical files.

Following is a simple digram showing SQL Architecture:



18.11.2 SQL Commands:

The standard SQL commands to interact with relational databases are CREATE, SELECT, INSERT, UPDATE, DELETE, and DROP. These commands can be classified into groups based on their nature:

DDL - Data Definition Language

DML - Data Manipulation Language

DCL - Data Control Language

DQL - Data Query Language

18.12 INSERTING DATA IN SQL:

The INSERT statement is much simpler than a query, particularly because the WHERE and ORDER BY clauses have no meaning when inserting data and therefore aren't used.

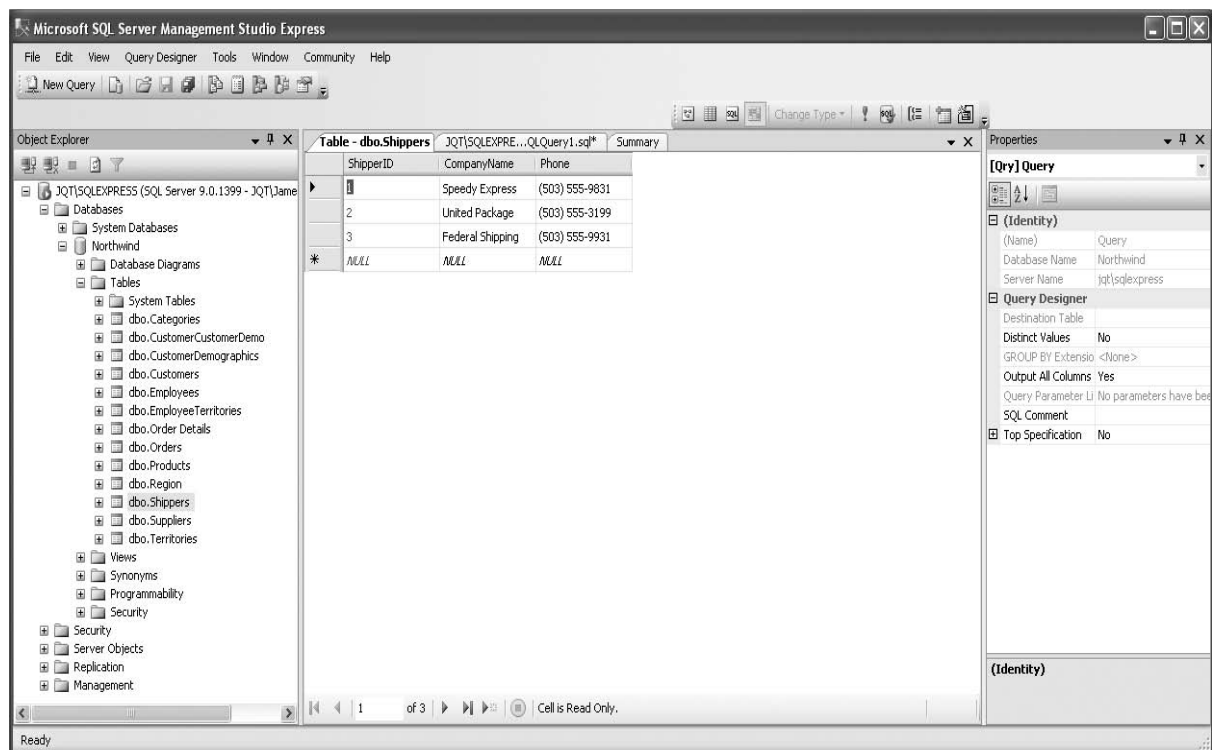
A basic INSERT statement has these parts:

INSERT INTO <table>

(<column1>, <column2>, ..., <columnN>)

VALUES (<value1>, <value2>, ..., <valueN>)

Now in object Explorer, right click on the table and click open the table. The table has three rows, which are displayed in a tabbed window;



Insert statement with stored procedure:

```
CREATE PROC [dbo].[usp_Employee_Insert]
@Id [int] ,
@Name [varchar] (50),
@City [varchar] (50),
@State [varchar] (50),
@Address [varchar] (50),
@ContactNo [varchar] (50),
@CreatedBy [varchar] (50),
@CreatedOn [varchar] (50),
@UpdatedBy [varchar] (50),
@UpdatedOn [varchar] (50)
```

```

AS
INSERT INTO [Vendor]
(
    [Name] ,
        [City] ,
        [State] ,
        [Address] ,
        [ContactNo] ,
        [CreatedBy] ,
        [CreatedOn] ,
        [UpdatedBy] ,
        [UpdatedOn]
)

VALUES
(
    @Name,
    @City ,
    @State ,
    @Address ,
    @ContactNo ,
    @CreatedBy ,
    @CreatedOn ,
    @UpdatedBy ,
    @UpdatedOn
)

```

This is

This is the Example of Insertion of data in table with Stored Procedure.

Deleting Data From SQL :

18.13 SQL DELETE STATEMENT:

```

DELETE FROM table_name
WHERE {CONDITION};

```

18.13.1 DELETION WITHOUT CONDITION:

```

DELETE FROM table_name ;

```

18.13.2 Deletion with stored Procedure:

```

CREATE PROC [dbo].[usp_Employee_Delete]
@Id int
AS DELETE FROM [Employee]

```

```
WHERE Id=@Id
```

18.14 Updating Data in SQL :

```
UPDATE table_name  
SET column1 = value1, column2 =  
value2....columnN=valueN  
[ WHERE CONDITION ];
```

18.14.1 Updating with stored procedure:

```
CREATE PROC [dbo].[Usp_Employee_Update]  
@Id [int] ,  
@Name [varchar] (50),  
@City [varchar] (50),  
@State [varchar] (50),  
@Address [varchar] (50),  
@ContactNo [varchar] (50),  
@CreatedBy [varchar] (50),  
@CreatedOn [varchar] (50),  
@UpdatedBy [varchar] (50),  
@UpdatedOn [varchar] (50)  
AS UPDATE [Employee]  
SET  
[Name]=@Name,  
[City] =@City ,  
[State] =@State ,  
[Address]=@Address ,  
[ContactNo]= @ContactNo ,  
[CreatedBy] =@CreatedBy ,  
[CreatedOn] =@CreatedOn ,  
[UpdatedBy] =@UpdatedBy ,  
[UpdatedOn] =@UpdatedOn  
WHERE Id=@Id
```

18.15 Retrieving Data in SQL:(Select Statement):-

A select statement is used to retrieve information from one or more tables. When data is retrieved from more than one table a join is performed, this is covered in the Advanced SQL section.

Exp: SELECT * FROM Employee;

Exp: **SELECT [DISTINCT] {columnlist,*} FROM tablename**
[WHERE condition [AND condition]]
[ORDER BY columnlist];

Exp:
SELECT column1, column2....columnN
FROM table_name;

18.16 XML(Extensible Markup Language):-

Stores XML data. You can store xml instances in a column or a variable (SQL Server 2005 only).

XML stands for eXtensible Markup Language.

XML is designed to transport and store data.

XML is important to know, and very easy to learn.

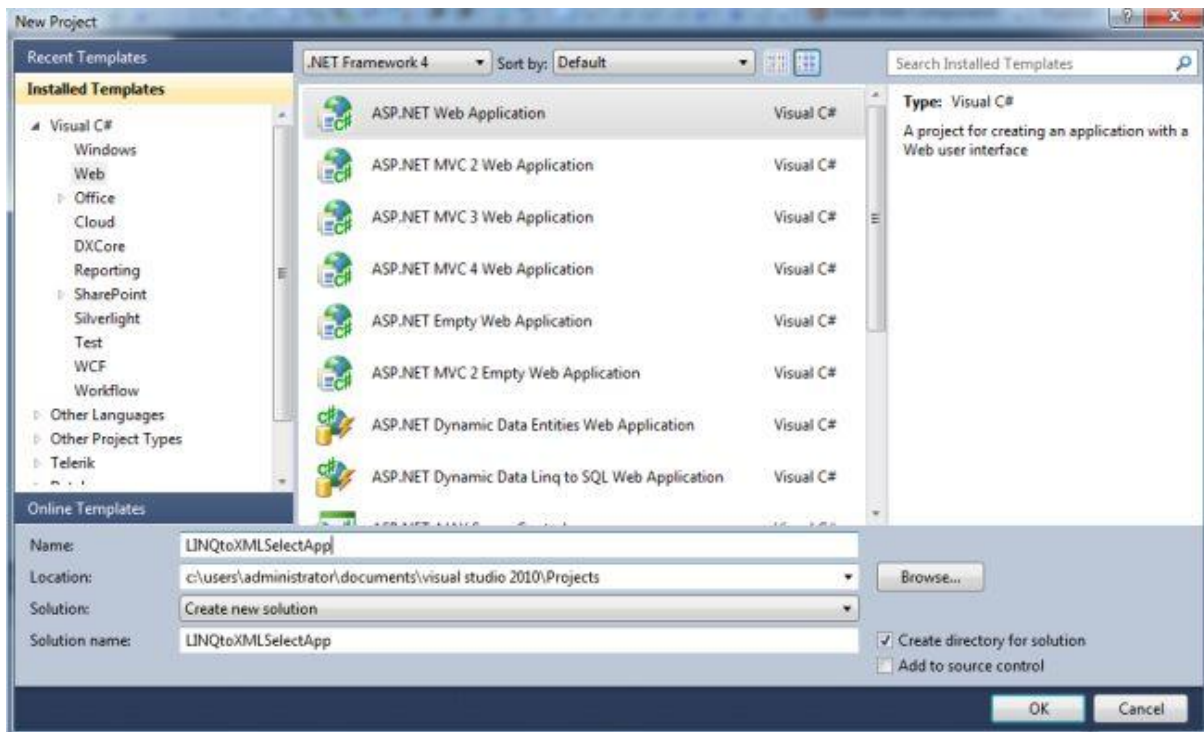
Exp:

```
<?xml version="1.0"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Insert data to XML using LinQ, to XML;

In simple terms "It provides flexibility to insert data into XML with the help of a LINQ query." .

Step 1: Create a new "ASP.NET Web Application", as in:



Step 2: The complete code of Employee.xml looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<Employees>
  <Employee>
    <Id>1</Id>
    <FirstName>Vijay</FirstName>
    <LastName>Prativadi</LastName>
    <Age>26</Age>
  </Employee>
  <Employee>
    <Id>2</Id>
    <FirstName>Sandeep</FirstName>
    <LastName>Reddy</LastName>
    <Age>28</Age>
  </Employee>
</Employees>
```

Step 3: The complete code of webform1.aspx looks like this:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="WebForm1.aspx.cs" Inherits="
LINQtoXMLInsertApp.WebForm1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
  <title></title>
```

```
</head>
<body>
  <form id="form1" runat="server">
    <center>
      <div>
        <table>
          <tr>
            <td colspan="2" align="center">
              <asp:Label ID="Label1" runat="server" Text="Insert Data using LINQ-to-
XML" Font-Bold="true"
              Font-Size="Large" Font-Families="Verdana" ForeColor="Maroon"></asp:Label>
            </td>
          </tr>
          <tr>
            <td>
              <asp:Label ID="Label6" runat="server" Text="Please Enter Id" Font-
Size="Large" Font-Families="Verdana"
              Font-Italic="true"></asp:Label>
            </td>
            <td>
              <asp:TextBox ID="TextBox4" runat="server"></asp:TextBox>
            </td>
          </tr>
          <tr>
            <td>
              <asp:Label ID="Label2" runat="server" Text="Please Enter FirstName" Font-
Size="Large"
              Font-Families="Verdana" Font-Italic="true"></asp:Label>
            </td>
            <td>
              <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
            </td>
          </tr>
          <tr>
            <td>
              <asp:Label ID="Label3" runat="server" Text="Please Enter LastName" Font-
Size="Large"
              Font-Families="Verdana" Font-Italic="true"></asp:Label>
            </td>
            <td>
              <asp:TextBox ID="TextBox2" runat="server"></asp:TextBox>
            </td>
          </tr>
          <tr>
            <td>
              <asp:Label ID="Label4" runat="server" Text="Please Enter Age" Font-
Size="Large" Font-Families="Verdana"
              Font-Italic="true"></asp:Label>
            </td>
            <td>
              <asp:TextBox ID="TextBox3" runat="server"></asp:TextBox>
            </td>
          </tr>
        </table>
      </div>
    </center>
  </form>
</body>
```

```

        </td>
        <td>
            <asp:TextBox ID="TextBox3" runat="server"> </asp:TextBox>
        </td>
    </tr>
    <tr>
        <td colspan="2" align="center">
            <asp:Button ID="Button1" runat="server" Text="Insert Data" Font-
Names="Verdana"Width="213px"
            BackColor="Orange" Font-Bold="True" OnClick="Button1_Click" />
        </td>
    </tr>
    <tr>
        <td colspan="2" align="center">
            <asp:Label ID="Label5" runat="server" Font-Bold="true" Font-
Names="Verdana"> </asp:Label>
        </td>
    </tr>
</table>
</div>
</center>
</form>
</body>
</html>

```

Step 4: The complete code of webform1.aspx.cs looks like this:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Xml.Linq;
namespace LINQtoXMLInsertApp
{
    public partial class WebForm1 : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            TextBox4.Focus();
        }
        protected void Button1_Click(object sender, EventArgs e)
        {
            if (string.IsNullOrEmpty(TextBox4.Text) || string.IsNullOrEmpty(TextBox1.Text)
||string.IsNullOrEmpty(TextBox2.Text) || string.IsNullOrEmpty(TextBox3.Text))
            {
                Label5.Text = "Please Enter Some Values";
            }
        }
    }
}

```

```

        Label5.ForeColor = System.Drawing.Color.Red;
    }
    else
    {
        XDocument document = XDocument.Load(Server.MapPath("Employee.xml"));
        document.Element("Employees").Add(new XElement("Employee", new XElement("Id",
        TextBox4.Text), new XElement("FirstName", TextBox1.Text), new XElement("LastName",
        TextBox2.Text), new XElement("Age", TextBox3.Text)));
        document.Save(Server.MapPath("Employee.xml"));
        Label5.Text = "Data Inserted Successfully";
        Label5.ForeColor = System.Drawing.Color.Green;
        TextBox4.Text = string.Empty;
        TextBox1.Text = string.Empty;
        TextBox2.Text = string.Empty;
        TextBox3.Text = string.Empty;
    }
}
}
}

```

Step 5: The output of the application looks like this:

Insert Data using LINQ-to-XML

Please Enter Id

Please Enter FirstName

Please Enter LastName

Please Enter Age

Insert Data

Step 6: The data inserting to XML output of the application looks like this:

Insert Data using LINQ-to-XML

Please Enter Id

Please Enter FirstName

Please Enter LastName

Please Enter Age

Insert Data

Data Inserted Successfully

Step 7: The data inserted to XML looks like this:

```

<?xml version="1.0" encoding="utf-8"?>
<Employees>

```



```

<Employee>
  <Id>1</Id>
  <FirstName>Vijay</FirstName>
  <LastName>Prativadi</LastName>
  <Age>26</Age>
</Employee>
<Employee>
  <Id>2</Id>
  <FirstName>Sandeep</FirstName>
  <LastName>Reddy</LastName>
  <Age>28</Age>
</Employee>
<Employee>
  <Id>3</Id>
  <FirstName>Ajit</FirstName>
  <LastName>Kumar</LastName>
  <Age>26</Age>
</Employee>
</Employees>

```

Deleting Data in XML:

```

<?xml version="1.0" encoding="utf-8"?>
<adResponses>
<ad adname="WQC" hitCount="10" />
<ad adname="Google" hitCount="6" />
<ad adname="Facebook" hitCount="4" />
<ad adname="Twitter" hitCount="6" />
</adResponses>

```

now:

```

string strXML = "<?xml version=\"1.0\" encoding=\"utf-8\"?>
<adResponses><ad adname=\"WQC\" hitCount=\"10\" />
<ad adname=\"Google\" hitCount=\"6\" />
<ad adname=\"Facebook\" hitCount=\"4\" />
<ad adname=\"Twitter\" hitCount=\"6\" />
</adResponses>";

```

```

XmlDocument xDoc = new XmlDocument();
xDoc.LoadXml(strXML);
XmlNodeList xNodeList =
xDoc.SelectNodes(@"//adResponses/ad[@hitCount=10]");
foreach (XmlNode xNode in xNodeList)
{
    xDoc.ChildNodes[1].RemoveChild(xNode);
}
strXML = xDoc.InnerXml.ToString();

```

Updating XML data

To update data in an XML column, use the SQL UPDATE statement. Include a WHERE clause when you want to update specific rows. The entire column value will be replaced. The input to the XML column must be a well-formed XML document. The application data type can be an XML, character, or binary type.

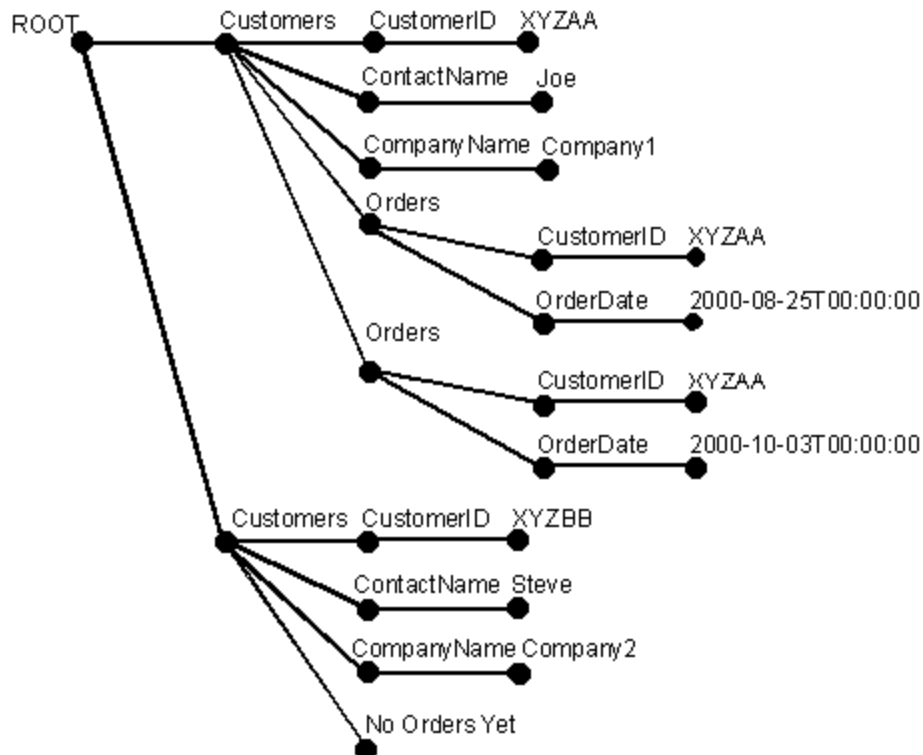
When you update an XML column, you might also want to validate the input XML document against a registered XML schema. You can do that with the XMLVALIDATE function.

You can use XML column values to specify which rows are to be updated. To find values within XML documents, you need to use XQuery expressions. One way of specifying XQuery expressions is the XMLEXISTS predicate, which allows you to specify an XQuery expression and determine if the expression results in an empty sequence. When XMLEXISTS is specified in the WHERE clause, rows will be updated if the XQuery expression returns a non-empty sequence.

```
<customerinfo xmlns="http://posample.org" Cid="1004">
  <name>Christine Haas</name>
  <addr country="Canada">
    <street>12 Topgrove</street>
    <city>Toronto</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N9Y-8G9</pcode-zip>
  </addr>
  <phone type="work">905-555-5238</phone>
  <phone type="home">416-555-2934</phone>
</customerinfo>
```

Retrieving Data in XML:

```
SELECT Customers.CustomerID, ContactName, CompanyName,
       Orders.CustomerID, OrderDate
FROM Customers, Orders
WHERE Customers.CustomerID = Orders.CustomerID
AND (Customers.CustomerID = N'ALFKI'
     OR Customers.CustomerID = N'XYZAA')
ORDER BY Customers.CustomerID
FOR XML AUTO
```



Converting SQL Data in XML format:

SQL to XML Converter

you can download SQL to XML converter from the web link like;
(<http://softwaresolution.informer.com/SQL-to-XML-Converter/>)
then convert desired format directly:

Step 1:

SQL Converter

Home About

Server name

Server Name

☒ Windows Authentication

☐ SQL Authentication

User

Password

Connect

Database

Test

View Content

Screen Shot for SQL Converter 1.9.

19

COM from C# Application

19.1 INTRODUCTION

Component Object Model (COM) is a binary-interface standard for software componentry introduced by Microsoft in 1993. It is used to enable interprocess communication and dynamic object creation in a large range of programming languages. The term COM is often used in the Microsoft software development industry as an umbrella term that encompasses the OLE, OLE Automation, ActiveX, COM+ and DCOM technologies

19.2 WHAT IS COM?

Microsoft COM (Component Object Model) technology in the Microsoft Windows-family of Operating Systems enables software components to communicate. COM is used by developers to create re-usable software components, link components together to build applications, and take advantage of Windows services. COM objects can be created with a variety of programming languages. Object-oriented languages, such as C++, provide programming mechanisms that simplify the implementation of COM objects. The family of COM technologies includes COM+, Distributed COM (DCOM) and ActiveX® Controls.

Microsoft provides COM interfaces for many Windows application programming interfaces such as Direct Show, Media Foundation, Packaging API, Windows Animation Manager, Windows Portable Devices, and Microsoft Active Directory (AD).

COM is used in applications such as the Microsoft Office Family of products. For example COM OLE technology allows Word documents to dynamically link to data in Excel spreadsheets and COM Automation allows users to build scripts in their applications to perform repetitive tasks or control one application from another.

The best resource for COM developers is the Microsoft Developer Network (MSDN). The MSDN Library contains information for developers on the Microsoft platform including a programming guide for COM development and the COM API programming reference. The Windows API is documented in Win32 and COM Development. You will also find information on COM+.

19.3 USING COM FROM .NET AND .NET FROM COM

The .NET Framework provides bi-directional interoperability with COM, which enables COM-based applications to use .NET components and .NET applications to use COM components. For information on how to access .NET components from COM see <http://msdn.microsoft.com/library/ms973802.aspx>. To learn how to use COM components from .NET see <http://msdn.microsoft.com/library/ms973800.aspx>.

19.4 RELATED TECHNOLOGIES

COM was the major software development platform for Windows and, as such, influenced development of a number of supporting technologies.

19.5 COM+

In order for Microsoft to provide developers with support for distributed transactions, resource pooling, disconnected applications, event publication and subscription, better memory and processor (thread) management, as well as to position Windows as an alternative to other enterprise-level operating systems, Microsoft introduced a technology called Microsoft Transaction Server (MTS) on Windows NT 4.

With Windows 2000, that significant extension to COM was incorporated into the operating system (as opposed to the series of external tools provided by MTS) and renamed COM+. At the same time, Microsoft de-emphasized DCOM as a separate entity. Components that made use of COM+ services were handled more directly by the added layer of COM+, in particular by operating system support for interception. In the first release of MTS, interception was tacked on - installing an MTS component would modify the Windows Registry to call the MTS software, and not the component directly.

Windows 2000 also revised the Component Services control panel application used to configure COM+ components.

An advantage of COM+ was that it could be run in "component farms". Instances of a component, if coded properly, could be pooled and reused by new calls to its initializing routine without unloading it from memory. Components could also be distributed (called from another machine). COM+ and Microsoft Visual Studio provided tools to make it easy to generate client-side proxies, so although DCOM was used to actually make the remote call, it was easy to do for developers.

COM+ also introduced a subscriber/publisher event mechanism called COM+ Events, and provided a new way of leveraging MSMQ (inter-application asynchronous messaging) with components called Queued Components. COM+ events extend the COM+ programming model to support late-bound events or method calls between the publisher or subscriber and the event system.

19.6 WHAT IS COM+?

COM+ is the name of the COM-based services and technologies first released in Windows 2000. COM+ brought together the technology of COM components and the application host of Microsoft Transaction Server (MTS). COM+ automatically handles programming tasks

such as resource pooling, disconnected applications, event publication and subscription and distributed transactions.

.NET

COM development has largely been superseded by Microsoft .NET, and Microsoft now focuses its marketing efforts on .NET, with .NET providing wrappers to the most commonly used COM controls. COM is still often used to hook up complex, high performance code to front end code implemented in Visual Basic or ASP. The .NET framework provides rapid development tools similar to Visual Basic for both Windows Forms and Web Forms with just-in-time compilation, back-end code can be implemented in any .NET Language including C#, Visual Basic and C++/CLI.

Despite this, COM remains a viable technology with an important software base. As of 2009, Microsoft has no plans for discontinuing either COM or support for COM. It is also ideal for script control of applications such as Office or Internet Explorer since it provides an interface for calling COM object methods from a script rather than requiring knowing the API at compile time. The GUID system used by COM has wide uses any time a unique ID is needed.

Several of the services that COM+ provides have been largely replaced by recent releases of .NET. For example, the System.Transactions namespace in .NET provides the TransactionScope class, which provides transaction management without resorting to COM+. Similarly, queued components can be replaced by Windows Communication Foundation with an MSMQ transport.

There is limited support for backward compatibility. A COM object may be used in .NET by implementing a runtime callable wrapper (RCW).[2] .NET objects that conform to certain interface restrictions may be used in COM objects by calling a COM callable wrapper (CCW).[3] From both the COM and .NET sides, objects using the other technology appear as native objects. See COM Interop.

WCF (Windows Communication Foundation) solves a number of COM's remote execution shortcomings, allowing objects to be transparently marshalled by value across process or machine boundaries.

USING COM COMPONENT FROM C#

This step-by-step process describes how to use a COM component from in Microsoft Visual Studio .NET by using Microsoft Visual C# .NET or in Microsoft Visual Studio 2005 by using Microsoft Visual C# 2005.

REQUIREMENTS

The following list outlines the recommended hardware, software, network infrastructure, and service packs that you may require:

- Visual C# .NET or Visual C# 2005

USING COM COMPONENTS FROM VISUAL STUDIO .NET

We can use COM components from in Microsoft Visual Studio .NET code by using the Microsoft .NET Framework Component Object Model (COM) interoperability layer (or COM Interop). Using Visual Studio .NET or Visual Studio 2005, we can easily access and use COM components.

1. Start Microsoft Visual Studio .NET or Microsoft Visual Studio 2005. Create a new **Console Application** in Visual C# .NET or in Visual C# 2005 and name the project MyCOMClient.
2. On the **Project** menu, click **Add Reference**.
3. In the **Add Reference** dialog box, click the **COM** tab. Notice that the ListView control lists all the COM components that are registered on the local computer in alphabetical order.
4. Locate and select the MyCOMComponent.dll file, click **Open**, and then click **OK** to close the dialog-box.

Note In Visual Studio 2005, you do not have to click **Open**.

5. In the Class1 Code window, add the following code to the Main function:
6. int mySum = 0;
7. MyCOMComponent.Class1Class myCOM = new
MyCOMComponent.Class1Class();
8. mySum = myCOM.Add(1,2);
- 9.
10. Console.WriteLine("1 + 2 = {0}", mySum.ToString());
11. Console.ReadLine();
12. On the **Debug** menu, click **Start** to build and run the application.

The following output appears in the Console window:

1 + 2 = 3

COMPLETE VISUAL C# .NET CODE LISTING

```
using System;
namespace MyCOMClient
{
    class Class1
    {
        static void Main(string[] args)
        {
            int mySum = 0;
            MyCOMComponent.Class1Class myCOM = new
MyCOMComponent.Class1Class();
            mySum = myCOM.Add(1,2);

            Console.WriteLine("1 + 2 = {0}", mySum.ToString());
        }
    }
}
```



```

        Console.ReadLine();
    }

}
}

```

CREATING COM COMPONENTS USING VISUAL C#.NET

Developers are sometimes asked to support older software systems that utilize obsolete technologies. This may be difficult when the development tools used to implement the older software system are not available and have been replaced by newer tools that do not seem to support the former tools' technologies. Faced with the need to replace a COM component that is used by VBScript in an ASP application, a developer may need to create the replacement using Visual Studio .NET. This guide is motivated at helping developers create COM components using Visual Studio.NET and C#.

CREATING THE COM OBJECT

Launch Microsoft Visual Studio .NET and create a new Visual C# Project with the Empty Project template. In our example we will create the project and name it "COMTest." Save the project.

Open a console window and navigate to the project's folder.

Create a key pair that will be used to sign the .NET assembly with a strong name. ["sn -k key.snk" at the command prompt]

Under Project->COMTest Properties, set the Output Type to Class Library. Output Type is found within General, which is found under Common Properties.

Add a class called "COMObject" to the COMTest project, which will create COMObject.cs.

Add an assembly attribute to COMObject.cs.

"[assembly:System.Reflection.AssemblyKeyFileAttribute(@"..\..\key.snk")]"

Generate a globally unique identifier for use with COMObject.

Start guidgen.exe (this should be easily done when opening a console window and entering "guidgen" at the command prompt).

Select the Registry Format option in the guidgen utility, generate a new guid, and copy the GUID.

Use System.Runtime.InteropServices.GuidAttribute to generate an attribute for class COMObject. Pass the GUID as a string to GUIDAttribute, but remove the curly braces that surround the GUID that was copied onto the clipboard.

Create a public member function for COMObject that is named "COMObjectFunction" with the following code:

```

public string COMObjectFunction()
{
    return "Hello, COM!";
}

```

Build the solution.

Register the Assembly. ["regasm COMTest.dll /tlb:COMTest.tlb /codebase COMTest" at the command prompt, while in the directory that contains COMTest.dll]

The assembly is now accessible using COM.

HOW TO USE COM COMPONENTS IN VISUAL STUDIO .NET WITH VISUAL C# .NET OR IN VISUAL STUDIO 2005 WITH VISUAL C# 2005

C# and .NET Security

20.1 INTRODUCTION

Introduction
Security Role
Code Security
Code Security Policy
Code Security Permissions
User Security

The Microsoft .NET Framework gives numerous techniques and a vast range of types in the security namespaces to help you build secure code and create secure Web applications. This chapter defines the .NET Framework security landscape by briefly introducing the security benefits of managed code development. This chapter also introduces and contrasts the two complimentary forms of security that are available to .NET Framework applications: user security and code security. Finally, the chapter briefly examines the security namespaces that you use to program .NET Framework security.

This chapter emphasizes how .NET Framework security applies to ASP.NET Web applications and Web services.

SECURITY ROLE

This chapter describes the security benefits inherent in using the .NET Framework and explains the complementary features of .NET Framework user (or *role-based*) security and .NET Framework code-based (or *code access*) security. We recommend that you use this chapter as follows:

- **Understand the two-layered defense provided by the .NET Framework.** Role-based security allows you to control user access to application resources and operations, while code access security can control which code can access resources and perform privileged operations.
- **Create applications that use the security concepts in this chapter.** This chapter tells you when you should use user-based security and when you should use code-based

security. After reading this chapter, you will be able to identify how any new applications you create can be more secure by using role-based or code-based security.

CODE SECURITY

Developing .NET Framework applications provides you with some immediate security benefits, although there are still many issues for you to think about. These issues are discussed in the Building chapters in Part III of this guide.

.NET Framework assemblies are built with managed code. Compilers for languages, such as the Microsoft Visual C#® development tool and Microsoft Visual Basic® .NET development system, output Microsoft intermediate language (MSIL) instructions, which are contained in standard Microsoft Windows portable executable (PE) .dll or .exe files. When the assembly is loaded and a method is called, the method's MSIL code is compiled by a just-in-time (JIT) compiler into native machine instructions, which are subsequently executed. Methods that are never called are not JIT-compiled.

The use of an intermediate language coupled with the run-time environment provided by the common language runtime offers assembly developers immediate security advantages.

- **File format and metadata validation.** The common language runtime verifies that the PE file format is valid and that addresses do not point outside of the PE file. This helps provide assembly isolation. The common language runtime also validates the integrity of the metadata that is contained in the assembly.
- **Code verification.** The MSIL code is verified for type safety at JIT compile time. This is a major plus from a security perspective because the verification process can prevent bad pointer manipulation, validate type conversions, check array bounds, and so on. This virtually eliminates buffer overflow vulnerabilities in managed code, although you still need to carefully inspect any code that calls unmanaged application programming interfaces (APIs) for the possibility of buffer overflow.
- **Integrity checking.** The integrity of strong named assemblies is verified using a digital signature to ensure that the assembly has not been altered in any way since it was built and signed. This means that attackers cannot alter your code in any way by directly manipulating the MSIL instructions.
- **Code access security.** The virtual execution environment provided by the common language runtime allows additional security checks to be performed at runtime. Specifically, code access security can make various run-time security decisions based on the identity of the calling code.

USER VS. CODE SECURITY

User security and code security are two complementary forms of security that are available to .NET Framework applications. User security answers the questions, "Who is the user and what can the user do?" while code security answers the questions "Where is the code from, who wrote the code, and what can the code do?" Code security involves authorizing the application's (not the user's) access to system-level resources, including the file system, registry, network, directory services, and databases. In this case, it does not matter who the

end user is, or which user account runs the code, but it does matter what the code is and is not allowed to do.

The .NET Framework user security implementation is called *role-based security*. The code security implementation is called *code access security*.

ROLE-BASED SECURITY

.NET Framework role-based security allows a Web application to make security decisions based on the identity or role membership of the user that interacts with the application. If your application uses Windows authentication, then a role is a Windows group. If your application uses other forms of authentication, then a role is application-defined and user and role details are usually maintained in SQL Server or user stores based on Active Directory.

The identity of the authenticated user and its associated role membership is made available to Web applications through **Principal** objects, which are attached to the current Web request.

Note When using the Role Manager feature in ASP.NET 2.0, if the roles are application-defined, the role information is not made available through Principal objects. Instead, the Role Manager obtains role information directly from the role store. For more information see [How To: Use Role Manager in ASP.NET 2.0](#).

Figure 6.1 shows a logical view of how user security is typically used in a Web application to restrict user access to Web pages, business operations, and data access.

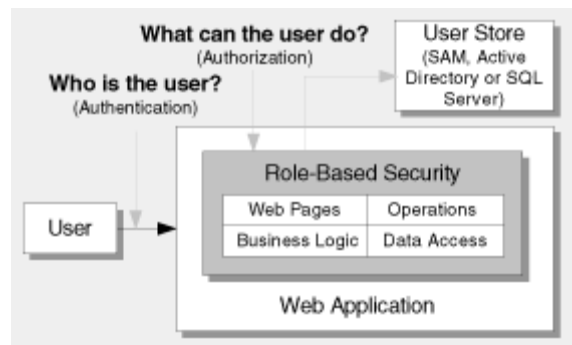


Figure 1.1

A logical view of (user) role-based security

CODE ACCESS SECURITY

Code access security authorizes code when it attempts to access secured resources, such as the file system, registry, network, and so on, or when it attempts to perform other privileged operations, such as calling unmanaged code or using reflection.

Code access security is an important additional defense mechanism that you can use to provide constraints on a piece of code. An administrator can configure code access security policy to restrict the resource types that code can access and the other privileged operations it can perform. From a Web application standpoint, this means that in the event of a compromised process where an attacker takes control of a Web application process or injects code to run inside the process, the additional constraints that code access security provides can limit the damage that can be done.

Figure 6.2 shows a logical view of how code access security is used in a Web application to constrain the application's access to system resources, resources owned by other applications, and privileged operations, such as calling unmanaged code.

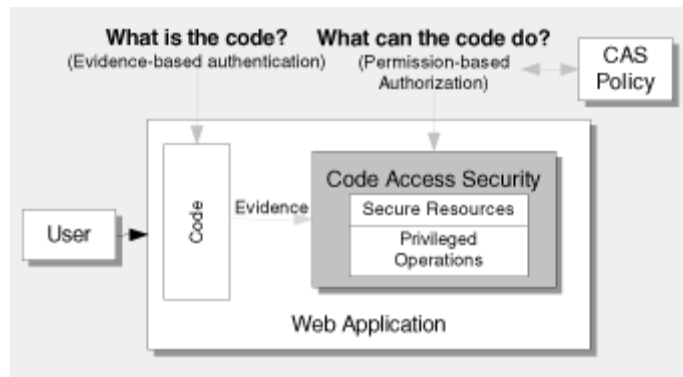


Figure 1.2

Logical view of code-based security

The authentication (identification) of code is based on evidence about the code, for example, its strong name, publisher, or installation directory. Authorization is based on the code access permissions granted to code by security policy. For more information about .NET Framework code access security, see Chapter 8, "[Code Access Security in Practice](#)."

.NET FRAMEWORK ROLE-BASED SECURITY

.NET Framework role-based security is a key technology that is used to authorize a user's actions in an application. Roles are often used to enforce business rules. For example, a financial application might allow only managers to perform monetary transfers that exceed a particular threshold.

Role-based security consists of the following elements:

- **Principals and identities**
- **PrincipalPermission objects**
- **Role-based security checks**
- **URL authorization**

PRINCIPALS AND IDENTITIES

Role-based security is implemented with **Principal** and **Identity** objects. The identity and role membership of the authenticated caller is exposed through a **Principal** object, which is attached to the current Web request. You can retrieve the object by using the **HttpContext.Current.User** property. If the caller is not required to authenticate with the application, for example, because the user is browsing a publicly accessible part of the site, the **Principal** object represents the anonymous Internet user.

Note When using the Role Manager feature in ASP.NET 2.0, if the roles are application-defined, the role information is not made available through Principal objects. Instead, the Role Manager obtains role information directly from the role store. For more information see [How To: Use Role Manager in ASP.NET 2.0](#).

There are many types of **Principal** objects and the precise type depends on the authentication mechanism used by the application. However, all **Principal** objects implement the **System.Security.Principal.IPrincipal** interface and they all maintain a list of roles of which the user is a member.

Principal objects also contain **Identity** objects, which include the user's name, together with flags that indicate the authentication type and whether or not the user has been authenticated. This allows you to distinguish between authenticated and anonymous users. There are different types of Identity objects, depending on the authentication type, although all implement the **System.Security.Principal.IIdentity** interface.

The following table shows the range of possible authentication types and the different types of **Principal** and **Identity** objects that ASP.NET Web applications use.

Table 1.1 Principal and Identity Objects Per Authentication Type

Authentication Type	Principal and Identity Type	Comments
Windows	WindowsPrincipal + WindowsIdentity	Verification of credentials is automatic and uses the Security Accounts Manager (SAM) or Active Directory. Windows groups are used for roles.
Forms	GenericPrincipal + FormsIdentity	You must add code to verify credentials and retrieve role membership from a user store. Note If you are using the Role Manager feature in ASP.NET 2.0, you do not need to write the code for retrieving user roles. For more information see How To: Use Role Manager in ASP.NET 2.0 .
Passport	GenericPrincipal + PassportIdentity	Relies on the Microsoft Passport SDK. PassportIdentity provides access to the passport authentication ticket.

PRINCIPAL PERMISSION OBJECTS

The **PrincipalPermission** object represents the identity and role that the current principal must have to execute code. **PrincipalPermission** objects can be used declaratively or imperatively in code.

Declarative Security

You can control precisely which users should be allowed to access a class or a method by adding a **PrincipalPermissionAttribute** to the class or method definition. A class-level attribute automatically applies to all class members unless it is overridden by a member-level attribute. The **PrincipalPermissionAttribute** type is defined within the **System.Security.Permissions** namespace.

Note You can also use the **PrincipalPermissionAttribute** to restrict access to structures and to other member types, such as properties and delegates.

The following example shows how to restrict access to a particular class to members of a **Managers** group. Note that this example assumes Windows authentication, where the format of the role name is in the format *MachineName\RoleName* or *DomainName\RoleName*. For other authentication types, the format of the role name is application specific and depends on the role-name strings held in the user store.

```
[PrincipalPermissionAttribute(SecurityAction.Demand,  
Role=@"DOMAINNAME\Managers")]  
public sealed class OnlyManagersCanCallMe  
{  
}
```

Note The trailing **Attribute** can be omitted from the attribute type names. This makes the attribute type name appear to be the same as the associated permission type name, which in this case is **PrincipalPermission**. They are distinct (but logically related) types.

The next example shows how to restrict access to a particular method on a class. In this example, access is restricted to members of the local administrators group, which is identified by the special "**BUILTIN\Administrators**" identifier.

```
[PrincipalPermissionAttribute(SecurityAction.Demand,  
Role=@"BUILTIN\Administrators")]  
public void SomeMethod()  
{  
}
```

Other built-in Windows group names can be used by prefixing the group name with "**BUILTIN**" (for example, "**BUILTIN\Users**" and "**BUILTIN\Power Users**").

IMPERATIVE SECURITY

If method-level security is not granular enough for your security requirements, you can perform imperative security checks in code by using **System.Security.Permissions.PrincipalPermission** objects.

The following example shows imperative security syntax using a **PrincipalPermission** object.

```
PrincipalPermission permCheck = new PrincipalPermission(  
    null, @"DomainName\WindowsGroup");  
permCheck.Demand();
```

To avoid a local variable, the code above can also be written as:

```
(new PrincipalPermission(null, @"DomainName\WindowsGroup")).Demand();
```

The code creates a **PrincipalPermission** object with a blank user name and a specified role name, and then calls the **Demand** method. This causes the common language runtime to interrogate the current **Principal** object that is attached to the current thread and check whether the associated identity is a member of the specified role. Because Windows authentication is used in this example, the role check uses a Windows group. If the current identity is not a member of the specified role, a **SecurityException** is thrown.

The following example shows how to restrict access to an individual user.

```
(new PrincipalPermission(@"DOMAINNAME\James", null)).Demand();
```

DECLARATIVE VS. IMPERATIVE SECURITY

You can use role-based security (and code access security) either declaratively using attributes or imperatively in code. Generally, declarative security offers the most benefits, although sometimes you must use imperative security (for example, when you need to use variables that are only available at runtime) to help make a security decision.

Advantages of Declarative Security

The main advantages of declarative security are the following:

- It allows the administrator or assembly consumer to see precisely which security permissions that particular classes and methods must run. Tools such as permview.exe provide this information. Knowing this information at deployment time can help resolve security issues and it helps the administrator configure code access security policy.
- It offers increased performance. Declarative demands are evaluated only once at load time. Imperative demands inside methods are evaluated each time the method that contains the demand is called.
- Security attributes ensure that the permission demand is executed before any other code in the method has a chance to run. This eliminates potential bugs where security checks are performed too late.
- Declarative checks at the class level apply to all class members. Imperative checks apply at the call site.

ADVANTAGES OF IMPERATIVE SECURITY

The main advantages of imperative security and the main reasons that you sometimes must use it are:

- It allows you to dynamically shape the demand by using values only available at runtime.
- It allows you to perform more granular authorization by implementing conditional logic in code.

ROLE-BASED SECURITY CHECKS

For fine-grained authorization decisions, you can also perform explicit role checks by using the **IPrincipal.IsInRole** method. The following example assumes Windows authentication, although the code would be very similar for Forms authentication, except that you would cast the **User** object to an object of the **GenericPrincipal** type.

```
// Extract the authenticated user from the current HTTP context.
// The User variable is equivalent to HttpContext.Current.User if you are using
// an .aspx or .asmx page
WindowsPrincipal authenticatedUser = User as WindowsPrincipal;
if (null != authenticatedUser)
{
    // Note: If you need to authorize specific users based on their identity
    // and not their role membership, you can retrieve the authenticated user's
    // username with the following line of code (normally though, you should
    // perform role-based authorization).
    // string username = authenticatedUser.Identity.Name;

    // Perform a role check
    if (authenticatedUser.IsInRole(@"DomainName\Manager") )
    {
        // User is authorized to perform manager functionality
    }
}
else
{
    // User is not authorized to perform manager functionality
    // Throw a security exception
}
```

Note When using the Role Manager feature in ASP.NET 2.0, you can also use the Roles API such as **Roles.IsUserInRole** for role checks. For more information, see "[How To: Use Role Manager in ASP.NET 2.0](#)."

URL AUTHORIZATION

Administrators can configure role-based security by using the **<authorization>** element in Machine.config or Web.config. This element configures the ASP.NET **UrlAuthorizationModule**, which uses the principal object attached to the current Web request in order to make authorization decisions.

The authorization element contains child **<allow>** and **<deny>** elements, which are used to determine which users or groups are allowed or denied access to specific directories or pages. Unless the **<authorization>** element is contained within a **<location>** element, the **<authorization>** element in Web.config controls access to the directory in which the Web.config file resides. This is normally the Web application's virtual root directory.

The following example from Web.config uses Windows authentication and allows Bob and Mary access but denies everyone else:

```
<authorization>
  <allow users="DomainName\Bob, DomainName\Mary" />
  <deny users="*" />
</authorization>
```

The following syntax and semantics apply to the configuration of the **<authorization>** element:

- **"*"** refers to all identities.
- **"?"** refers to unauthenticated identities (that is, the anonymous identity).
- You do not need to impersonate for URL authorization to work.
- Users and roles for URL authorization are determined by your authentication settings:

Note When using the Role Manager feature in ASP.NET 2.0, you only need to enable Role Manager and configure the role provider to point to the role store. Because the **IPrincipal** object internally uses the Role Manager, you do not need to explicitly set the role information in the Principal object. For more information see [How To: Use Role Manager in ASP.NET 2.0](#).

- When you have **<authentication mode="Windows" />**, you are authorizing access to Windows user and group accounts.

User names take the form *"DomainName\WindowsUserName"*.

Role names take the form *"DomainName\WindowsGroupName"*.

Note The local administrators group is referred to as *"BUILTIN\Administrators"*. The local users group is referred to as *"BUILTIN\Users"*.

- When you have **<authentication mode="Forms" />**, you are authorizing against the user and roles for the **IPrincipal** object that was stored in the current HTTP context. For example, if you used Forms to authenticate users against a database, you will be authorizing against the roles retrieved from the database.
- When you have **<authentication mode="Passport" />**, you authorize against the Passport User ID (PUID) or roles retrieved from a store. For example, you

can map a PUID to a particular account and set of roles stored in a Microsoft SQL Server database or Active Directory.

- When you have `<authentication mode="None" />`, you may not be performing authorization. "None" specifies that you do not want to perform any authentication or that you do not want to use any of the ASP.NET authentication modules, but you do want to use your own custom mechanism.

However, if you use custom authentication, you should create an **IPrincipal** object with roles and store it in the **HttpContext.Current.User** property. When you subsequently perform URL authorization, it is performed against the user and roles (no matter how they were retrieved) maintained in the **IPrincipal** object.

CONFIGURING ACCESS TO A SPECIFIC FILE

To configure access to a specific file, place the `<authorization>` element inside a `<location>` element as shown below.

```
<location path="somepage.aspx" />
  <authorization>
    <allow users="DomainName\Bob, DomainName\Mary" />
    <deny users="*" />
  </authorization>
</location>
```

You can also point the **path** attribute at a specific folder to apply access control to all the files in that particular folder. For more information about the `<location>` element, see Chapter 19, "[Securing Your ASP.NET Application and Web Services](#)."

.NET FRAMEWORK SECURITY NAMESPACES

To program .NET Framework security, you use the types in the .NET Framework security namespaces. This section introduces these namespaces and the types that you are likely to use when you develop secure Web applications. For a full list of types, see the .NET Framework documentation. The security namespaces are listed below and are shown in Figure 6.3.

- **System.Security**
- **System.Web.Security**
- **System.Security.Cryptography**
- **System.Security.Principal**
- **System.Security.Policy**
- **System.Security.Permissions**

Note The .NET Framework 2.0 has introduced new security-related namespaces such as **System.Security.AccessControl**, **System.Security.Cryptography.Pkcs**, **System.Security.Cryptography.X509Certificates**, **System.Security.Cryptography.Xml**, and **System.Net.Security**. For more information, see [.NET Framework Class Library](#).

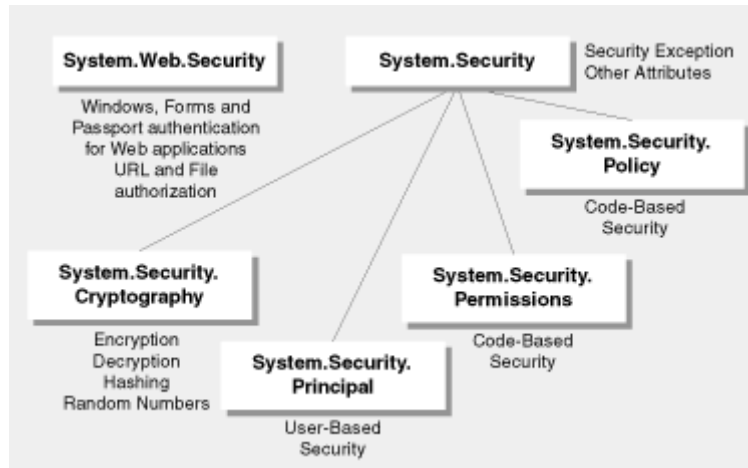


Figure 1.3

.NET Framework security namespaces in .NET 1.1

SYSTEM.SECURITY

This namespace contains the **CodeAccessPermission** base class from which all other code access permission types derive. You are unlikely to use the base class directly. You are more likely to use specific permission types that represent the rights of code to access specific resource types or perform other privileged operations. For example, **FileIOPermission** represents the rights to perform file I/O, **EventLogPermission** represents the rights for code to access the event log, and so on. For a full list of code access permission types, see Table 6.2 later in this chapter.

The **System.Security** namespace also contains classes that encapsulate permission sets. These include the **PermissionSet** and **NamedPermissionSet** classes. The types you are most likely to use when building secure Web applications are:

- **SecurityException.** The exception type used to represent security errors.

Note In .NET 2.0, the **SecurityException** class has been improved to facilitate debugging security issues and provide detailed information about security exceptions.

- **AllowPartiallyTrustedCallersAttribute.** An assembly-level attribute used with strong named assemblies that must support partial trust callers. Without this attribute, a strong named assembly can only be called by full trust callers (callers with unrestricted permissions.)
- **SupressUnmanagedSecurityAttribute.** Used to optimize performance and eliminate the demand for the unmanaged code permission issued by the Platform Invocation Services (P/Invoke) and Component Object Model (COM) interoperability layers. This attribute must be used with caution because it exposes a potential security risk. If an attacker gains control of unmanaged code, he is no longer restricted by code

access security. For more information about using this attribute safely, see "Unmanaged Code" in Chapter 8, "[Code Access Security in Practice](#)."

Note In .NET 2.0, the **System.Security** namespace has new classes such as **SecurityContext** and **SecureString**. For more information, see [.NET Framework Class Library - System.Security Namespace](#).

SYSTEM.WEB.SECURITY

This namespace contains the classes used to manage Web application authentication and authorization. This includes Windows, Forms, and Passport authentication and URL and File authorization, which are controlled by the **UrlAuthorizationModule** and **FileAuthorizationModule** classes, respectively. The types you are most likely to use when you build secure Web applications are:

- **FormsAuthentication**. Provides static methods to help with Forms authentication and authentication ticket manipulation.
- **FormsIdentity**. Used to encapsulate the user identity that is authenticated by Forms authentication.
- **PassportIdentity**. Used to encapsulate the user identity that is authenticated by Passport authentication.

SYSTEM.SECURITY.CRYPTOGRAPHY

This namespace contains types that are used to perform encryption and decryption, hashing, and random number generation. This is a large namespace that contains many types. Many encryption algorithms are implemented in managed code, while others are exposed by types in this namespace that wrap the underlying cryptographic functionality provided by the Microsoft Win32®-based CryptoAPI.

SYSTEM.SECURITY.PRINCIPAL

This namespace contains types that are used to support role-based security. They are used to restrict which users can access classes and class members. The namespace includes the **IPrincipal** and **IIdentity** interfaces. The types you are most likely to use when building secure Web applications are:

- **GenericPrincipal** and **GenericIdentity**. Allow you to define your own roles and user identities. These are typically used with custom authentication mechanisms.
- **WindowsPrincipal** and **WindowsIdentity**. Represents a user who is authenticated with Windows authentication together with the user's associated Windows group (role) list.

Note In .NET 2.0, the **WindowsIdentity** class now supports a new constructor that accepts a user name represented by the user principal name (UPN). This constructor uses the Kerberos S4U (Service-for-User) extension to obtain a Windows token for the user.

SYSTEM.SECURITY.POLICY

This namespace contains types that are used to implement the code access security policy system. It includes types to represent code groups, membership conditions, policy levels, and evidence.

SYSTEM.SECURITY.PERMISSIONS

This namespace contains the majority of permission types that are used to encapsulate the rights of code to access resources and perform privileged operations. The following table shows the permission types that are defined in this namespace (in alphabetical order).

Table 1.2 Permission Types Within the System.Security.Permissions Namespace

Permission	Description
DirectoryServicesPermission	Required to access Active Directory.
DNSPermission	Required to access domain name system (DNS) servers on the network.
EndpointPermission	Defines an endpoint that is authorized by a SocketPermission object.
EnvironmentPermission	Controls read and write access to individual environment variables. It can also be used to restrict all access to environment variables.
EventLogPermission	Required to access the event log.
FileDialogPermission	Allows read-only access to files only if the file name is specified by the interactive user through a system-provided file dialog box. It is normally used when FileIOPermission is not granted.
FileIOPermission	Controls read, write, and append access to files and directory trees. It can also be used to restrict all access to the file system.
IsolatedStorageFilePermission	Controls the usage of an application's private virtual file system (provided by isolated storage). Isolated storage creates a unique and private storage area for the sole use by an application or component.
IsolatedStoragePermission	Required to access isolated storage.
MessageQueuePermission	Required to access Microsoft Message Queuing message queues.
OdbcPermission	Required to use the ADO.NET ODBC data provider. (Full trust is also required.)
OleDbPermission	Required to use the ADO.NET OLE DB data provider. (Full trust is also required.)
OraclePermission	Required to use the ADO.NET Oracle data provider. (Full trust is also required.)
PerformanceCounterPermission	Required to access system performance counters.

PrincipalPermission	Used to restrict access to classes and methods based on the identity and role membership of the user.
PrintingPermission	Required to access printers.
ReflectionPermission	Controls access to metadata. Code with the appropriate ReflectionPermission can obtain information about the public, protected, and private members of a type.
RegistryPermission	Controls read, write, and create access to registry keys (including subkeys). It can also be used to restrict all access to the registry.
SecurityPermission	This is a meta-permission that controls the use of the security infrastructure itself.
ServiceControllerPermission	Can be used to restrict access to the Windows Service Control Manager and the ability to start, stop, and pause services.
SocketPermission	Can be used to restrict the ability to make or accept a connection on a transport address.
SqlClientPermission	Can be used to restrict access to SQL Server data sources.
UIPermission	Can be used to restrict access to the clipboard and to restrict the use of windows to "safe" windows in an attempt to avoid attacks that mimic system dialog boxes that prompt for sensitive information such as passwords.
WebPermission	Can be used to control access to HTTP Internet resources.

The **SecurityPermission** class warrants special attention because it represents the rights of code to perform privileged operations, including asserting code access permissions, calling unmanaged code, using reflection, and controlling policy and evidence, among others. The precise right determined by the **SecurityPermission** class is determined by its **Flags** property, which must be set to one of the enumerated values defined by the **SecurityPermissionFlags** enumerated type (for example, **SecurityPermissionFlags.UnmanagedCode**).

Note The .NET Framework 2.0 has introduced new permissions such as **DataProtectionPermission**, **GacIdentityPermission**, **KeyContainerPermission**, and **StorePermission**. For more information, see [.NET Framework Class Library - System.Security.Permissions Namespace](#).

SUMMARY

This chapter has introduced you to the .NET Framework security landscape by contrasting user security and code security and by examining the security namespaces. The .NET Framework refers to these two types of security as role-based security and code access security, respectively. Both forms of security are layered on top of Windows security.

Role-based security is concerned with authorizing user access to application-managed resources (such as Web pages) and operations (such as business and data access logic). Code access security is concerned with constraining privileged code and controlling precisely which code can access resources and perform other privileged operations. This is a powerful

additional security mechanism for Web applications because it restricts what an attacker is able to do, even if the attacker manages to compromise the Web application process. It is also an extremely powerful feature for providing application isolation. This is particularly true for hosting companies or any organization that hosts multiple Web applications on the same Web server.

Authentication, Authorization, User and Role Managment and general Security in .NET

I need to know how to go about implementing general security for a C# application. What options do I have in this regard? I would prefer to use an existing framework if it meets my needs - I don't want to re-invent the wheel.

My requirements are as follows:

- the usual username/password authentication
- managing of users - assign permissions to users
- managing of roles - assign users to roles, assign permissions to roles
- authorization of users based on their username and role

I am looking for a free / open-source framework/library that has been time-tested and used by the .Net community.

15 down My application takes a client/server approach, with the server running as a
vote windows service, connecting to a SQL Server database. Communication between
favorite client and server will be through WCF.

14

One other thing that is important is that I need to be able to assign specific users or roles permissions to View/Update/Delete a specific entity, whether it be a Customer, or Product etc. For e.g. Jack can view a certain 3 of 10 customers, but only update the details of customers Microsoft, Yahoo and Google, and can only delete Yahoo.

c# security authorization roles user

asked Aug 3 '09 at 15:29

share | improve this question edited Aug 3 '09 at 15:32



Saajid Ismail
1,48521735

1 Just want to make sure you know: C# is the language. It doesn't have security. .NET is the platform. It's where the security is. - John Saunders Aug 3 '09 at 15:32

Thnx John - I understand the difference. - Saajid Ismail Aug 3 '09 at 17:24

6 Answers

active oldest votes

For coarse-grained security, you might find the inbuilt principal code useful; the user object (and their roles) are controlled in .NET by the "principal", but usefully the runtime itself can enforce this.

The implementation of a principal can be implementation-defined, and you can usually inject your own; for example in WCF.

To see the runtime enforcing coarse access (i.e. which *functionality* can be accessed, but not limited to which specific *data*):

```
static class Roles {
    public const string Administrator = "ADMIN";
}
static class Program {
    static void Main() {
        Thread.CurrentPrincipal = new GenericPrincipal(
            new GenericIdentity("Fred"), new string[] { Roles.Administrator });
        DeleteDatabase(); // fine
        Thread.CurrentPrincipal = new GenericPrincipal(
            new GenericIdentity("Barney"), new string[] { });
        DeleteDatabase(); // boom
    }
}
[PrincipalPermission(SecurityAction.Demand, Role = Roles.Administrator)]
public static void DeleteDatabase()
{
    Console.WriteLine(
        Thread.CurrentPrincipal.Identity.Name + " has deleted the
        database...");
}
```

up vote 17
down vote
accepted
+100

However, this doesn't help with the fine-grained access (i.e. "Fred can access customer A but not customer B").

Additional; Of course, for fine-grained, you can simply check the required roles at runtime, by checking `IsInRole` on the principal:

```
static void EnforceRole(string role)
{
    if (string.IsNullOrEmpty(role)) { return; } // assume anon OK
    IPrincipal principal = Thread.CurrentPrincipal;
    if (principal == null || !principal.IsInRole(role))
```

```

        {
            throw new SecurityException("Access denied to role: " + role);
        }
    }
}
public static User GetUser(string id)
{
    User user = Repository.GetUser(id);
    EnforceRole(user.AccessRole);
    return user;
}

```

You can also write your own principal / identity objects that do lazy tests / caching of the roles, rather than having to know them all up-front:

```

class CustomPrincipal : IPrincipal, IIdentity
{
    private string cn;
    public CustomPrincipal(string cn)
    {
        if (string.IsNullOrEmpty(cn)) throw new ArgumentNullException("cn");
        this.cn = cn;
    }
    // perhaps not ideal, but serves as an example
    readonly Dictionary<string, bool> roleCache =
        new Dictionary<string, bool>();
    public override string ToString() { return cn; }
    bool IIdentity.IsAuthenticated { get { return true; } }
    string IIdentity.AuthenticationType { get { return "iris scan"; } }
    string IIdentity.Name { get { return cn; } }
    IIdentity IPrincipal.Identity { get { return this; } }

    bool IPrincipal.IsInRole(string role)
    {
        if (string.IsNullOrEmpty(role)) return true; // assume anon OK
        lock (roleCache)
        {
            bool value;
            if (!roleCache.TryGetValue(role, out value)) {
                value = RoleHasAccess(cn, role);
                roleCache.Add(role, value);
            }
            return value;
        }
    }
    private static bool RoleHasAccess(string cn, string role)
    {
        //TODO: talk to your own security store
    }
}

```

[share](#) | [improve this answer](#)

Introduction

Over the past years, I've learned many things from CodeProject ... and now I'm giving back to the CodeProject. Since I didn't find any articles on Code Access Security, here's my one. Enjoy!

I'm not going to bore you with theory, but before we wet our feet, there are some concepts, keywords that you should learn. .NET has two kinds of security:

1. **Role Based Security** (not being discussed in this article)
2. **Code Access Security**

The Common Language Runtime (CLR) allows code to perform only those operations that the code has permission to perform. So CAS is the CLR's security system that enforces security policies by preventing unauthorized access to protected **resources** and **operations**. Using the Code Access Security, you can do the following:

- **Restrict what your code can do**
- **Restrict which code can call your code**
- **Identify code**

We'll be discussing about these things through out this article. Before that, you should get familiar with the jargon.

Jargon

Code access security consists of the following elements:

- **permissions**
- **permission sets**
- **code groups**
- **evidence**
- **policy**

Permissions

Permissions represent access to a protected resource or the ability to perform a protected operation. The .NET Framework provides several permission classes, like FileIOPermission (when working with files), UIPermission (permission to use a user interface), SecurityPermission (this is needed to execute the code and can be even used to bypass security) etc. I won't list all the permission classes here, they are listed below.

Permission sets

A permission set is a collection of permissions. You can put FileIOPermission and UIPermission into your own permission set and call it "My_PermissionSet". A permission set can include any number of permissions. FullTrust, LocalIntranet, Internet, Execution and Nothing are some of the built in permission sets in .NET Framework. FullTrust has all the permissions in the world, while Nothing has no permissions at all, not even the right to execute.

Code groups

Code group is a logical grouping of code that has a specified condition for membership. Code from <http://www.somewebsite.com/> can belong to one code group, code containing a specific strong name can belong to another code group and code from a specific assembly can belong to another code group. There are built-in code groups like My_Computer_Zone, LocalIntranet_Zone, Internet_Zone etc. Like permission sets, we can create code groups to meet our requirements based on the **evidence** provided by .NET Framework. Site, Strong Name, Zone, URL are some of the types of evidence.

Policy

Security policy is the configurable set of rules that the CLR follows when determining the permissions to grant to code. There are four policy levels - Enterprise, Machine, User and Application Domain, each operating independently from each other. Each level has its own code groups and permission sets. They have the hierarchy given below.

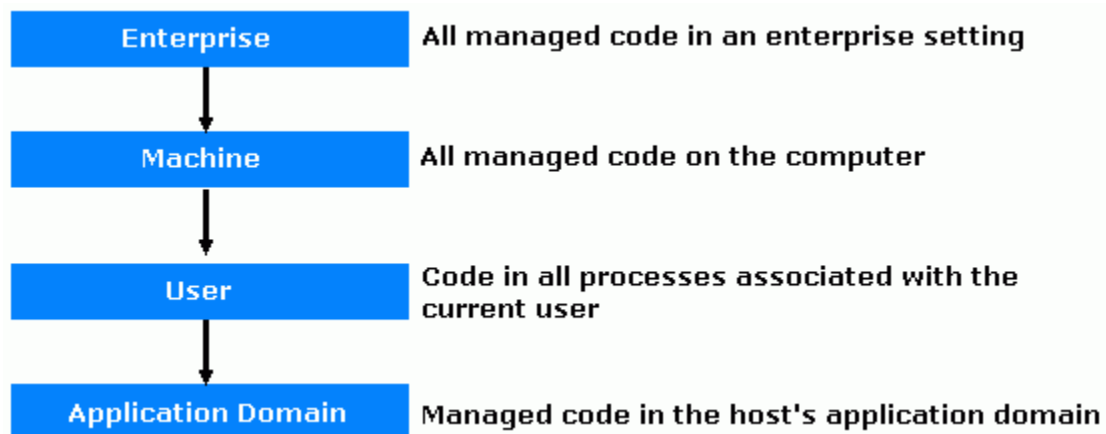


Figure 1.4

Okay, enough with the theory, it's time to put the theory into practice.

Quick Example

Let's create a new Windows application. Add two buttons to the existing form. We are going to work with the file system, so add the System.IO namespace.

☒Collapse | [Copy Code](#)
using System.IO;

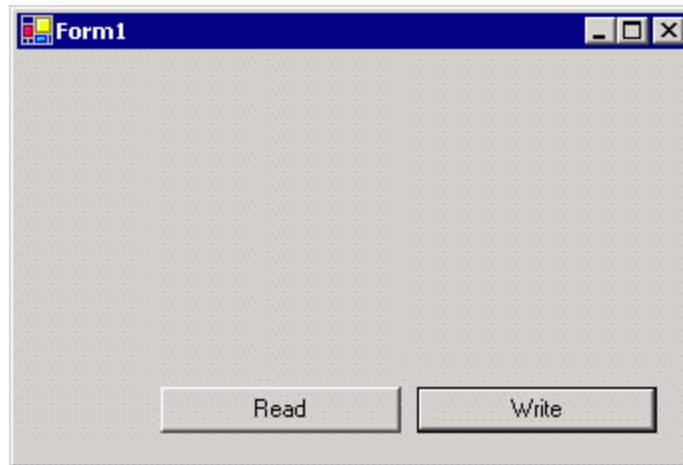


Figure 1.5

Write the following code:

[Collapse](#) | [Copy Code](#)

```
private void btnWrite_click(object sender, System.EventArgs e)
{
    StreamWriter myFile = new StreamWriter("c:\\Security.txt");
    myFile.WriteLine("Trust No One");
    myFile.Close();
}

private void btnRead_click(object sender, System.EventArgs e)
{
    StreamReader myFile = new StreamReader("c:\\Security.txt");
    MessageBox.Show(myFile.ReadLine())
    myFile.Close()
}
```

The version number should be intact all the time, for our example to work. Make sure that you set the version number to a fixed value, otherwise it will get incremented every time you compile the code. We're going to sign this assembly with a strong name which is used as evidence to identify our code. That's why you need to set the version number to a fixed value.

[Collapse](#) | [Copy Code](#)

```
[assembly: AssemblyVersion("1.0.0.0")]
```

That's it ... nothing fancy. This will write to a file named *Security.txt* in C: drive. Now run the code, it should create a file and write the line, everything should be fine ... unless of course you don't have a C: drive. Now what we are going to do is put our assembly into a code group and set some permissions. Don't delete the *Security.txt* file yet, we are going to need it later. Here we go.

.NET Configuration Tool

We can do this in two ways, from the **.NET Configuration Tool** or from the command prompt using **caspol.exe**. First we'll do this using the .NET Configuration Tool. Go to Control Panel --> Administrative Tools --> Microsoft .NET Framework Configuration. You can also type "**mscorcfg.msc**" at the .NET command prompt. You can do cool things with this tool ... but right now we are only interested in setting code access security.

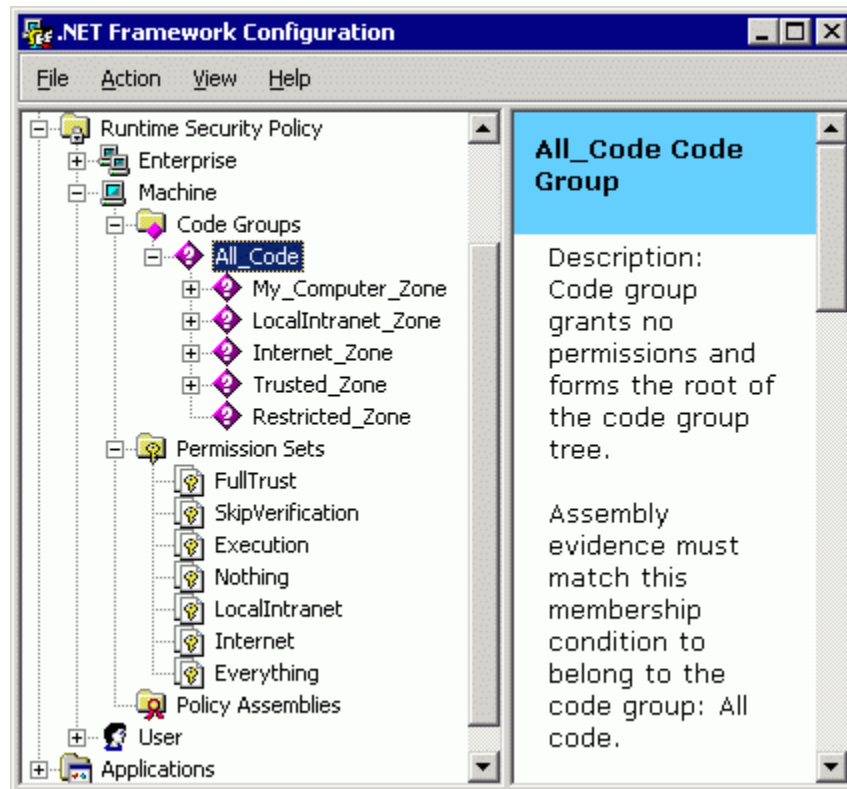
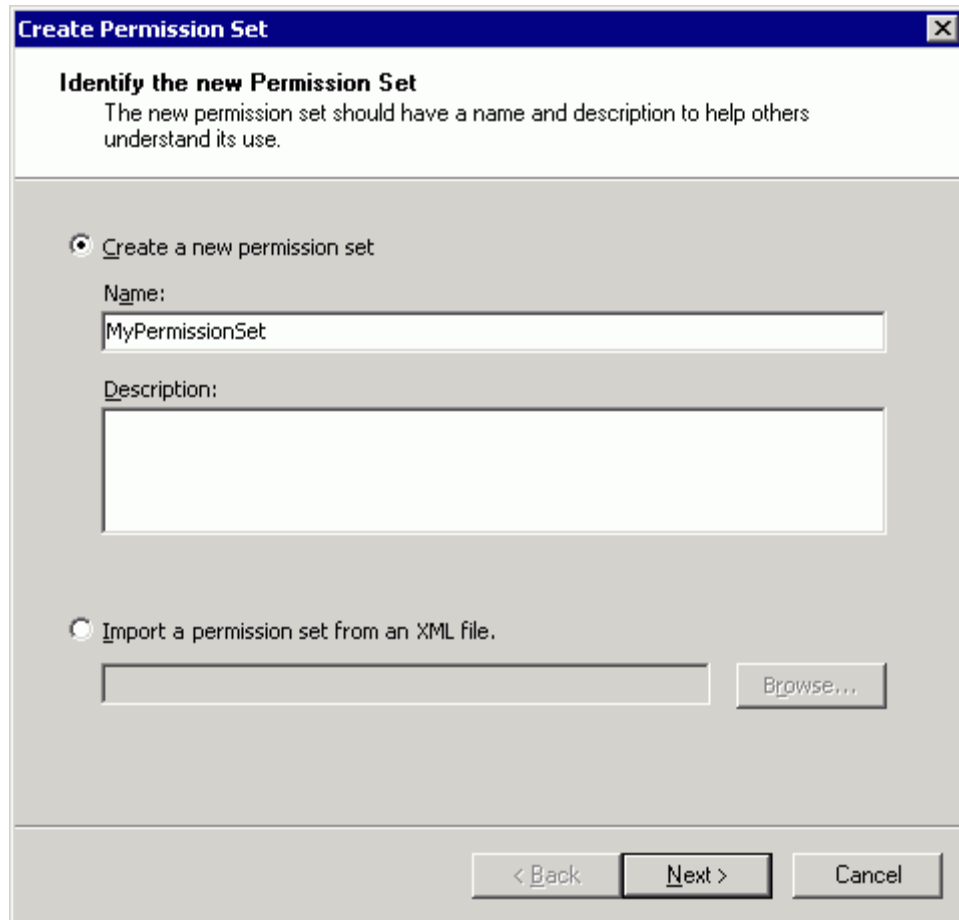


Figure 1.6

Creating a new permission set

Expand the **Runtime Security Policy** node. You can see the security policy levels - Enterprise, Machine and User. We are going to change the security settings in Machine policy. First we are going to create our own custom permission set. Right click the **Permission Sets** node and choose **New**. Since I couldn't think of a catchy name, I'm going to name it MyPermissionSet.



The image shows a Windows-style dialog box titled "Create Permission Set". It has a blue title bar with a close button (X) in the top right corner. The main content area is divided into two sections. The top section, titled "Identify the new Permission Set", contains the text: "The new permission set should have a name and description to help others understand its use." Below this, there are two radio buttons. The first radio button is selected and is labeled "Create a new permission set". Below this radio button are two text input fields: "Name:" with the text "MyPermissionSet" entered, and "Description:" which is currently empty. The second radio button is labeled "Import a permission set from an XML file." and is not selected. Below this radio button is an empty text input field and a "Browse..." button. At the bottom of the dialog box, there are three buttons: "< Back", "Next >", and "Cancel".

Create Permission Set

Identify the new Permission Set
The new permission set should have a name and description to help others understand its use.

☒ Create a new permission set

Name:
MyPermissionSet

Description:

☐ Import a permission set from an XML file.

Browse...

< Back Next > Cancel

Figure 1.7

In the next screen, we can add permissions to our permission set. In the left panel, we can see all the permissions supported by the .NET Framework. Now get the properties of **File IO** permission. Set the **File Path** to C:\ and check **Read** only, don't check others. So we didn't give write permission, we only gave read permission. Please note that there is another option saying "**Grant assemblies unrestricted access to the file system.**" If this is selected, anything can be done without any restrictions for that particular resource, in this case the file system.

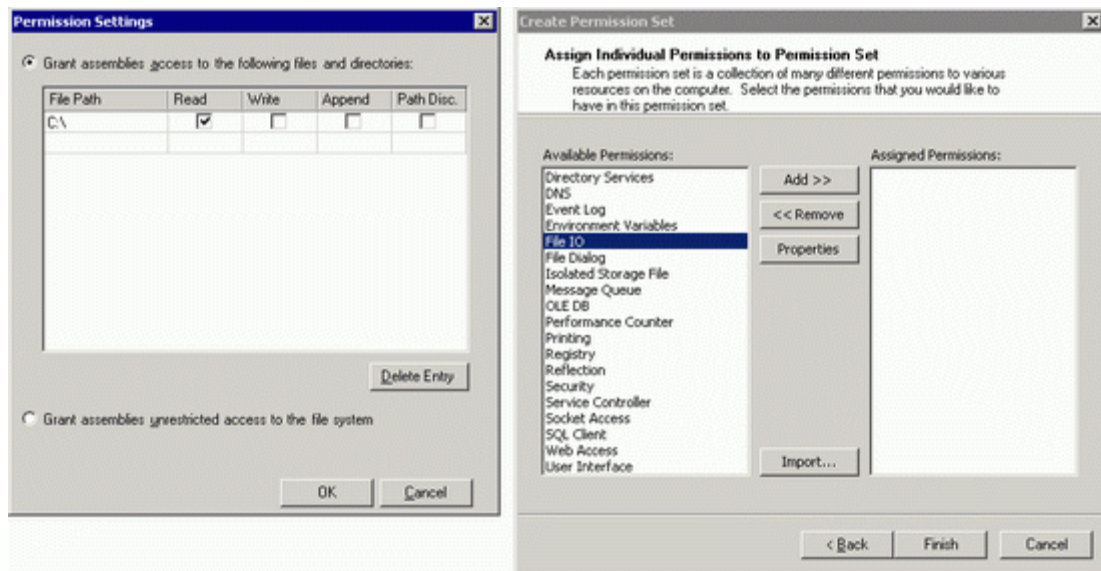


Figure 1.8

Now we have to add two more permissions - **Security** and **User Interface**. Just add them and remember to set the "**Grant assemblies unrestricted access**". I'll explain these properties soon. Without the **Security** permission, we don't have the right to execute our code, and without the **User Interface** permission, we won't be able to show a UI. If you're done adding these three permissions, you can see there is a new permission set created, named MyPermissionSet.

Creating a new code group

Now we will create a code group and set some conditions, so our assembly will be a member of that code group. Notice that in the code groups node, **All_Code** is the parent node. Right Click the **All_Code** node and choose **New**. You'll be presented with the **Create Code Group wizard**. I'm going to name it MyCodeGroup.

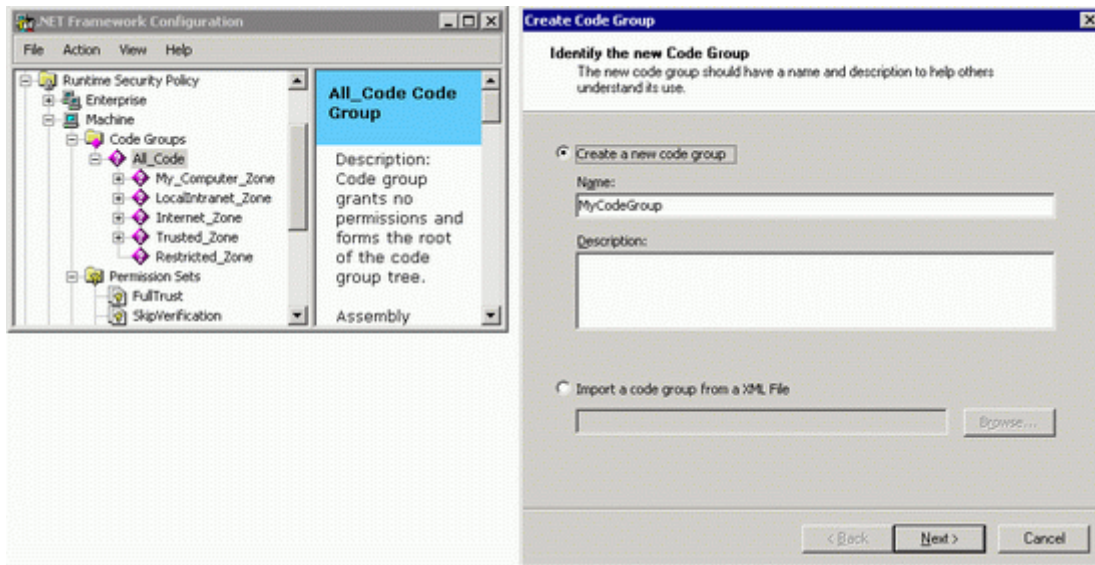


Figure 1.9

In the next screen, you have to provide a condition type for the code group. Now these are the **evidence** that I mentioned earlier. For this example, we are going to use the **Strong Name** condition type. First, sign your assembly with a strong name and build the project. Now press the **Import** button and select your assembly. Public Key, Name and Version will be extracted from the assembly, so we don't have to worry about them. Now move on to the next screen. We have to specify a permission set for our code group. Since we have already created one - MyPermissionSet, select it from the list box.

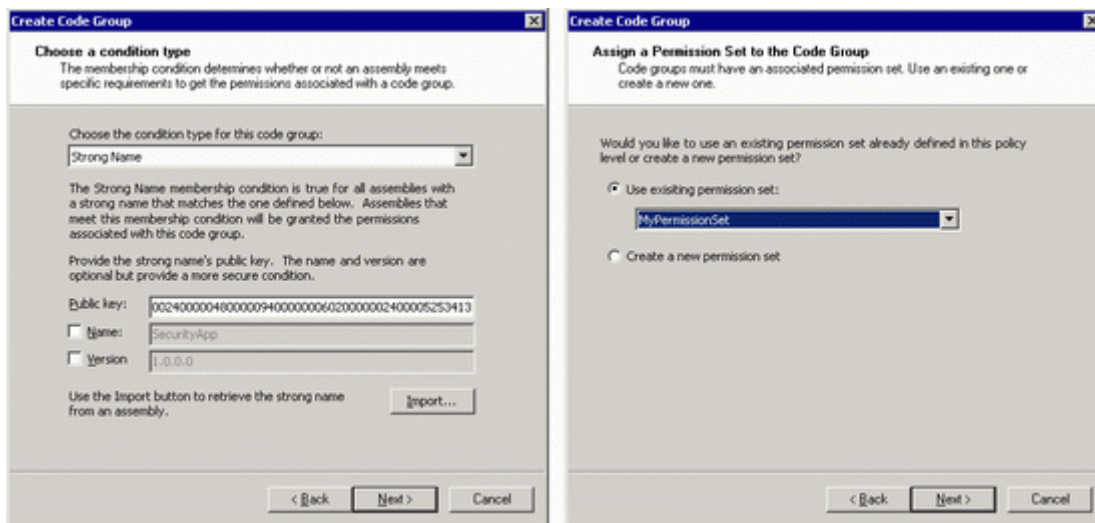


Figure 1.9

Exclusive and LevelFinal

If you haven't messed around with the default .NET configuration security settings, your assembly already belongs to another built-in code group - My_Computer_Zone. When

permissions are calculated, if a particular assembly falls into more than one code group within the same policy level, the final permissions for that assembly will be the union of all the permissions in those code groups. I'll explain how to calculate permissions later, for the time being we only need to run our assembly only with our permission set and that is MyPermissionSet associated with the MyCodeGroup. So we have to set another property to do just that. Right click the newly created **MyCodeGroup** node and select **Properties**. Check the check box saying "This policy level will only have the permissions from the permission set associated with this code group." This is called the Exclusive attribute. If this is checked then the run time will never grant more permissions than the permissions associated with this code group. The other option is called LevelFinal. These two properties come into action when calculating permissions and they are explained below in detail.

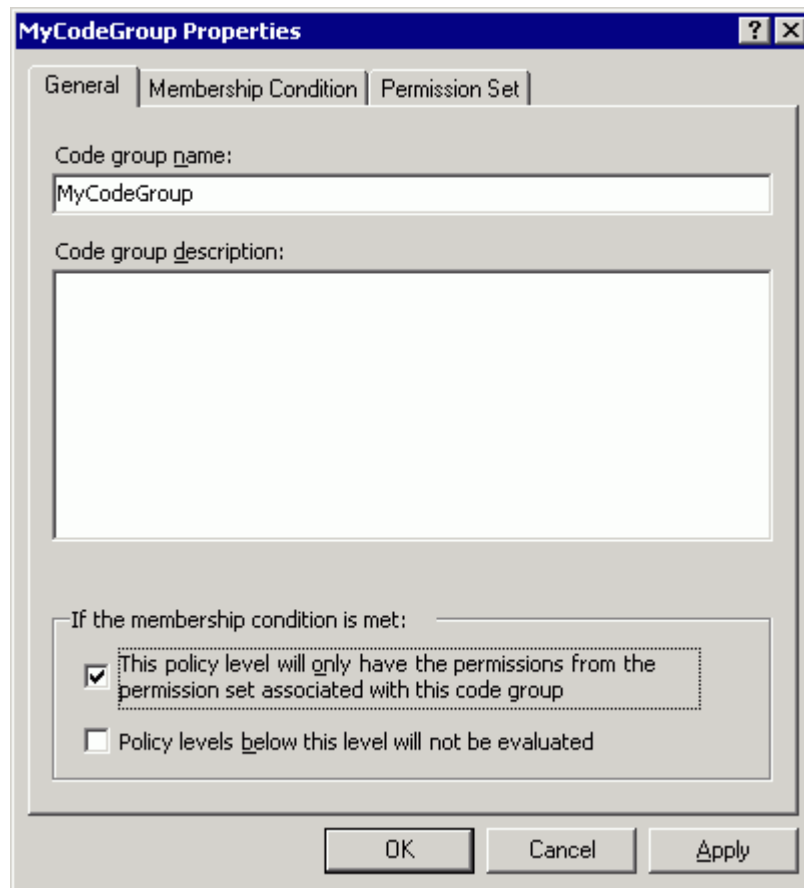


Figure 1.10

I know we have set lots of properties, but it'll all make sense at the end (hopefully).

Okay .. it's time to run the code. What we have done so far is, we have put our code into a code group and given permissions only to read from C: drive. Run the code and try both buttons. **Read** should work fine, but when you press **Write**, an exception will be thrown because we didn't set permission to write to C: drive. Below is the error message that you get.

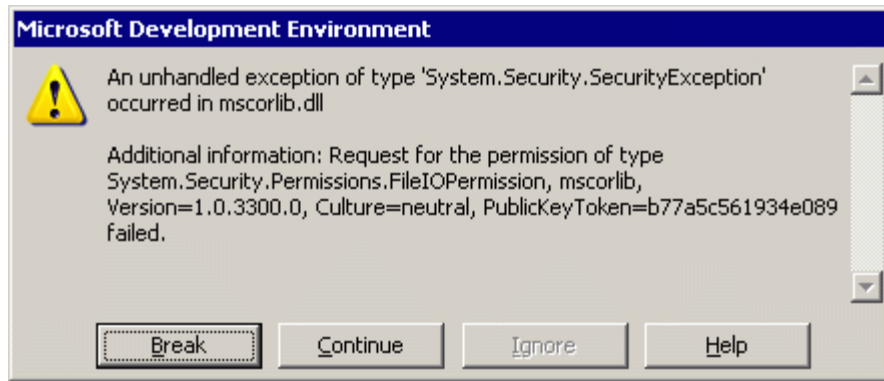


Figure 1.11

So thanks to Code Access Security, this kind of restriction to a resource is possible. There's a whole lot more that you can do with Code Access Security, which we're going to discuss in the rest of this article.

Functions of Code Access Security

According to the documentation, Code Access Security performs the following functions: (straight from the documentation)

- Defines **permissions** and **permission sets** that represent the right to access various system resources.
- Enables administrators to configure **security policy** by associating sets of permissions with groups of code (**code groups**).
- Enables code to **request the permissions it requires in order to run**, as well as the **permissions that would be useful to have**, and specifies which **permissions the code must never have**.
- **Grants permissions** to each assembly that is loaded, based on the permissions requested by the code and on the operations permitted by security policy.
- Enables code to **demand** that its callers have specific permissions. Enables code to demand that its callers possess a digital signature, thus allowing only callers from a particular organization or site to call the protected code.
- **Enforces restrictions** on code at run time by comparing the granted permissions of every caller on the call stack to the **permissions that callers must have**.

We have already done the top two, and that is the administrative part. There's a separate namespace that we haven't looked at yet - System.Security, which is dedicated to implementing security.

Security Namespace

These are the main classes in System.Security namespace:

Classes	Description
CodeAccessPermission	Defines the underlying structure of all code

	access permissions.
PermissionSet	Represents a collection that can contain many different types of permissions.
SecurityException	The exception that is thrown when a security error is detected.

These are the main classes in System.Security.Permissions namespace:

Classes	Description
EnvironmentPermission	Controls access to system and user environment variables.
FileDialogPermission	Controls the ability to access files or folders through a file dialog.
FileIOPermission	Controls the ability to access files and folders.
IsolatedStorageFilePermission	Specifies the allowed usage of a private virtual file system.
IsolatedStoragePermission	Represents access to generic isolated storage capabilities.
ReflectionPermission	Controls access to metadata through the System.Reflection APIs.
RegistryPermission	Controls the ability to access registry variables.
SecurityPermission	Describes a set of security permissions applied to code.
UIPermission	Controls the permissions related to user interfaces and the clipboard.

You can find more permission classes in other namespaces. For example, SocketPermission and WebPermission in System.Net namespace, SqlClientPermission in System.Data.SqlClient namespace, PerformanceCounterPermission in System.Diagnostics namespace etc. All these classes represent a protected resource.

Next, we'll see how we can use these classes.

Declarative vs. Imperative

You can use two different kinds of syntax when coding, declarative and imperative.

Declarative syntax

Declarative syntax uses attributes to mark the method, class or the assembly with the necessary security information. So when compiled, these are placed in the metadata section of the assembly.

⊞Collapse | [Copy Code](#)

```
[FileIOPermission(SecurityAction.Demand, Unrestricted=true)]
public class MyClass
{
    public MyClass() {...} // all these methods
    public void MyMethod_A() {...} // demands unrestricted access to
    public void MyMethod_B() {...} // the file system
}
```

Imperative syntax

Imperative syntax uses runtime method calls to create new instances of security classes.

⊞Collapse | [Copy Code](#)

```
public class MyClass
{
    public MyClass() {}

    public void Method_A()
    {
        // Do Something

        FileIOPermission myPerm =
            new FileIOPermission(PermissionState.Unrestricted);
        myPerm.Demand();
        // rest of the code won't get executed if this failed

        // Do Something
    }

    // No demands
    public void Method_B()
    {
        // Do Something
    }
}
```

The main difference between these two is, declarative calls are evaluated at compile time while imperative calls are evaluated at runtime. Please note that compile time means during JIT compilation (IL to native).

There are several actions that can be taken against permissions.

Demand	Damands
LinkDemand	
InheritanceDemand	
RequestMinimum	Requests
RequestOptional	
RequestRefuse	
Assert	Overrides
Deny	
PermitOnly	

First, let's see how we can use the declarative syntax. Take the `UIPermission` class. Declarative syntax means using attributes. So we are actually using the `UIPermissionAttribute` class. When you refer to the MSDN documentation, you can see these public properties:

- Action - one of the values in `SecurityAction` enum (common)
- Unrestricted - unrestricted access to the resource (common)
- Clipboard - type of access to the clipboard, one of the values in `UIPermissionClipboard` enum (UIPermission specific)
- Window - type of access to the window, one of the values in `UIPermissionWindow` enum (UIPermission specific).

Action and Unrestricted properties are common to all permission classes. Clipboard and Window properties are specific to `UIPermission` class. You have to provide the action that you are taking and the other properties that are specific to the permission class you are using. So in this case, you can write like the following:

⊞Collapse | [Copy Code](#)

```
[UIPermission(SecurityAction.Demand,
    Clipboard=UIPermissionClipboard.AllClipboard)]
```

or with both Clipboard and Window properties:

⊞Collapse | [Copy Code](#)

```
[UIPermission(SecurityAction.Demand,
    Clipboard=UIPermissionClipboard.AllClipboard,
    Window=UIPermissionWindow.AllWindows)]
```

If you want to declare a permission with unrestricted access, you can do it as the following:

⊞Collapse | [Copy Code](#)

```
[UIPermission(SecurityAction.Demand, Unrestricted=true)]
```

When using imperative syntax, you can use the constructor to pass the values and later call the appropriate action. We'll take the RegistryPermission class.

☒Collapse | [Copy Code](#)

```
RegistryPermission myRegPerm =  
    new RegistryPermission(RegistryPermissionAccess.AllAccess,  
        "HKEY_LOCAL_MACHINE\\Software");  
myRegPerm.Demand();
```

If you want unrestricted access to the resource, you can use PermissionState enum in the following way:

☒Collapse | [Copy Code](#)

```
RegistryPermission myRegPerm = new  
    RegistryPermission(PermissionState.Unrestricted);  
myRegPerm.Demand();
```

This is all you need to know to use any permission class in the .NET Framework. Now, we'll discuss about the actions in detail.

Security Demands

Demands are used to ensure that every caller who calls your code (directly or indirectly) has been granted the demanded permission. This is accomplished by performing a stack walk. What .. a cat walk? No, that's what your girl friend does. I mean a stack walk. When demanded for a permission, the runtime's security system walks the call stack, comparing the granted permissions of each caller to the permission being demanded. If any caller in the call stack is found without the demanded permission then a SecurityException is thrown. Please look at the following figure which is taken from the MSDN documentation.

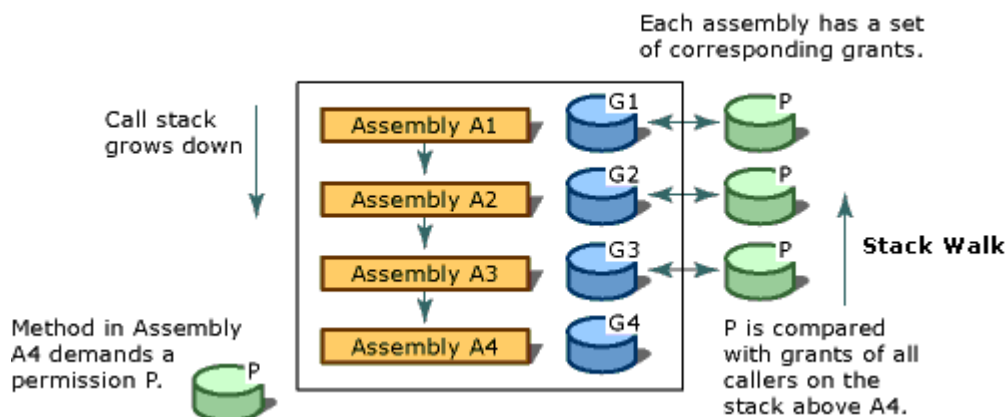


Figure 1.11

Different assemblies as well as different methods in the same assembly are checked by the stack walk.

Now back to demands. These are the three types of demands.

- Demand
- Link Demand
- Inheritance Demand

Demand

Try this sample coding. We didn't use security namespaces before, but we are going to use them now.

☒Collapse | [Copy Code](#)

```
using System.Security;
using System.Security.Permissions;
```

Add another button to the existing form.

☒Collapse | [Copy Code](#)

```
private void btnFileRead_Click(object sender, System.EventArgs e)
{
    try
    {
        InitUI(1);
    }
    catch (SecurityException err)
    {
        MessageBox.Show(err.Message,"Security Error");
    }
    catch (Exception err)
    {
        MessageBox.Show(err.Message,"Error");
    }
}
```

InitUI just calls the ShowUI function. Note that it has been denied permission to read the C: drive.

☒Collapse | [Copy Code](#)

```
// Access is denied for this function to read from C: drive
// Note: Using declarative syntax
[FileIOPermission(SecurityAction.Deny,Read="C:\\")]
private void InitUI(int uino)
{
    // Do some initializations
    ShowUI(uino); // call ShowUI
}
```

ShowUI function takes uino in and shows the appropriate UI.

☒Collapse | [Copy Code](#)

```
private void ShowUI(int uino)
```

```

{
    switch (uino)
    {
        case 1: // That's our FileRead UI
            ShowFileReadUI();
            break;
        case 2:
            // Show someother UI
            break;
    }
}

```

ShowFileReadUI shows the UI related to reading files.

⊞Collapse | [Copy Code](#)

```

private void ShowFileReadUI()
{
    MessageBox.Show("Before calling demand");
    FileIOPermission myPerm = new
        FileIOPermission(FileIOPermissionAccess.Read, "C:\\");
    myPerm.Demand();
    // All callers must have read permission to C: drive
    // Note: Using imperative syntax

    // code to show UI
    MessageBox.Show("Showing FileRead UI");
    // This is excuted if only the Demand is successful.
}

```

I know that this is a silly example, but it's enough to do the job.

Now run the code. You should get the "**Before calling demand**" message, and right after that the custom error message - "**Security Error**". What went wrong? Look at the following figure:

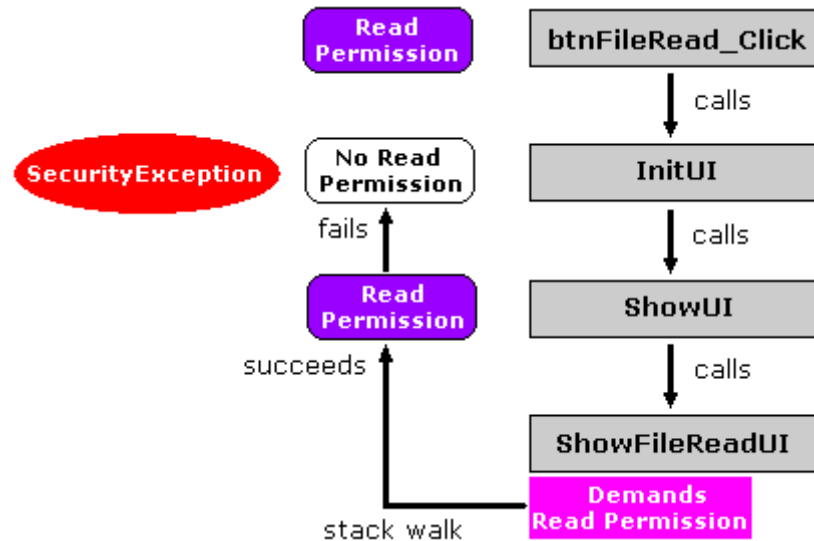


Figure 1.12

We have denied read permission for the InitUI method. So when ShowFileReadUI demands read permission to C: drive, it causes a stack walk and finds out that not every caller is granted the demanded permission and throws an exception. Just comment out the Deny statement in InitUI method, then this should be working fine because all the callers have the demanded permission.

Note that according to the documentation, most classes in .NET Framework already have demands associated with them. For example, take the StreamReader class. StreamReader automatically demands FileIOPermission. So placing another demand just before it causes an unnecessary stack walk.

Link Demand

A link demand only checks the immediate caller (direct caller) of your code. That means it doesn't perform a stack walk. Linking occurs when your code is bound to a type reference, including function pointer references and method calls. A link demand can only be applied declaratively.

⊞Collapse | [Copy Code](#)

```

[FileIOPermission(SecurityAction.LinkDemand,Read="C:\\")]
private void MyMethod()
{
    // Do Something
}
  
```

Inheritance Demand

Inheritance demands can be applied to classes or methods. If it is applied to a class, then all the classes that derive from this class must have the specified permission.

⊟Collapse | [Copy Code](#)

```
[SecurityPermission(SecurityAction.InheritanceDemand)]
private class MyClass()
{
    // what ever
}
```

If it is applied to a method, then all the classes that derive from this class must have the specified permission to override that method.

⊟Collapse | [Copy Code](#)

```
private class MyClass()
{
    public class MyClass() {}

    [SecurityPermission(SecurityAction.InheritanceDemand)]
    public virtual void MyMethod()
    {
        // Do something
    }
}
```

Like link demands, inheritance demands are also applied using declarative syntax only.

Requesting Permissions

Imagine a situation like this. You have given a nice form to the user with 20+ fields to enter and at the end, all the information would be saved to a text file. The user fills all the necessary fields and when he tries to save, he'll get this nice message saying it doesn't have the necessary permission to create a text file! Of course you can try to calm him down explaining all this happened because of a thing called stack walk .. caused by a demand .. and if you are really lucky you can even get away by blaming Microsoft (believe me ... sometimes it works!).

Wouldn't it be easier if you can request the permissions prior to loading the assembly? Yes you can. There are three ways to do that in Code Access Security.

- **RequestMinimum**
- **RequestOptional**
- **RequestRefuse**

Note that these can only be applied using declarative syntax in the assembly level, and not to methods or classes. The best thing in requesting permissions is that the administrator can view the requested permissions after the assembly has been deployed, using the *permview.exe* (Permission View Tool), so what ever the permissions needed can be granted.

RequestMinimum

You can use `RequestMinimum` to specify the permissions your code **must** have in order to run. The code will be only allowed to run if all the required permissions are granted by the security policy. In the following code fragment, a request has been made for permissions to write to a key in the registry. If this is not granted by the security policy, the assembly won't even get loaded. As mentioned above, this kind of request can only be made in the assembly level, declaratively.

⊞Collapse | [Copy Code](#)

```
using System;
using System.Windows.Forms;
using System.IO;

using System.Security;
using System.Security.Permissions;

// placed in assembly level
// using declarative syntax
[assembly:RegistryPermission(SecurityAction.RequestMinimum,
    Write="HKEY_LOCAL_MACHINE\\Software")]

namespace SecurityApp
{
    // Rest of the implementation
}
```

RequestOptional

Using `RequestOptional`, you can specify the permissions your code can use, but **not required** in order to run. If somehow your code has not been granted the optional permissions, then you must handle any exceptions that is thrown while code segments that need these optional permissions are being executed. There are certain things to keep in mind when working with `RequestOptional`.

If you use `RequestOptional` with `RequestMinimum`, no other permissions will be granted except these two, if allowed by the security policy. Even if the security policy allows additional permissions to your assembly, they won't be granted. Look at this code segment:

⊞Collapse | [Copy Code](#)

```
[assembly:FileIOPermission(SecurityAction.RequestMinimum, Read="C:\\")]
[assembly:FileIOPermission(SecurityAction.RequestOptional, Write="C:\\")]
```

The only permissions that this assembly will have are read and write permissions to the file system. What if it needs to show a UI? Then the assembly still gets loaded but an exception will be thrown when the line that shows the UI is executing, because even though the security policy allows `UIPermission`, it is not granted to this assembly.

Note that, like `RequestMinimum`, `RequestOptional` doesn't prevent the assembly from being loaded, but throws an exception at run time if the optional permission has not been granted.

RequestRefuse

You can use RequestRefuse to specify the permissions that you want to ensure will **never be granted** to your code, even if they are granted by the security policy. If your code only wants to read files, then refusing write permission would ensure that your code cannot be misused by a malicious attack or a bug to alter files.

⊞Collapse | [Copy Code](#)

```
[assembly:FileIOPermission(SecurityAction.RequestRefuse, Write="C:\\\\")]
```

Overriding Security

Sometimes you need to override certain security checks. You can do this by altering the behavior of a permission stack walk using these three methods. They are referred to as *stack walk modifiers*.

- **Assert**
- **Deny**
- **PermitOnly**

Assert

You can call the Assert method to stop the stack walk from going beyond the current stack frame. So the callers above the method that has used Assert are not checked. If you can trust the upstream callers, then using Assert would do no harm. You can use the previous example to test this. Modify the code in ShowUI method, just add the two new lines shown below:

⊞Collapse | [Copy Code](#)

```
private void ShowUI(int uino)
{
    // using imperative syntax to create a instance of FileIOPermission
    FileIOPermission myPerm = new
        FileIOPermission(FileIOPermissionAccess.Read, "C:\\\\");
    myPerm.Assert(); // don't check above stack frames.

    switch (uino)
    {
        case 1: // That's our FileRead UI
            ShowFileReadUI();
            break;
        case 2:
            // Show someother UI
            break;
    }

    CodeAccessPermission.RevertAssert(); // cancel assert
}
```

Make sure that the Deny statement is still there in InitUI method. Now run the code. It should be working fine without giving any exceptions. Look at the following figure:

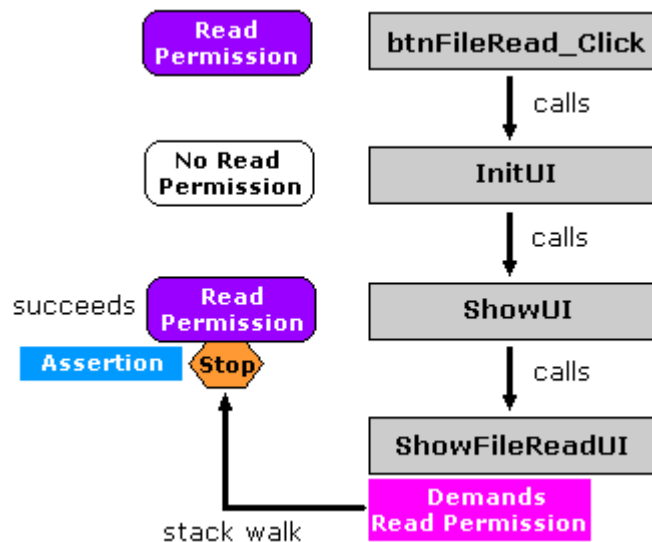


Figure 1.13

Even though InitUI doesn't have the demanded permission, it is never checked because the stack walk stops from ShowUI. Look at the last line. RevertAssert is a static method of CodeAccessPermission. It is used after an Assert to cancel the Assert statement. So if the code below RevertAssert is accessing some protected resources, then a normal stack walk would be performed and all callers would be checked. If there's no Assert for the current stack frame, then RevertAssert has no effect. It is a good practice to place the RevertAssert in a finally block, so it will always get called.

Note that to use Assert, the Assertion flag of the SecurityPermission should be set.

Warning from Microsoft!: If asserts are not handled carefully it may lead into luring attacks where malicious code can call our code through trusted code.

Deny

We have used this method already in the previous example. The following code sample shows how to deny permission to connect to a restricted website using imperative syntax:

[Collapse](#) | [Copy Code](#)

```

WebPermission myWebPermission =
    new WebPermission(NetworkAccess.Connect,
        "http://www.somewebsite.com");
myWebPermission.Deny();

// Do some work

CodeAccessPermission.RevertDeny(); // cancel Deny
  
```

RevertDeny is used to remove a previous Deny statement from the current stack frame.

PermitOnly

You can use PermitOnly in some situations when needed to restrict permissions granted by security policy. The following code fragment shows how to use it imperatively. When PermitOnly is used, it means only the resources you specify can be accessed.

⊞Collapse | [Copy Code](#)

```
WebPermission myWebPermission =  
    new WebPermission(NetworkAccess.Connect,  
        "http://www.somewebsite.com");  
myWebPermission.PermitOnly();  
  
// Do some work  
  
CodeAccessPermission.PermitOnly(); // cancel PermitOnly
```

You can use PermitOnly instead of Deny when it is more convenient to describe resources that can be accessed instead of resources that cannot be accessed.

Calculating Permissions

In the first example, we configured the machine policy level to set permissions for our code. Now we'll see how those permissions are calculated and granted by the runtime when your code belongs to more than one code group in the same policy level or in different policy levels.

The CLR computes the allowed permission set for an assembly in the following way:

1. Starting from the **All_Code** code group, all the child groups are searched to determine which groups the code belongs to, using identity information provided by the **evidence**. (If the parent group doesn't match, then that group's child groups are not checked.)
2. When all matches are identified for a particular policy level, the **permissions associated with those groups** are combined in an additive manner (**union**).
3. This is repeated for each policy level and **permissions associated with each policy level are intersected** with each other.

So all the permissions associated with matching code groups in one policy level are added together (**union**) and the result for each policy level is **intersected** with one another. An intersection is used to ensure that policy lower down in the hierarchy cannot add permissions that were not granted by a higher level.

Look at the following figure taken from a MSDN article, to get a better understanding:

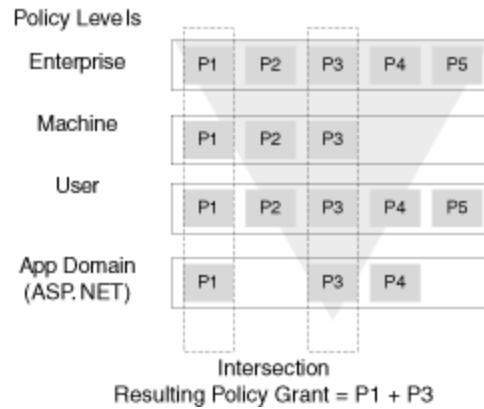


Figure 13

Have a quick look at the **All_Code** code group's associated permission set in Machine policy level. Hope it makes sense by now.

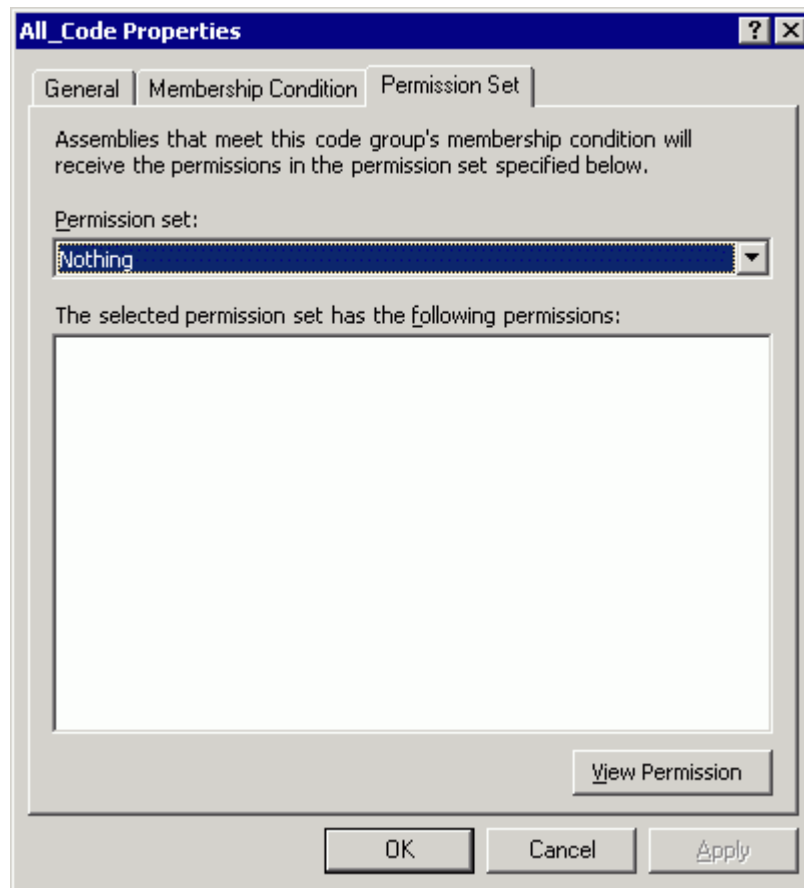


Figure 1.14

The runtime computes the allowed permission set differently if the Exclusive or LevelFinal attribute is applied to the code group. If you are not suffering from short term memory loss,

you should remember that we set the Exclusive attribute for our code group - MyCodeGroup in the earlier example.

Here's what happens if these attributes are set.

- Exclusive - The permissions with the code group marked as Exclusive are taken as the only permissions for that policy level. So permissions associated with other code groups are not considered when computing permissions.
- LevelFinal - Policy levels (except the application domain level) below the one containing this code group are not considered when checking code group membership and granting permissions.

Now you should have a clear understanding why we set the Exclusive attribute earlier.

Nice Features in .NET Configuration Tool

There are some nice features in **.NET Configuration Tool**. Just right click the **Runtime Security Policy** node and you'll see what I'm talking about.

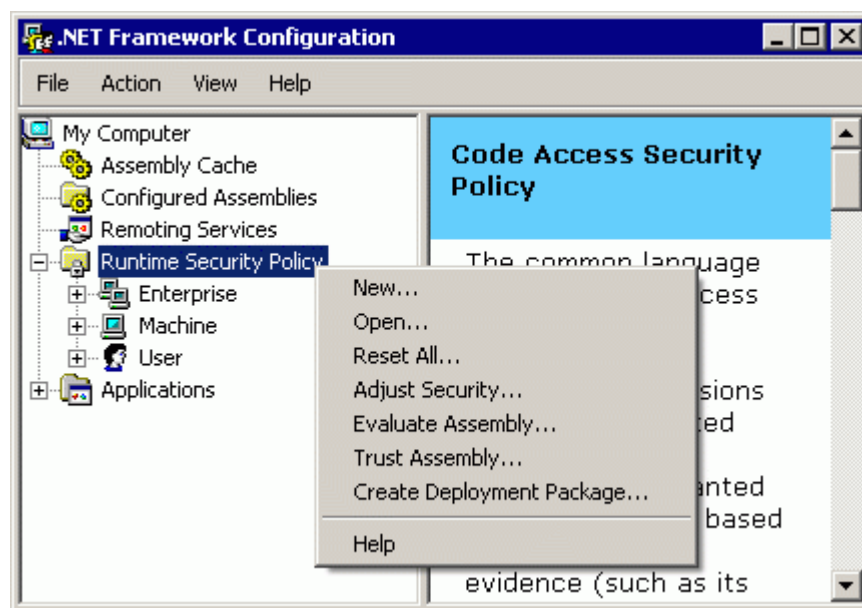


Figure 1.15

Among other options there are two important ones.

- **Evaluate Assembly** - This can be used to find out which code group(s) a particular assembly belongs to, or which permissions it has.
- **Create Deployment Package** - This wizard will create a policy deployment package. Just choose the policy level and this wizard will wrap it into a Windows Installer Package (.msi file), so what ever the code groups and permissions in your development PC can be quickly transferred to any other machine without any headache.

Tools

Permissions View Tool - permview.exe

The Permissions View tool is used to view the minimal, optional, and refused permission sets requested by an assembly. Optionally, you can use *permview.exe* to view all declarative security used by an assembly. Please refer to the MSDN documentation for additional information.

Examples:


- *permview SecurityApp.exe* - Displays the permissions requested by the assembly *SecurityApp.exe*.

Code Access Security Policy Tool - caspol.exe

The Code Access Security Policy tool enables users and administrators to modify security policy for the machine policy level, the user policy level and the enterprise policy level. Please refer to the MSDN documentation for additional information.

Examples:

Here's the output when you run "**caspol -listgroups**", this will list the code groups that belong to the default policy level - Machine level.



```
Visual Studio .NET Command Prompt
C:\Documents and Settings\Fox Mulder>caspol -listgroups
Microsoft (R) .NET Framework CasPol 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Security is ON
Execution checking is ON
Policy change prompt is ON

Level = Machine
Code Groups:
1. All code: Nothing
  1.1. Zone - MyComputer: FullTrust
    1.1.1. StrongName - 0024000004800000940000000602000000240000525341
    1.1.2. StrongName - 000000000000000040000000000000: FullTrust
  1.2. Zone - Intranet: LocalIntranet
    1.2.1. All code: Same site Web.
    1.2.2. All code: Same directory FileIO - Read, PathDiscovery
  1.3. Zone - Internet: Internet
    1.3.1. All code: Same site Web.
  1.4. Zone - Trusted: Internet
    1.4.1. All code: Same site Web.
  1.5. Zone - Untrusted: Nothing
  1.6. StrongName - 002400000480000094000000060200000024000052534131000
Success
C:\Documents and Settings\Fox Mulder>
```

Figure 1.16

Note that label "1." is for **All_Code** node because it is the parent node. Its child nodes are labeled as "1.x", and their child nodes are labeled as "1.x.x", get the picture?

- **caspol -listgroups** - Displays the code groups
- **caspol -machine -addgroup 1. -zone Internet Execution** - Adds a child code group to the root of the machine policy code group hierarchy. The new code group is a member of the **Internet** zone and is associated with the **Execution** permission set.
- **caspol -user -chgggroup 1.2. Execution** - Changes the permission set in the user policy of the code group labeled 1.2. to the **Execution** permission set.
- **caspol -security on** - Turns code access security on.
- **caspol -security off** - Turns code access security off.

Summary

- Using .NET Code Access Security, you can restrict what your code can do, restrict which code can call your code and identify code.
- There are four policy levels - Enterprise, Machine, User and Application Domain which contains code groups with associated permissions.
- Can use declarative syntax or imperative syntax.
- Demands can be used to ensure that every caller has the demanded permission.
- Requests can be used to request (or refuse) permissions in the grant time.
- Granted permissions can be overridden.

That's it. Take a look at the [Functions of Code Access Security](#) again. You should have a clear understanding by now than the first time you saw it. There are things like creating custom code access permissions, and best practices when using Code Access Security, which I haven't discussed in this article. If you want more information, you may refer to the following:

- MSDN Documentation
- [Code Access Security in Practice](#) (MSDN article)

I suggest that you go through the whole article one more time, just to make sure you didn't miss anything. If it's still not clear, don't worry, it's not your fault, it's my fault. :)

Happy Coding !!!

History

- Dec 22, 2003 - Original Version
- Jan 12, 2004 - Corrected a bug in code. (Security Demands section - ShowUI1(1) to InitUI(1) in btnFileRead_Click event) - pointed out by **Mark Focas**, Thanks a lot Mark! Updated screen shots (reduced the file size by 4 times). Added section - "Nice Features in .NET Configuration Tool". Added Figure 16 (caspol screen shot).

License

This article has no explicit license attached to it but may contain usage terms in the article text or the download files themselves. If in doubt please contact the author via the discussion board below.