For Beginners

Visual C# 4.0

A journey
towards Programming

Chirag Panchal MCP, MCPD, MCTS, MCT
(Microsoft Certified Trainer)

CMP INFOTECH

Chapter – 1 .NET FRAMEWORK

- **A .Net framework is a Software Development Environment platform.**
- **A runtime engine to merge code.**
- **A platform designed to develop Windows, Web and Mobile Applications.**
- **It is an Environment to create application such as**

  WINDOWS APPLICATION :- VB.Net, C#, J#

  WEB APPLICATION :- Asp.Net ⇨ C#, VB.Net, J#

  MOBILE APPLICATION:- Asp.Net with mobile application
- **All the above application can be made using different programming languages such as :-**

  Visual Basic.Net (i.e. Vb.Net)

  C#.Net

  XML (Extensible Markup Language)

  Asp.Net (Active server pages.net)

  J#

  Visual C# (VC++)

  Common Object Business Oriented Language (COBOL.Net)
- **In all there are nearly about 50 languages**
- **It is a product of Microsoft.**
- **.Net is exciting new computing Platform to develop applications such as CUI(Character user Interface) , GUI(Graphical user interface)**
- **Net Components and services**
  - **.Net product and services: - .Net provides different services such as**

    SERVER EXPLORER :- It shows all servers

    TOOL BOX :- It gives you readymade tools such as button, textbox, etc

    SOLUTION EXPLORER :- Different class files, Windows Forms, Web Forms.

    PROPERTIES :- Each tool from tool box is having its properties and events
  - **.Net supports third party services: - .Net also supports third party services with the help of which you can add different controls and services inside the framework such as DateTime picker in Asp.Net, AJAX control in Asp.Net 2.0 etc.**
  - **.Net versions and applications and versions :- .Net has got different versions over a period of time but the framework version is different as it is a tool to run any .Net application**

.Net versions :- 2002, 2003, 2005, 2008, 2010

Framework version :-1.0, 1.2, 2.0, 3.0, 3.5, 4.0

○ .Net
○ Different Languages
○ Class Library Files
○ Framework

- **.Net framework 3.0 has got a compact framework of size 0.8 mb as compared to .Net framework 2.0 which has got size of 22.4 mb which includes :-**

  WCF:- Windows Communication Foundation.

  WPF: - Windows Presentation Foundation.

  WCR:-Windows Card Reader.
- **WWF:-Windows Work Flow Foundation.**

  Silver light etc…
- **3.5 has new features such as-**

  LINQ: - Language Integrated Query, properties, anonymous constructors, simple properties, anonymous types, 'var' keyword, lambda expressions, extension methods etc.
- **4.0 has got features such as :-**

  PLINQ :- Parallel language integrated query

Features of DotNet

➢ Partly Platform Independent:

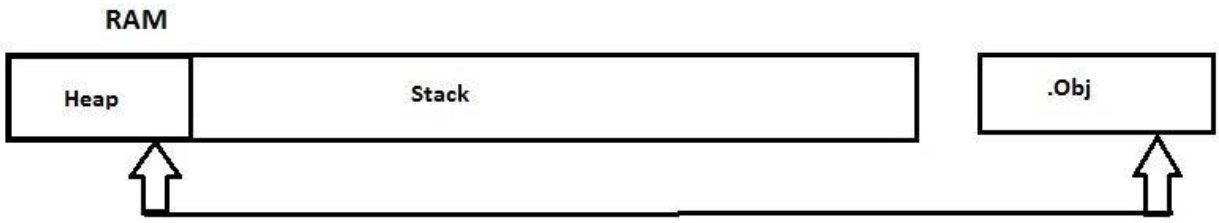
- .Net needs a platform to run and deploy applications.
- It can run and deploy both on windows.
- It can only deploy on Linux and other O.S. by installing the required framework in the O.S.

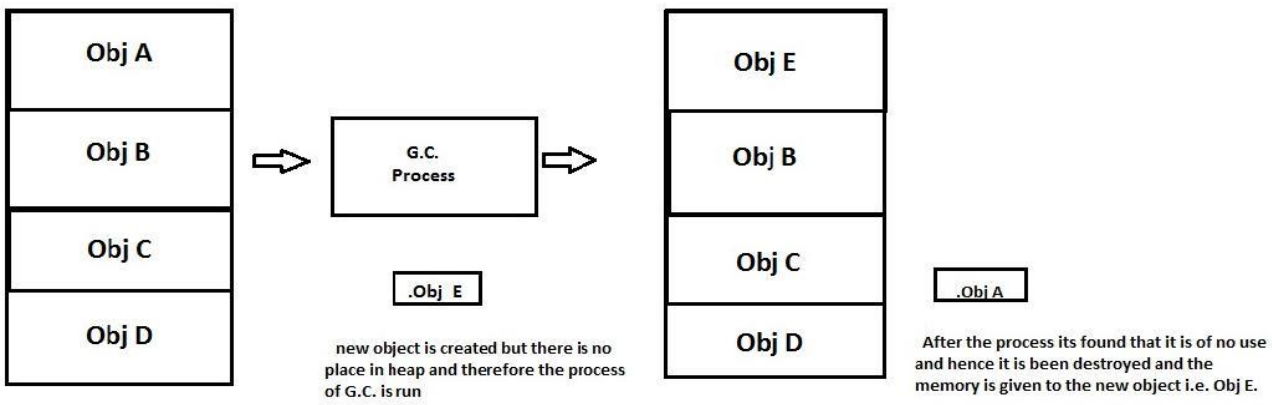
➢ Orchestration:

- The process of making the application deploy on the O.S. such as Microsoft, Linux, Mac, etc.

➢ Garbage Collection:

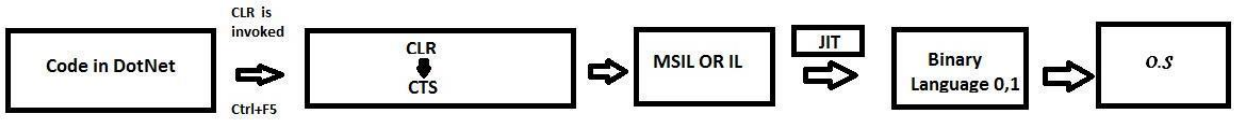
- It is a process run by .Net automatically so that the memory in ram is not keep in use for a long period.
- The Programmer can control it as well but it is preferred to keep to .Net framework to run it.
- When an object is created it is been allocated a memory in the ram in the heap area once the heap area is full or after some time. .Net framework runs the process of G.C. where it checks for the objects that are not use for a long period of time these objects are then destroy and other parts such as method, structure, and array are all stored in the stack area.



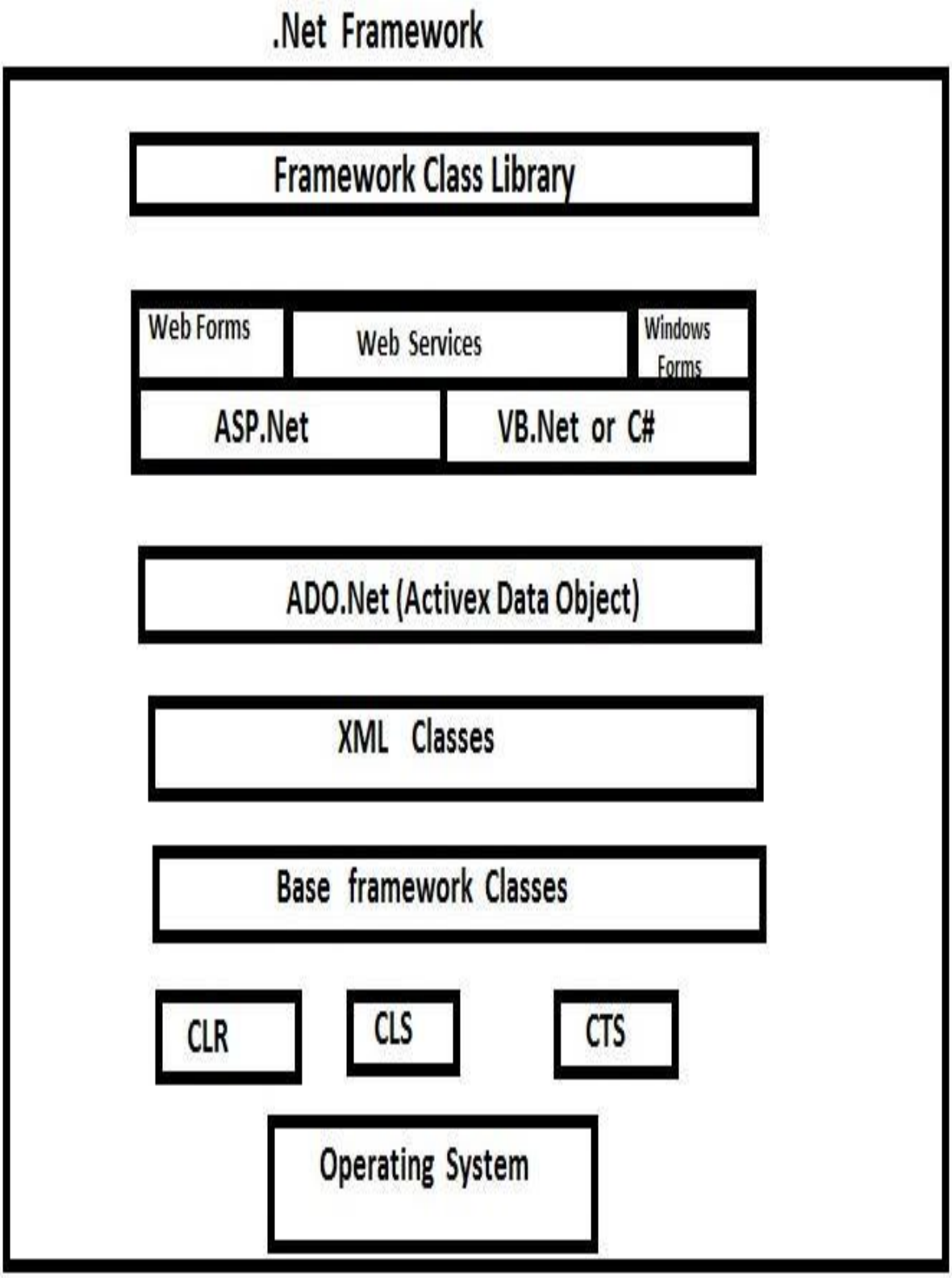


- **Language Interoperability:**
  - **The process of converting the code into MSIL, which then makes all the language of .Net understand the code this process is done by the CLR.**

- **Common Language runtime (CLR):**
  - **Code written in .Net when executed is run through a series of process i.e. CLR, CTS, and CLS.**
  - **When the code written in .Net is first executed the CLR runs when it checks the code and CTS (Common Type Specification) checks for the types according to the language choose by the programmer. The code is then converted into MSIL (Microsoft Intermediate Language) or IL (Intermediate Language).**
  - **Once the code is converted into MSIL/IL it is into a common language, which then makes it very easy for the other languages of .Net to understand.**
  - **These codes can be written once and then be kept on re-use in all the other languages. This is the work of CLS(Common Language Specification) to look into whether the code is written proper or not.**
  - **Once the code is in MSIL it is only understood by the dotnet and not by the O.S. so JIT (Just in Time compiler) comes in which converts the MSIL code into the Binary code, which is then understood by the O.S.**



- **Framework Class Library (F.C.L.):**
  - **Framework Class Library is the place where the dotnet has stored all the classes and namespaces inside it which indeed has all the methods and properties.**
- **Security:**
  - **Dotnet is a very secured application and it has got some inbuilt namespaces which we can use it such as cryptography.**
- **Scalability:**
  - **Dotnet is very scalable application which means the number of users work on dotnet application are more or less, it works in the same manner provided it has been supported with the best hardware.**


.Net Framework
Framework Class Library
Web Forms
Web Services
Windows Forms
ASP.Net
VB.Net or C#
ADO.Net (Activex Data Object)
XML Classes
Base framework Classes
CLR
CLS
CTS
Operating System

Chapter – 2 C# - OVERVIEW

C# is a modern, general-purpose object oriented programming language developed by Microsoft.

Anders Hejlsberg and his team developed C # during the development of .Net Framework.

C# is design for Common Language Infrastructure (CLI), which consists of the executable code and runtime environment that allows use of various high-level languages to be use on different computer platforms and architectures.

The following reasons make C# a widely used professional language:

- **Modern, general-purpose programming language.**
- **Object oriented.**
- **Component oriented.**
- **Easy to learn.**
- **Structured language.**
- **It produces efficient programs.**
- **It can be compiled on a variety of computer platforms.**
- **Part of .Net Framework.**

Strong Programming Features of C#

Although C# constructs closely follows traditional high-level languages C and C++ and being an object oriented programming language, it has strong resemblance with Java, it has numerous strong programming features that make it endearing to multitude of programmers worldwide.

C# - Environment

In this chapter, we will discuss the tools required for creating C# programming. We have already mentioned that C# is part of .Net framework and is used for writing .Net applications. Therefor before discussing the available tools for running a C# program, let us understand how C# relates to the .Net framework.

The .Net Framework

The .Net framework is a revolutionary platform that helps you to write the following types of applications:

- **Windows applications**
- **Web applications**
- **Web services**

The .Net framework applications are multi-platform applications. The framework has been designed in such a way that it can be used from any of the following languages: C#, C++, Visual Basic, Jscript, COBOL etc. All these languages can access the framework as well as communicate with each other.

The .Net framework consists of an enormous library of codes used by the client languages like C#. Following are some of the components of the .Net framework:

- **Common Language Runtime (CLR)**
- **The .Net Framework Class Library**
- **Common Language Specification**
- **Common Type System**
- **Metadata and Assemblies**
- **Windows Forms**
- **ASP.Net and ASP.Net AJAX**
- **ADO.Net**
- **Windows Workflow Foundation (WF)**
- **Windows Presentation Foundation**
- **Windows Communication Foundation (WCF)**
- **LINQ**

Integrated Development Environment (IDE) For C#

Microsoft provides the following development tools for C# programming:

- **Visual Studio 2010 (VS)**
- **Visual C# 2010 Express (VCE)**
- **Visual Web Developer**

The last two are freely available from Microsoft official website. Using these tools, you can write all kinds of C# programs from simple command-line applications to more complex applications. You can also write C# source code files using a basic text editor, like Notepad, and compile the code into assemblies using the command-line compiler, which is again a part of the .NET Framework.

Visual C# Express and Visual Web Developer Express edition are trim down versions of Visual Studio and has the same look and feel. They retain most features of Visual Studio. In this tutorial, we have used Visual C # 2010 Express.

Writing C# Programs on Linux or Mac OS

Although the.NET Framework runs on the Windows operating system, there are some alternative versions that work on other operating systems. Mono Framework is an open-source version of the .NET Framework, which includes a C# compiler and runs on several operating systems, including various flavors of Linux and Mac OS.

The stated purpose of Mono is not only to be able to run Microsoft .NET applications cross-platform, but also to bring better development tools to Linux developers. Mono can be run on many operating systems including Android, S

Before we study basic building blocks of the C# programming language, let us look at a bare minimum C# program structure so that we can take it as a reference in upcoming chapters.

Chapter – 3 C# - PROGRAM STRUCTURE

C# Hello World Example

A C# program consists of the following parts:

- Namespace declaration
- A class
- Class methods
- Class attributes (Variables)
- A Main method
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World":

```
Using    System;  // namespace

namespace HelloWorldApplication
{
  class HelloWorld
  {
      public static void Main(string[] args)
      {
         /* my first program in C# */
         Console.WriteLine("Hello World");
         Console.ReadLine();
       }
   }
}
```

When the above code is compile and executed, it produces following result:

```
Hello World
```

Let us look at various parts of the above program:

The first line of the program using System; - the using keyword is use to include the System namespace in the program. A program generally has multiple using statements.

The next line has the namespace declaration. A namespace is a collection of Logically Related classes. The *HelloWorldApplication* namespace contains the class *HelloWorld*.

The next line has a class declaration, the class *HelloWorld*, contains the data and method definitions that your program uses. Classes generally would contain more than one method. Methods define the behavior of the class. However, the *HelloWorld* class has only one method Main.

The next line defines the Main method, which is the entry point for all C# programs. The Main method states what the class will do when executed. Always Execution of a Program starts from Main Method.

The next line /*...*/ will be ignore by the compiler and it has been put to add additional comments in the program.

The Main method specifies its behavior with the statement Console.WriteLine("Hello World");
***WriteLine* is a method of the *Console* class defined in the *System* namespace. This statement causes the message "Hello, World!" to be displayed on the screen.**

The last line Console.ReadLine(); is for the VS.NET Users. This makes the program wait for a key press and it prevents the screen from running and closing quickly when the program is launched from Visual Studio .NET.

It's worth to note the following points:

- **C# is case sensitive.**
- **All statements and expression must end with a semicolon (;).**
- **The program execution starts at the Main method.**
- **Unlike Java, file name could be different from the class name.**

Compile & Execute a C# Program:

If you are using Visual Studio.Net for compiling and executing C# programs, take the following steps:

1. **Start Visual Studio.**
2. **On the menu bar, choose File, New, Project.**
3. **Choose Visual C# from templates, and then choose Console Application.**
4. **Specify a name for your project, Click on Browse button and specify the Path of your Project where you want to save the Application and then click on OK button.**
5. **The new project appears in Solution Explorer.**
6. **Write code in the Code Editor.**
7. **Click the Run button or the CTRL + F5 key to run the project. A Command Prompt window appears that contains the line Hello World.**
8. **You can compile a C# program by using the command line instead of the Visual Studio IDE:**
9. **Open a text editor (Notepad) and add the above mentioned code.**
10. **Save the file as helloworld.cs**

11. Open the command prompt tool Start Microsoft Visual Studio 2010 Visual Studio Tools Visual Studio Coomand Prompt and go to the directory (Use CD FOLDERNAME) where you saved the file.
12. Type csc helloworld.cs and press enter to compile your code.
13. If there are no errors in your code the command prompt will take you to the next line and would generate helloworld.exe executable file.
14. Next, type helloworld to execute your program.
15. You will be able to see "Hello World" printed on the screen.

Basic Syntax

C# is an object oriented programming language. In Object Oriented Programming methodology a program consists of various objects that interact with each other by means of actions. The actions that an object may take are called methods. Objects of the same kind are said to have the same type or, more often, are said to be in the same class.

For example, let us consider a Rectangle object. It has attributes like length and width. Depending upon the design, it may need ways for accepting the values of these attributes, calculating area and display details.

Let us look at an implementation of a Rectangle class and discuss C# basic syntax, on the basis of our observations in it:

```
using System; // namespace

namespace RectangleApplication
{
   class Rectangle
   {
      // member variables
      double length;
      double width;
      public void Acceptdetails()
      {
           length = 4.5;
           width = 3.5;
      }
      public double GetArea()
      {
          return length * width;
      }
     public void Display()
      {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
  }

  class ExecuteRectangle
```

```
  {
     public static void Main(string[] args)
     {
        Rectangle r = new Rectangle();
        r.Acceptdetails();
        r.Display();
        Console.ReadLine();
     }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
Length: 4.5
Width: 3.5
Area: 15.75
```

The *using* Keyword

The first statement in any C# program is

```
using System;
```

The using keyword is used for including the namespaces in the program. A program can include multiple using statements.

The *class* Keyword

The class keyword is used for declaring a class.

Comments in C#

Comments are used for explaining code. Compilers ignore the comment entries. The multiline comments in C# programs start with /* and terminates with the characters */ as shown below:

```
/* This program demonstrates
The basic syntax of C# programming  Language */
```

Single line comments are indicated by the '//' symbol. For example,

```
//end class Rectangle
```

Member Variables

Variables are attributes or data members of a class, used for storing data. In the preceding program, the *Rectangle* class has two member variables named *length* and *width*.

Member Functions

Functions are set of statements that perform a specific task. The member functions of a class are declared within the class. Our sample class Rectangle contains three member functions: *AcceptDetails*, *GetArea* and *Display*.

Instantiating a Class

In the preceding program, the class *ExecuteRectangle* is used as a class which contains the *Main()* method and instantiates the *Rectangle* class.

Identifiers

An identifier is a name used to identify a class, variable, function, or any other user-defined item. The basic rules for naming classes in C# are as follows:

- A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9) or underscore.
- The first character in an identifier cannot be a digit.
- It must not contain any embedded space or symbol like ? - +! @ # % ^ & * ( ) [ ] { } . ; : " ' / and \.
- However an underscore (_) can be used.
- It should not be a C# keyword.

C# Keywords

Keywords are reserved words predefined to the C# compiler. These keywords cannot be used as identifiers, however, if you want to use these keywords as identifiers, you may prefix the keyword with the @ character.

In C# some identifiers have special meaning in context of code, such as get and set, these are called contextual keywords.

The following table lists the reserved keywords and contextual keywords in C#:

Reserved Keywords (* = generic modifier)						
abstract	As	base	bool	break	byte	case
catch	Char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	Goto	if	implicit	in	in *	int
interface	internal	is	lock	long	namespace	new
null	object	operator	out	out *	override	params
private	protected	public	readonly	ref	return	sbyte
sealed	Short	sizeof	stackalloc	static	string	struct
switch	This	throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using	virtual	void
volatile	While					
Contextual Keywords						
Add	Alias	ascending	descending	dynamic	from	get
Global	Group	into	join	let	orderby	partial (type)
partial (method)	remove	select	set			

Chapter – 4 C# - DATA TYPES

In C#, variables are categorized into the following types:

- **Value types**
- **Reference types**

Value Types

Value type variables can be assigned a value directly. They are derived from the class System.ValueType. The value types directly contain data. Some examples are int, char, float, which stores numbers, alphabets and floating-point numbers respectively. When you declare an int type, the system allocates memory to store the value.

The following table lists the available value types in C# 2010:

Type	Represents	Ranges	Default Value
bool	Boolean value	True or False	False
byte	8-bit unsigned integer	0 to 255	0
char	16-bit Unicode character	U +0000 to U +ffff	'\0'
decimal	128-bit precise decimal values with 28-29 significant digits	$(-7.9 \times 10^{28}$ to $7.9 \times 10^{28}) / 10^{0 \text{ to } 28}$	0.0M
double	64-bit double-precision floating point type	$(+/-)5.0 \times 10^{-324}$ to $(+/-)1.7 \times 10^{308}$	0.0D
float	32-bit single-precision floating point type	-3.4×10^{38} to $+ 3.4 \times 10^{38}$	0.0F
int	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0
long	64-bit signed integer type	-923,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
sbyte	8-bit signed integer type	-128 to 127	0
short	16-bit signed integer type	-32,768 to 32,767	0
uint	32-bit signed integer type	0 to 4,294,967,295	0
ulong	64-bit signed integer type	0 to 18,446,744,073,709,551,615	0
ushort	16-bit signed integer type	0 to 65,535	0

To get the exact size of a type or a variable on a particular platform, you can use the sizeof method. The expression *sizeof(type)* yields the storage size of the object or type in bytes. Following is an example to get the size of *int* type on any machine:

```
using System;

namespace DataTypeApplication
{
  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Size of int: {0}", sizeof(int));
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
Size of  int: 4
```

Reference Types

The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables.
In other words, they refer to a memory location. Using more than one variable, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value. Example of built in reference types are: object, dynamic and string. User defined reference type are class, interface, or delegate. We will discuss these types in later chapter.

Object Type

The Object Type is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for System.Object class. So object types can be assigned values of any other types, value types, reference types, predefined or user-defined types. However, before assigning values, it needs type conversion.

When a value type is converted to object type, it is called boxing and on the other hand, when an object type is converted to a value type it is called unboxing.

```
object obj; obj = 100; // this  is  boxing
```

Dynamic Type

You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at runtime.

Syntax for declaring a dynamic type is:

dynamic <variable_name>	= value;
For example, dynamic d	= 20;
String Type	Dynamic types are similar to object types except that, type checking for object type variables takes place at compile time, whereas that for the dynamic type variables take place at run time. The String Type allows you to assign any string values to a variable. The string type is an alias for the System.String quoted and @quoted.
For example,	class. It is derived from object type. The value for a string type can be assigned using string literals in two forms:
String str	= "CMP INFOTECH"; A @quoted string literal looks like: @"CMP INFOTECH";

C# - Type Conversion

Type conversion is basically type casting, or converting one type of data to another type. In C#, type casting has two forms:

Implicit type conversion - these conversions are performed by C# in a type-safe manner. Examples are conversions from smaller to larger integral types, and conversions from derived classes to base classes.

Explicit type conversion - these conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.

The following example shows an explicit type conversion:

```
namespace TypeConversionApplication
{
   class ExplicitConversion
   {
      public static void Main(string[] args)
      {
         double d = 5673.74;
         int i;
         // cast double to int.
         i = (int)d;
         Console.WriteLine(i);
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
5673
```

C# Type Conversion Methods

C# provides the following built-in type conversion methods:

Sr No.	MethodsName	Description
1	ToBoolean	Converts a type to a Boolean value, where possible.
2	ToByte	Converts a type to a byte.
3	ToChar	Converts a type to a single Unicode character, where possible.
4	ToDateTime	Converts a type (integer or string type) to date-time structures.
5	ToDecimal	Converts a floating point or integer type to a decimal type.
6	ToDouble	Converts a type to a double type.
7	ToInt16	Converts a type to a 16-bit integer.
8	ToInt32	Converts a type to a 32-bit integer.
9	ToInt64	Converts a type to a 64-bit integer.
10	ToSbyte	Converts a type to a signed byte type.
11	ToSingle	Converts a type to a small floating-point number.
12	ToString	Converts a type to a string.
13	ToType	Converts a type to a specified type.
14	ToUInt16	Converts a type to an unsigned int type.
15	ToUInt32	Converts a type to an unsigned long type.
16	ToUInt64	Converts a type to an unsigned big integer.

The following example converts various value types to string type:

```
namespace TypeConversionApplication
{
   class StringConversion
   {
      public static void Main(string[] args)
      {
          int i = 75;
          float f = 53.005f;
          double d = 2345.7652;
          bool b = true;
        Console.WriteLine(i.ToString());
        Console.WriteLine(f.ToString());
        Console.WriteLine(d.ToString());
        Console.WriteLine(b.ToString());
        Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
75
53.005
2345.7652
True
```

Chapter – 5 C# - VARIABLE

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C# has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

We have already discussed various data types. The basic value types provided in C# can be categorized as:

Type	Example
Integral types	sbyte, byte, short, ushort, int, uint, long, ulong and char
Floating point types	float and double
Decimal types	decimal
Boolean types	true or false values, as assigned
Nullable types	Nullable data types

C# also allows defining other value types of variable like enum and reference types of variables like class, which we will cover in subsequent chapters. For this chapter, let us study only basic variable types.

Variable Declaration in C#

Syntax for variable declaration in C# is:

```
<data_type> <variable_list>;
```

Here, data_type must be a valid C# data type including char, int, float, double, or any user defined data type etc., and variable_list may consist of one or more identifier names separated by commas.

Some valid variable declarations along with their definition are shown here:

```
int i, j, k;
char c, ch;
float f,,salary;
double d;
```

You can initialize a variable at the time of declaration as:

```
int i = 100;
```

Variable Initialization in C#

Variables are initialized (assigned a value) with an equal sign followed by a constant expression. The general form of initialization is:

```
variable_name = value;
```

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as:

```
<data_type> <variable_name> = value;
```

Some examples are:

```
int d = 3, f = 5;    /* initializing d and f. */
byte z = 22;         /* initializes z. */
double pi = 3.14159; /* declares an approximation of pi. */
char x = 'x';        /* the variable x has the value 'x'. */
```

It is a good programming practice to initialize variables properly otherwise, sometime program would produce unexpected result.

Try following example, which makes use of various types of variables:

```
namespace VariableDeclaration
{
   class  Program
   {
      public static void Main(string[] args)
      {
         short a;
         int b;
         double c;
         /* actual initialization */
         a = 10;
         b = 20;
         c = a + b;
         Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
a = 10, b = 20, c = 30
```

Accepting Values from User

The Console class in the System namespace provides a function ReadLine() for accepting input from the user and store it into a variable.

For example,

```
int  num;
num  = Convert.ToInt32(Console.ReadLIne());
```

The function Convert.ToInt32() converts the data entered by the user to int data type, because Console.ReadLine() accepts the data in string format.

Constants And Literals

The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals. Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are also enumeration constants as well.

The constants are treated just like regular variables except that their values cannot be modified after their definition.

Character Constants

Character literals are enclosed in single quotes e.g., 'x' and can be stored in a simple variable of char type. A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t').

There are certain characters in C# when they are proceeded by a back slash they will have special meaning and they are used to represent like newline (\n) or tab (\t). Here you have a list of some of such escape sequence codes:

Escape sequence	Meaning
\\	\ character
\'	' character
\"	" character
\?	? character
\b	Backspace
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

Following is the example to show few escape sequence characters:

```
namespace EscapeChar
{
   class  Program
   {
      public static void Main(string[] args)
      {
         Console.WriteLine("Hello\tWorld\n\n");
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Hello    World
```

String Literals

String literals or constants are enclosed in double quotes "" or with @"". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating the parts using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear" "hello,
\ dear"
"hello, " "d" "ear"
@"hello dear"
```

Defining Constants

Constants are defined using the const keyword. Syntax for defining a constant is:

```
const <data_type> <constant_name> = value;
```

The following program demonstrates defining and using a constant in your program:

```
using System;

namespace DeclaringConstants
{
   class Program
   {
      public static void Main(string[] args)
      {
         const double pi = 3.14159; // constant declaration
         double r;
         Console.WriteLine("Enter Radius: ");
         r = Convert.ToDouble(Console.ReadLine());
         double areaCircle = pi * r * r;
         Console.WriteLine("Radius: {0}, Area: {1}", r, areaCircle);
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Enter Radius: 3
Radius: 3,
Area: 28.27431
```

Chapter – 6 C# - OPERATORS

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C# is rich in built-in operators and provides the following type of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators

Arithmetic Operators

Following table shows all the arithmetic operators supported by C#. Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiply both operands	A * B will give 200
/	Divide numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator increases integer value by one	A++ will give 11
--	Decrement operator decreases integer value by one	A-- will give 9

Relational Operators

Following table shows all the relational operators supported by C#. Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
=	Checks if the value of two operands is equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.

<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

Following table shows all the logical operators supported by C#. Assume variable A holds Boolean value true and variable B holds Boolean value false then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Arithmetic Assignment Operators

There are following assignment operators supported by C#:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A

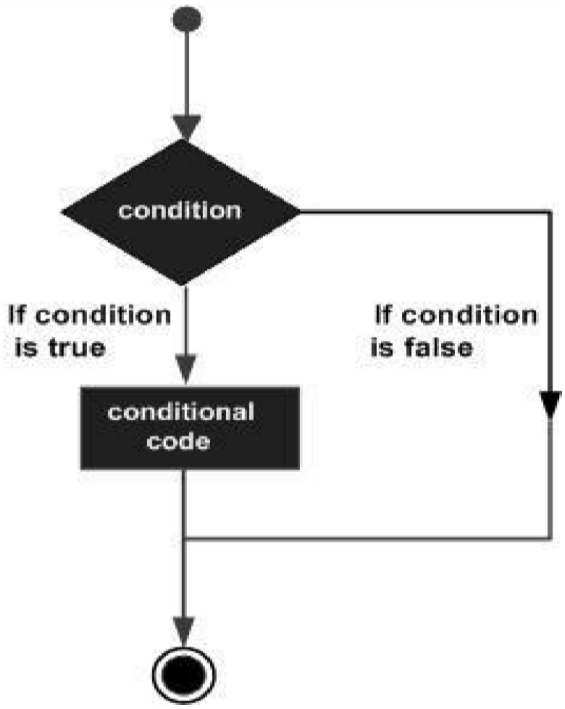
Operators Precedence in C#

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example x = 7 + 3 * 2; Here x is assigned 13, not 20 because operator * has higher precedence than + so it first get multiplied with 3*2 and then adds into 7. s

Chapter – 7 C# - DECISION MAKING

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.



Following is the general from of a typical decision making structure found in most of the programming languages:

C# provides following types of decision-making statements. Click the following links to check their detail.

Statement	Description
if statement	An if statement consists of a Boolean expression followed by one or more statements.
if...else statement	An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.
nested if statements	You can use one if or else if statement inside another if or else if statement(s).
switch statement	A switch statement allows a variable to be tested for equality against a list of values.
nested switch statements	You can use one switch statement inside another switch statement(s).

The ? : Operator :

We have covered conditional operator ? : in previous chapter which can be used to replace if...else statements. It has the following general form:

```
Exp1 ? Exp2 : Exp3;
```

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined like this: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

If Statement

An if statement consists of a boolean expression followed by one or more statements.

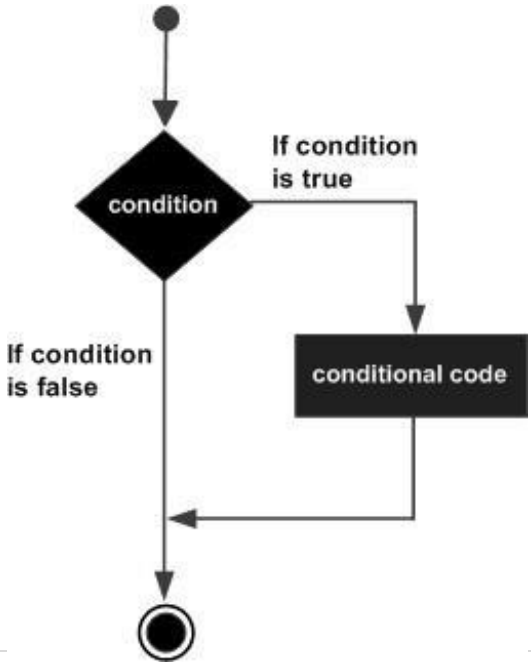
Syntax:

The syntax of an if statement in C# is:

```
if(boolean_expression)
{
   /* statement(s) will execute if the boolean expression is true */
}
```

If the boolean expression evaluates to true then the block of code inside the if statement will be executed. If boolean expression evaluates to false then the first set of code after the end of the if statement(after the closing curly brace) will be executed.

Flow Diagram:



Example:

```
using System;

namespace DecisionMaking
{
   class Program
   {
      public static void Main(string[] args)
      {
         /* local variable definition */
         int a = 10;
         /* check the boolean condition using if statement */
         if (a < 20)
         {
            /* if condition is true then print the following */
             Console.WriteLine("a is less than 20");
         }
         Console.WriteLine("value of a is : {0}", a);
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
a is less than 20;
value of a is : 10
```

Nested If Statements

It is always legal in C# to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Syntax:

The syntax for a nested if statement is as follows:

```
if( boolean_expression 1)
{
   /* Executes when the boolean expression 1 is true */
   if(boolean_expression 2)
   {
      /* Executes when the boolean expression 2 is true */
   }
}
```

You can nest else if...else in the similar way as you have nested *if* statement.

Example:

```
using System;

namespace DecisionMaking
{
   class Program
   {
      public static void Main(string[] args)
      {
         /* local variable definition */
         int a = 100;
         int b = 200;
         /* check the boolean condition */
         if (a == 100)
         {
            /* if condition is true then check the following */
            if (b == 200)
            {
                /* if condition is true then print the following */
                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();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

IF...ELSE STATEMENT

An if statement can be followed by an optional else statement, which executes when the boolean expression is false.

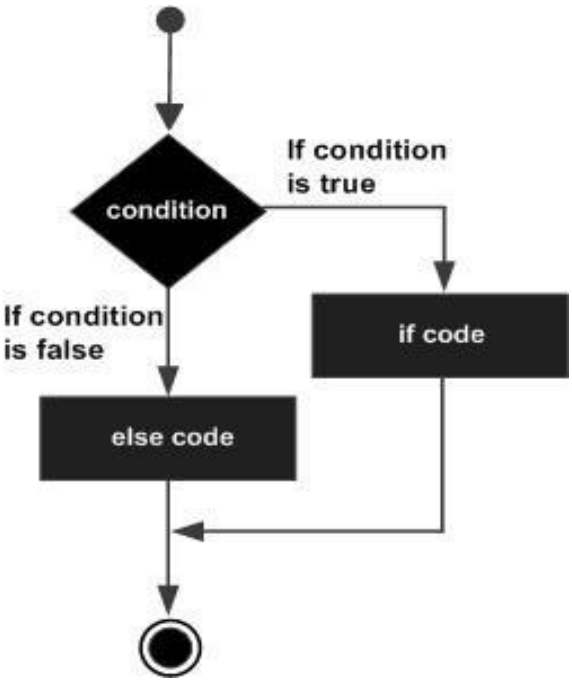
Syntax:

The syntax of an if...else statement in C# is:

```
If (boolean_expression)
{
   /* statement(s) will execute if the boolean expression is true */
}
else
{
  /* statement(s) will execute if the boolean expression is false */
}
```

If the boolean expression evaluates to true then the if block of code will be executed otherwise else block of code will be executed.

Flow Diagram:



Example:

```
using System;

namespace DecisionMaking
{
   class Program
   {
      public static void Main(string[] args)
      {
         /* local variable definition */
         int a = 100;
         /* check the boolean condition */
          if (a < 20)
          {
              /* if condition is true then print the following */
              Console.WriteLine("a is less than 20");
          }
          else
          {
              /* if condition is false then print the following */
              Console.WriteLine("a is not less than 20");
          }
          Console.WriteLine("value of a is : {0}", a);
          Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
a is not less than 20; value of a is : 100
```

The if...else if...else Statement

- An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.
- When using if, else if, else statements there are few points to keep in mind.
- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax:

The syntax of an if...else if...else statement in C# is:

```
if(boolean_expression 1)
{
   /* Executes when the boolean expression 1 is true */
}
else if( boolean_expression 2)
{
   /* Executes when the boolean expression 2 is true */
}
else if( boolean_expression 3)
{
   /* Executes when the boolean expression 3 is true */
}
else
{
   /* executes when the none of the above condition is true */
}
```

Example:

```
using System;

namespace DecisionMaking
{
    class Program
    {
        public static void Main(string[] args)
        {
            /* local variable definition */
            int a = 100;
            /* check the boolean condition */
             if (a == 10)
            {
                /* if condition is true then print the following */
                Console.WriteLine("Value of a is 10");
            }
```

```
         else if (a == 20)
         {
            /* if else if condition is true */
            Console.WriteLine("Value of a is 20");
         }
         else if (a == 30)
         {
            /* if else if condition is true  */
            Console.WriteLine("Value of a is 30");
         }
        else
         {
            /* if none of the conditions is true */
            Console.WriteLine("None of the values is matching");
         }
         Console.WriteLine("Exact value of a is: {0}", a);
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
None of the values is matching
Exact value of a is: 100
```

SWITCH STATEMENT

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

Syntax:

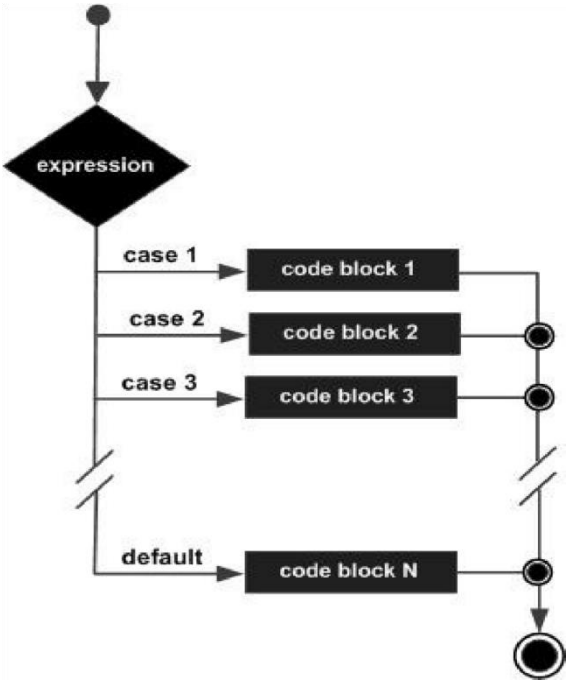
The syntax for a switch statement in C# is as follows:

```
switch(expression)
{
   case constant-expression  :
      statement(s);
      break; /* optional*/
   case constant-expression  :
      statement(s);
       break; /* optional */
      /* you can have any number of case statements */
   default : /* Optional */
      statement(s);
}
```

The following rules apply to a switch statement:

1. **The expression used in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.**
2. **You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.**
3. **The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.**
4. **When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.**
5. **When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.**
6. **Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.**
7. **A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.**

Flow Diagram:



Example:

```
using System;

namespace DecisionMaking
{
   class Program
   {
      public static void Main(string[] args)
      {
         /* local variable definition */
         char grade = 'B';
         switch (grade)
         {
             case 'A':
                Console.WriteLine("Excellent!");
                 break;
             case 'B':
             case 'C':
                Console.WriteLine("Well done");
                 break;
             case 'D':
                Console.WriteLine("You passed");
                 break;
             case 'F':
                Console.WriteLine("Better try again");
                 break;
             default:
                Console.WriteLine("Invalid grade");
                 break;
         }
         Console.WriteLine("Your grade is  {0}", grade);
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Well done
Your grade is B
```

NESTED SWITCH STATEMENTS

It is possible to have a switch as part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

Syntax:

The syntax for a nested switch statement is as follows:

```
switch(ch1)
{    case 'A':
     printf("This A is part of outer switch" );
     switch(ch2)
     {
       case 'A': printf("This A is part of inner switch" );
       break;
       case 'B': /* inner B case code */
     }
     break;
     case 'B': /* outer B case code */
}
```

Example:

```
using System;
namespace DecisionMaking
{
   class Program
   {
      public  static void Main(string[] args)
      {
          int a = 100;
          int b = 200;
         switch (a)
         {       case 100:
                 Console.WriteLine("This is part of outer switch ");
                 switch (b)
                 {
                     case 200:
                     Console.WriteLine("This is part of inner switch ");
                     break;
                 }
                 break;
         }
         Console.WriteLine("Exact value of a is : {0}", a);
         Console.WriteLine("Exact value of b is : {0}", b);
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
This is part of outer switch
This is part of inner switch
Exact value of a is : 100
Exact value of b is : 200
```

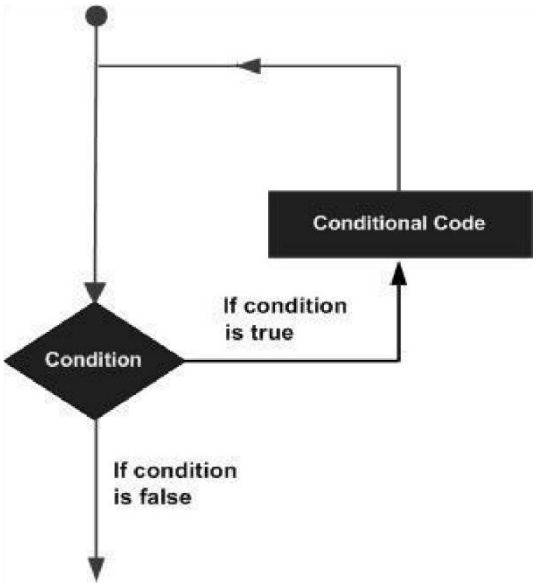
Chapter – 8 C# - LOOPS

There may be a situation when you need to execute a block of code several number of times. In general statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:

C# provides following types of loop to handle looping requirements. Click the following links to check their detail.



Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body
nested loops	You can use one or more loop inside any another while, for or do..while loop.

A **while** loop statement in C# repeatedly executes a target statement as long as a given condition is true.

Syntax:

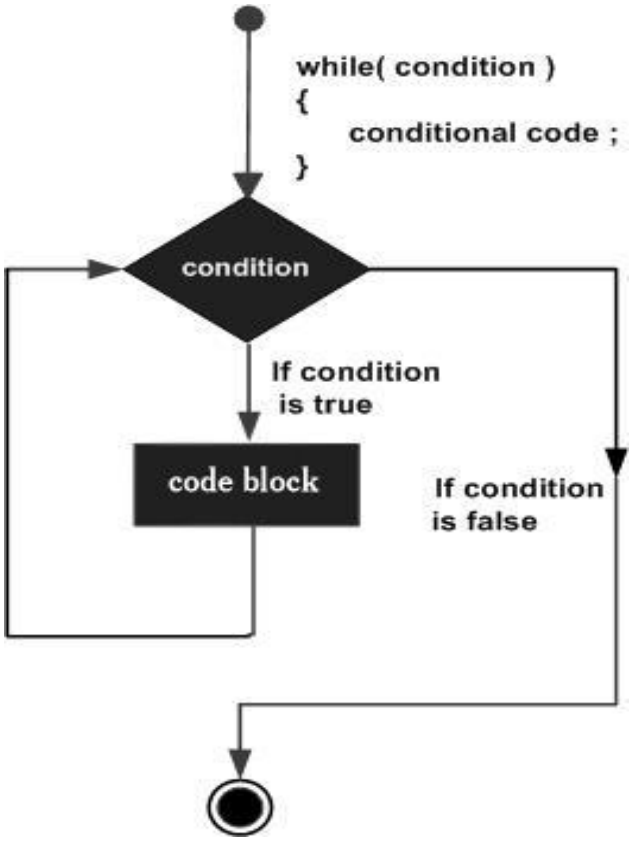
The syntax of a while loop in C# is:

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

Here statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any nonzero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram:



Here key point of the *while* loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
using System;

namespace Loops
{
   class Program
   {
      public static void Main(string[] args)
      {
         /* local variable definition */
         int a = 10;
         /* while loop execution */
         while (a < 20)
         {
            Console.WriteLine("value of a: {0}", a);
            a++;
         }
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

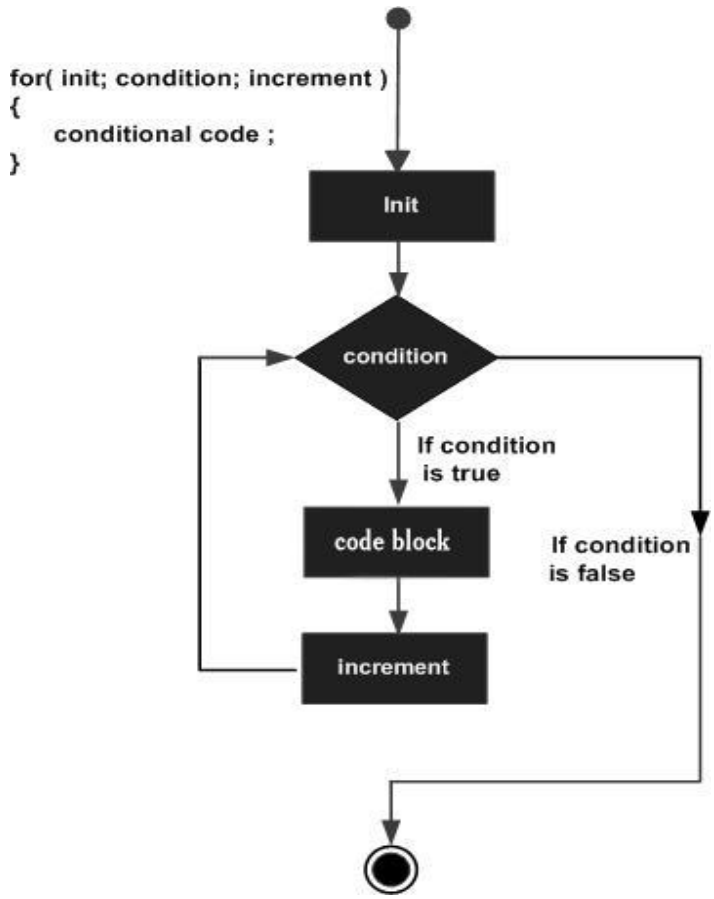
The syntax of a for loop in C# is:

```
for ( init; condition; increment )
{
   statement(s);
}
```

Here is the flow of control in a for loop:

1. The init step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
2. Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
3. After the body of the for loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram:



Example:

```
using System;

namespace Loops
{
   class Program
   {
      public static void Main(string[] args)
      {
         /* for loop execution */
         for (int a = 10; a < 20; a = a + 1)
         {
            Console.WriteLine("value of a: {0}", a);
         }
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Unlike for and while loops, which test the loop condition at the top of the loop, the do...while loop checks its condition at the bottom of the loop.

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

The syntax of a do...while loop in C# is:

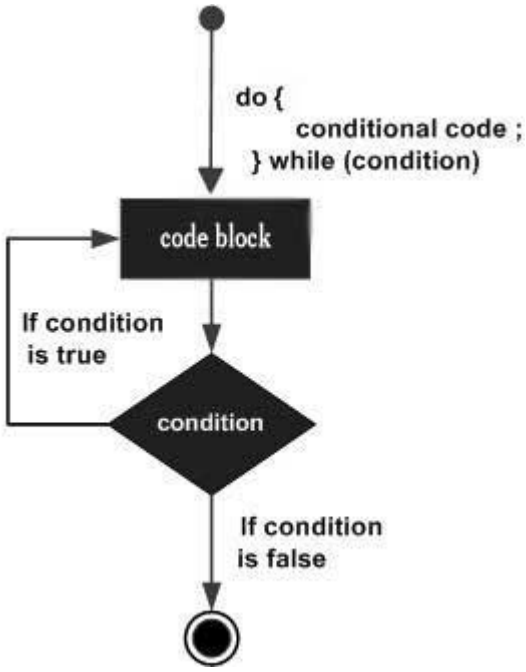
```
do
{
  statement(s);

}while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

Flow Diagram:



Example:

```
using System;

namespace Loops
{
   class Program
   {
      public static void Main(string[] args)
      {
         /* local variable definition */
         int a = 10;
         /* do loop execution */
         do
         {
           Console.WriteLine("value of a: {0}", a);
            a = a + 1;
         } while (a < 20);
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
value of a: 10
value of a: 11
```

```
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

C# allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax:

The syntax for a nested for loop statement in C# is as follows:

```
for ( init; condition; increment )
{
   for ( init; condition; increment )
   {
      statement(s);
   }
   statement(s);
}
```

The syntax for a nested while loop statement in C# is as follows:

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

The syntax for a nested do...while loop statement in C# is as follows:

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

}while( condition );
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

Example:

The following program uses a nested for loop to find the prime numbers from 2 to 100:

```
using System;

namespace Loops
{
  class Program
  {
    public static void Main(string[] args)
    {
      /* local variable definition */
      int i, j;
      for (i = 2; i < 100; i++)
      {
        for (j = 2; j <= (i / j); j++)
          if ((i % j) == 0) break; // if factor found, not prime
        if (j > (i / j))
          Console.WriteLine("{0} is prime", i);
      }
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C# provides the following control statements. Click the following links to check their detail.

Control Statement	Description
break statement	Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

The break statement in C# has following two usage:

1. When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
2. It can be used to terminate a case in the switch statement.

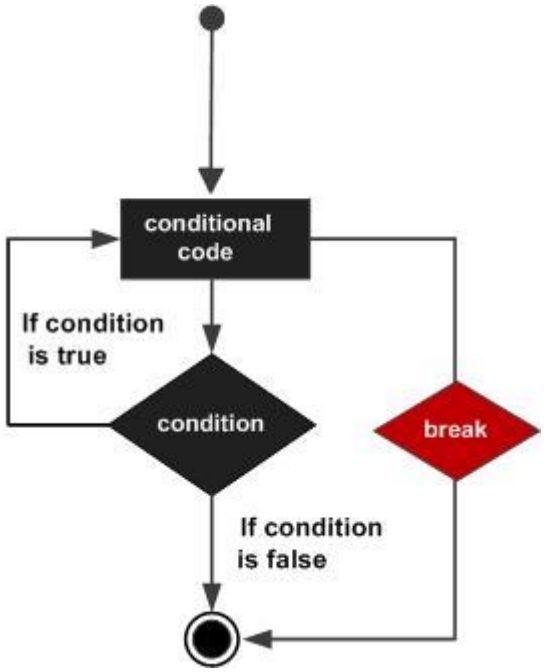
If you are using nested loops (i.e. one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax for a break statement in C# is as follows:

```
break;
```

Flow Diagram:



Example:

```
using System;

namespace Loops
{
   class Program
   {
      public static void Main(string[] args)
      {
         /* local variable definition */
         int a = 10;

         /* while loop execution */
         while (a < 20)
```

```
        {
            Console.WriteLine("value of a: {0}", a);
            a++;
            if (a > 15)
            {
                /* terminate the loop using break statement */
                break;
            }
        }
        Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

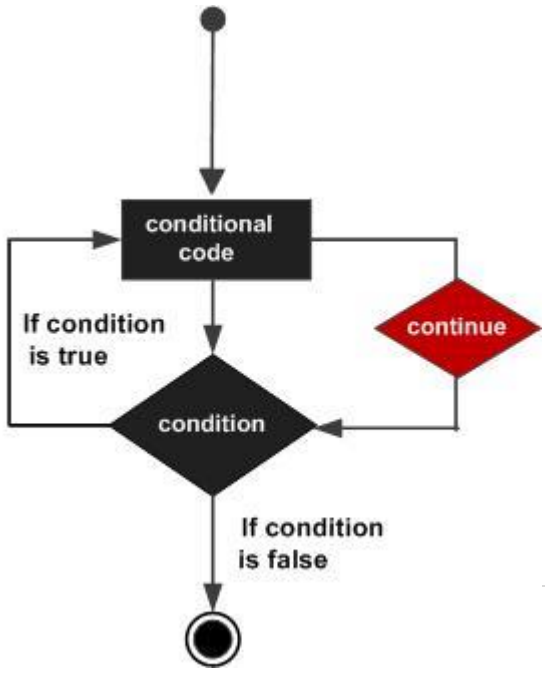
The continue statement in C# works somewhat like the break statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.
For the for loop, continue statement causes the conditional test and increment portions of the loop to execute. For the while and do...while loops, continue statement causes the program control passes to the conditional tests.

Syntax:

The syntax for a continue statement in C# is as follows:

```
continue;
```

Flow Diagram:



Example:

```
using System;

namespace Loops
{

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

```
    {
        /* local variable definition */
        int a = 10;

        /* do loop execution */
        do
        {
            if (a == 15)
            {
                /* skip the iteration */
                a = a + 1;
                continue;
            }
            Console.WriteLine("value of a: {0}", a);
            a++;

        } while (a < 20);

        Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

The Infinite Loop:

A loop becomes infinite loop if a condition never becomes false. The for loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

```
using System;

namespace Loops
{
  class Program
  {
    public static void Main(string[] args)
    {
      for ( ; ; )
      {
        Console.WriteLine("Hey! I am Trapped");
      }
```

```
    }
  }
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but programmers more commonly use the for(;;) construct to signify an infinite loop.

Chapter – 9 C# - ABSTRACTION & ENCAPSULATION

Encapsulation is defined 'as the process of enclosing one or more items within a physical or logical package'.

Encapsulation, in object oriented programming methodology, prevents access to implementation details.

Abstraction and encapsulation are related features in object oriented programming. Abstraction allows making relevant information visible and encapsulation enables a programmer to *implement the desired level of abstraction*.

Encapsulation is implemented by using access specifiers. An access specifier defines the scope and visibility of a class member. C# supports the following access specifiers:

- Public
- Private
- Protected
- Internal
- Protected internal

Public Access Specifier

Public access specifier allows a class to expose its member variables and member functions to other functions and objects. Any public member can be accessed from outside the class.

The following example illustrates this:

```
using System;

namespace RectangleApplication
{
   class Rectangle
   {
      //member variables
      public double length;
      public double width;

      public double GetArea()
      {
         return length * width;
      }

      public void Display()
      {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
```

```
   }//end class Rectangle


 class ExecuteRectangle
   {
      public static void Main(string[] args)
      {
         Rectangle r = new Rectangle();
         r.length = 4.5;
         r.width = 3.5;
         r.Display();
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Length:  4.5  Width:
3.5
Area: 15.75
```

In the preceding example, the member variables length and width are declared public, so they can be accessed from the function Main() using an instance of the Rectangle class, named r.
The member function *Display()* and *GetArea()* can also access these variables directly without using any instance of the class.

The member functions *Display()* is also declared public, so it can also be accessed from *Main()* using an instance of the Rectangle class, named r.

Private Access Specifier

Private access specifier allows a class to hide its member variables and member functions from other functions and objects. Only functions of the same class can access its private members. Even an instance of a class cannot access its private members.

The following example illustrates this:

```
using System;

namespace RectangleApplication
{
   class Rectangle
   {
      //member variables
      private double length;
      private double width;

      public void Acceptdetails()
      {
         Console.WriteLine("Enter Length: ");
         length = Convert.ToDouble(Console.ReadLine());\
         Console.WriteLine("Enter Width: ");
```

```
            width = Convert.ToDouble(Console.ReadLine());
        }
        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }//end class Rectangle

    class ExecuteRectangle
    {
        public static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.Acceptdetails();
            r.Display();
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces following result:

```
Enter Length: 4.4
Enter Width: 3.3
Length: 4.4
Width: 3.3
Area: 14.52
```

In the preceding example, the member variables length and width are declared private, so they cannot be accessed from the function Main(). The member functions *AcceptDetails()* and *Display()* can access these variables. Since the member functions *AcceptDetails()* and *Display()* are declared public, they can be accessed from *Main()* using an instance of the Rectangle class, named r.

Protected Access Specifier

Protected access specifier allows a child class to access the member variables and member functions of its base class. This way it helps in implementing inheritance. We will discuss this in more details in the inheritance chapter.

Internal Access Specifier

Internal access specifier allows a class to expose its member variables and member functions to other functions and objects in the current assembly. In other words, any member with internal access specifier can be accessed from any class or method defined within the application in which the member is defined.

The following program illustrates this:

```
using System;
namespace RectangleApplication
{
   class Rectangle
   {
      //member variables
      internal double length;
      internal double width;

      double GetArea()
      {
          return length * width;
      }

     public void Display()
      {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
   }//end class Rectangle
   class ExecuteRectangle
   {
      public static void Main(string[] args)
      {
         Rectangle r = new Rectangle();
         r.length = 4.5;
         r.width = 3.5;
         r.Display();
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Length: 4.5
Width: 3.5
Area: 15.75
```

In the preceding example, notice that the member function *GetArea()* is not declared with any access specifier. Then what would be the default access specifier of a class member if we don't mention any? It is private.

Protected Internal Access Specifier

The protected internal access specifier allows a class to hide its member variables and member functions from other class objects and functions, except a child class within the same application. This is also used while implementing inheritance.

Chapter – 10 C# - METHODS & FUNCTIONS

A method is a group of statements that together perform a task. Every C# program has at least one class with a method named Main.

To use a method, you need to:

- Define the method
- Call the method

Defining Methods in C#

When you define a method, you basically declare the elements of its structure. The syntax for defining a method in C# is as follows:

```
<Access Specifier> <Return Type> <Method Name> (Parameter List)
{
  Method Body
}
```

Following are the various elements of a method:

- Access Specifier : This determines the visibility of a variable or a method from another class.
- Return type: A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is void.
- Method name: Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.
- Parameter list: Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
- Method body: This contains the set of instructions needed to complete the required activity.

Example:

Following code snippet shows a function *FindMax* that takes two integer values and returns the larger of the two. It has public access specifier, so it can be accessed from outside the class using an instance of the class.

```
class NumberManipulator
{
  public int FindMax(int num1, int num2)
  {
    /* local variable declaration */
    int result;

    if (num1 > num2)
      result = num1;
    else
      result = num2;

    return result;
  }
  ...
}
```

Calling Methods in C#

You can call a method using the name of the method. The following example illustrates this:

```
using System;

namespace CalculatorApplication
{
  class NumberManipulator
  {
    public int FindMax(int num1, int num2)
    {
      /* local variable declaration */
      int result;
      if (num1 > num2)
      result = num1;
      else
      result = num2;
      return result;
    }
    public static void Main(string[] args)
    {
      /* local variable definition */
      int a = 100;
      int b = 200;
      int ret;
      NumberManipulator n = new NumberManipulator();
      //calling the FindMax method
      ret = n.FindMax(a, b);
      Console.WriteLine("Max value is : {0}", ret );
      Console.ReadLine();
    }
```

```
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Max value is : 200
```

You can also call public method from other classes by using the instance of the class. For example, the method *FindMax* belongs to the *NumberManipulator* class, you can call it from another class *Test*.

```
using System;

namespace CalculatorApplication
{
   class NumberManipulator
   {
      public int FindMax(int num1, int num2)
      {
         /* local variable declaration */
         int result;

         if (num1 > num2)
            result = num1;
         else
            result = num2;

         return result;
      }
   }
   class Test
   {
      static void Main(string[] args)
      {
         /* local variable definition */
         int a = 100;
         int b = 200;
         int ret;
         NumberManipulator n = new NumberManipulator();
         //calling the FindMax method
         ret = n.FindMax(a, b);
         Console.WriteLine("Max value is : {0}", ret );
         Console.ReadLine();

      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Max value is : 200
```

Recursive Method Call

A method can call itself. This is known as recursion. Following is an example that calculates factorial for a given number using a recursive function:

```
using System;
namespace CalculatorApplication
{
   class NumberManipulator
   {
      public int factorial(int num)
      {
         /* local variable declaration */
         int result;
         if (num == 1)
         {
           return 1;
         }
         else
         {
            result = factorial(num - 1) * num;
            return result;
         }
      }
      public static void Main(string[] args)
      {
         NumberManipulator n = new NumberManipulator();
         //calling the factorial method
         Console.WriteLine("Factorial of 6 is : {0}", n.factorial(6));
         Console.WriteLine("Factorial of 7 is : {0}", n.factorial(7));
         Console.WriteLine("Factorial of 8 is : {0}", n.factorial(8));
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Factorial of 6 is: 720
Factorial of 7 is: 5040
Factorial of 8 is: 40320
```

Passing Parameters to a Method

When method with parameters is called, you need to pass the parameters to the method. In C#, there are three ways that parameters can be passed to a method:

Mechanism	Description
Value parameters	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Reference parameters	This method copies the reference to the memory location of an argument into the formal parameter. This means that changes made to the parameter affect the argument.
Output parameters	This method helps in returning more than one value.

This is the default mechanism for passing parameters to a method. In this mechanism, when a method is called, a new storage location is created for each value parameter.

The values of the actual parameters are copied into them. So, the changes made to the parameter inside the method have no effect on the argument. The following example demonstrates the concept:

```
using System;

namespace CalculatorApplication
{
   class NumberManipulator
   {
      public void swap(int x, int y)
      {
         int temp;

         temp = x; /* save the value of x */
         x = y;    /* put y into x */
         y = temp; /* put temp into y */
      }
      public static void Main(string[] args)
      {
         NumberManipulator n = new NumberManipulator();
         /* local variable definition */
         int a = 100;
         int b = 200;

         Console.WriteLine("Before swap, value of a : {0}", a);
         Console.WriteLine("Before swap, value of b : {0}", b);

         /* calling a function to swap the values */
         n.swap(a, b);

         Console.WriteLine("After swap, value of a : {0}", a);
         Console.WriteLine("After swap, value of b : {0}", b);

         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

It shows that there is no change in the values though they had been changed inside the function.

A reference parameter is a reference to a memory location of a variable. When you pass parameters by reference, unlike value parameters, a new storage location is not created for these parameters. The reference parameters represent the same memory location as the actual parameters that are supplied to the method.

In C#, you declare the reference parameters using the ref keyword. The following example demonstrates this:

```
using System;
namespace CalculatorApplication
{
  class NumberManipulator
  {
    public void swap(ref int x, ref int y)
    {
      int temp;

      temp = x; /* save the value of x */
      x = y;   /* put y into x */
      y = temp; /* put temp into y */
     }

    static void Main(string[] args)
    {
      NumberManipulator n = new NumberManipulator();
      /* local variable definition */
      int a = 100;
      int b = 200;

      Console.WriteLine("Before swap, value of a : {0}", a);
      Console.WriteLine("Before swap, value of b : {0}", b);

      /* calling a function to swap the values */
      n.swap(ref a, ref b);

      Console.WriteLine("After swap, value of a : {0}", a);
      Console.WriteLine("After swap, value of b : {0}", b);

      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

It shows that the values have been changed inside the *swap* function and this change reflects in the *Main* function.

A return statement can be used for returning only one value from a function. However, using output parameters, you can return two values from a function. Output parameters are like reference parameters, except that they transfer data out of the method rather than into it.

The following example illustrates this:

```
using System;

namespace CalculatorApplication
{
  class NumberManipulator
  {
    public void getValue(out int x )
    {
      int temp = 5;
      x = temp;
    }

    public static void Main(string[] args)
    {
      NumberManipulator n = new NumberManipulator();
      /* local variable definition */
      int a = 100;

      Console.WriteLine("Before method call, value of a : {0}", a);

      /* calling a function to get the value */
      n.getValue(out a);

      Console.WriteLine("After method call, value of a : {0}", a);
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
Before method call, value of a : 100
After method call, value of a : 5
```

The variable supplied for the output parameter need not be assigned a value the method call. Output parameters are particularly useful when you need to return values from a method through the parameters without assigning an initial value to the parameter. Look at the following example, to understand this:

```
using System;
namespace CalculatorApplication
{
  class NumberManipulator
  {
    public void getValues(out int x, out int y )
    {
      Console.WriteLine("Enter the first value: ");
      x = Convert.ToInt32(Console.ReadLine());
      Console.WriteLine("Enter the second value: ");
      y = Convert.ToInt32(Console.ReadLine());
    }


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

```
    {
      NumberManipulator n = new NumberManipulator();
      /* local variable definition */
      int a , b;

      /* calling a function to get the values */
      n.getValues(out a, out b);

      Console.WriteLine("After method call, value of a : {0}", a);
      Console.WriteLine("After method call, value of b : {0}", b);
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces following result (depending upon the user input):

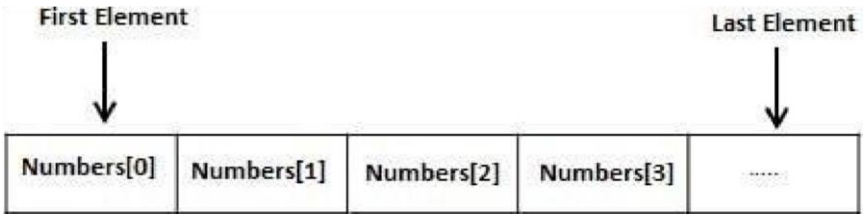
```
Enter the first value: 7
Enter the second value: 8
After method call, value of a : 7
After method call, value of b : 8
```

Chapter – 11 C# - ARRAYS

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays :

To declare an array in C#, you can use the following syntax:

```
datatype[] arrayName;
```

- where, d*atatype* is used to specify the type of elements to be stored in the array.
- *[ ]* specifies the rank of the array. The rank specifies the size of the array.
- *arrayName* specifies the name of the array.

For example,

```
double[]  balance;
```

Initializing an Array

Declaring an array does not initialize the array in the memory. When the array variable is initialized, you can assign values to the array.

Array is a reference type, so you need to use the new keyword to create an instance of the array.

For example,

```
double[] balance  =  new  double[10];
```

Assigning Values to an Array

You can assign values to individual array elements, by using the index number, like:

```
double[] balance = new double[10];
balance[0]  =  4500.0;
```

You can assign values to the array at the time of declaration, like:

```
double[] balance = { 2340.0, 4523.69, 3421.0};
```

You can also create and initialize an array, like:

```
 int [] marks = new int[5]  { 99,  98, 92, 97, 95};
```

In the preceding case, you may also omit the size of the array, like:

```
int [] marks = new int[]  { 99,  98, 92, 97, 95};
```

You can also copy an array variable another get array variable. In that case, both the target and source would into a point to the same memory tar location:

```
int [] marks = new int[]  { 99,  int[] score = marks;  98, 92,  97, 95};
```

When you create an array, C# compiler implicitly initializes each array element to a default value depending on the array type. For example for an int array all elements would be initialized to 0.

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double salary = balance[9];
```

Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

```
using System;
namespace ArrayApplication
{
  class MyArray
  {
    static void Main(string[] args)
    {
      /* n is an array of 10 integers */
      int []  n = new int[10];
      int i,j;

      /* initialize elements of array n */
      for ( i = 0; i < 10; i++ )
      {
```

```
        n[ i ] = i + 100;
      }

      /* output each array element's value */
      for (j = 0; j < 10; j++ )
      {
        Console.WriteLine("Element[{0}] = {1}", j, n[j]);
      }
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

Using the *foreach* Loop

In the previous example, we have used a for loop for accessing each array element. You can also use a foreach statement to iterate through an array.

```
using System;

namespace ArrayApplication
{
  class MyArray
  {
    public static void Main(string[] args)
    {
       int []  n = new int[10]; /* n is an array of 10 integers */

      /* initialize elements of array n */
      for ( int i = 0; i < 10; i++ )
      {
        n[i] = i + 100;
      }
      /* output each array element's value */
      foreach (int j in n )
      {
        int i = 0;
        Console.WriteLine("Element[{0}] = {1}", i, j);
        i++;
      }
      Console.ReadLine();
    }
  }
```

```
}
```

When the above code is compiled and executed, it produces following result:

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

C# Arrays in Detail

Arrays are important to C# and should need lots of more detail. There are following few important concepts related to array which should be clear to a C# programmer:

Concept	Description
Multi-dimensional arrays	C# supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
Jagged arrays	C# supports multidimensional arrays, which are arrays of arrays.
Passing arrays to functions	You can pass to the function a pointer to an array by specifying the array's name without an index.
Param arrays	This is used for passing unknown number of parameters to a function.
The Array Class	Defined in System namespace, it is the base class to all arrays, and provides various properties and methods for working with arrays.

C# allows multidimensional arrays. Multi-dimensional arrays are also called rectangular array.

You can declare a 2 dimensional array of strings as:

```
string [,] names;
```

or, a three dimensional array of int variables:

```
int [ , , ] m;
```

Two-Dimensional Arrays:

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays.

A two dimensional array can be thought of as a table which will have x number of rows and y number of columns. Following is a 2-dimentional array, which contains three rows and four columns:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form a[i , j], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int [,] a = int [3,4] = {
 {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
 {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
 {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

Accessing Two-Dimensional Array Elements

An element in 2-dimensional array is accessed by using the subscripts ie. row index and column index of the array. For example:

```
int val = a[2,3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above diagram. Let us check below program where we have used nested loop to handle a two dimensional array:

```
using System;

namespace ArrayApplication
{
   class MyArray
   {
      static void Main(string[] args)
      {
         /* an array with 5 rows and 2 columns*/
         int[,] a = new int[5, 2] {{0,0}, {1,2}, {2,4}, {3,6}, {4,8} };
         int i, j;

         /* output each array element's value */
         for (i = 0; i < 5; i++)
         {
```

```
            for (j = 0; j < 2; j++)
            {
               Console.WriteLine("a[{0},{1}] = {2}", i, j, a[i,j]);
            }
         }
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
a[0,0]: 0
a[0,1]: 0
a[1,0]: 1
a[1,1]: 2
a[2,0]: 2
a[2,1]: 4
a[3,0]: 3
a[3,1]: 6
a[4,0]: 4
a[4,1]: 8
```

A Jagged array is an array of arrays. You can declare a jagged array *scores* of int values as:

```
int [][] scores;
```

Declaring an array, does not create the array in memory. To create the above array:

```
int[][] scores = new int[5][];
for (int i = 0; i < scores.Length; x++)
{
   scores[i] = new int[4];
}
```

You can initialize a jagged array as:

```
int[][] scores = new int[2][]{new int[]{92,93,94},new int[]{85,66,87,88}};
```

Where, scores is an array of two arrays of integers -- scores[0] is an array of 3 integers and scores[1] is an array of 4 integers.

Example

The following example illustrates using a jagged array:

```
using System;

namespace ArrayApplication
{
   class MyArray
   {
```

```
      static void Main(string[] args)
      {
         /* a jagged array of 5 array of integers*/
         int[ ][ ] a = new int[ ][ ]{new int[ ]{0,0},new int[ ]{1,2}, new int[ ]{2,4},new int[ ]{ 3, 6 }, new int[ ]{ 4, 8 } };

         int i, j;

         /* output each array element's value */
         for (i = 0; i < 5; i++)
         {
            for (j = 0; j <; 2; j++)
            {
               Console.WriteLine("a[{0}][{1}] = {2}", i, j, a[i][j]);
            }
         }
        Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

You can pass an array as a function argument in C#. The following example demonstrates this:

```
using System;

namespace ArrayApplication
{
  class MyArray
  {
    double getAverage(int[] arr, int size)
    {
      int i;
      double avg;
      int sum = 0;

      for (i = 0; i < size; ++i)
      {
        sum += arr[i];
      }

      avg = (double)sum / size;
      return avg;
    }
    public static void Main(string[] args)
    {
```

```
      MyArray app = new MyArray();
      /* an int array with 5 elements */
      int [] balance = new int[]{1000, 2, 3, 17, 50};
      double avg;

      /* pass pointer to the array as an argument */
      avg = app.getAverage(balance, 5 ) ;

      /* output the returned value */
      Console.WriteLine( "Average value is: {0} ", avg );
      Console.ReadLine();
   }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
Average value is: 214.4
```

At times, while declaring a method you are not sure of the number of arguments passed as a parameter. C# param arrays (or parameter arrays) come into help at these times.

The following example demonstrates this:

```
using System;

namespace ArrayApplication
{
  class ParamArray
  {
    public int AddElements(params int[] arr)
    {
      int sum = 0;
      foreach (int i in arr)
      {
        sum += i;
      }
      return sum;
    }
  }

  class TestClass
  {
    public static void Main(string[] args)
    {
      ParamArray app = new ParamArray();
      int sum = app.AddElements(512, 720, 250, 567, 889);
      Console.WriteLine("The sum is: {0}", sum);
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
The sum is: 2938
```

The Array class is the base class for all the arrays in C#. It is defined in the System namespace. The Array class provides various properties and methods to work with arrays.

Properties of the Array Class

The following table provides some of the most commonly used properties of the Array class:

S.N	Property Name & Description
1	IsFixedSize Gets a value indicating whether the Array has a fixed size.
2	IsReadOnly Gets a value indicating whether the Array is read-only.
3	Length Gets a 32-bit integer that represents the total number of elements in all the dimensions of the Array.
4	LongLength Gets a 64-bit integer that represents the total number of elements in all the dimensions of the Array.
5	Rank Gets the rank (number of dimensions) of the Array.

Methods of the Array Class

The following table provides some of the most commonly used properties of the Array class:

S.N	Method Name & Description
1	Clear Sets a range of elements in the Array to zero, to false, or to null, depending on the element type.
2	Copy(Array, Array, Int32) Copies a range of elements from an Array starting at the first element and pastes them into another Array starting at the first element. The length is specified as a 32-bit integer.
3	CopyTo(Array, Int32) Copies all the elements of the current one-dimensional Array to the specified one-dimensional Array starting at the specified destination Array index. The index is specified as a 32-bit integer.
4	GetLength Gets a 32-bit integer that represents the number of elements in the specified dimension of the Array.
5	GetLongLength Gets a 64-bit integer that represents the number of elements in the specified dimension of the Array.
6	GetLowerBound Gets the lower bound of the specified dimension in the Array.
7	GetType Gets the Type of the current instance. (Inherited from Object.)
8	GetUpperBound Gets the upper bound of the specified dimension in the Array.
9	GetValue(Int32) Gets the value at the specified position in the one-dimensional Array. The index is specified as a 32-

	bit integer.
10	IndexOf(Array, Object) Searches for the specified object and returns the index of the first occurrence within the entire one-dimensional Array.
11	Reverse(Array) Reverses the sequence of the elements in the entire one-dimensional Array.
12	SetValue(Object, Int32) Sets a value to the element at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer.
13	Sort(Array) the elements in an entire one-dimensional Array using the IComparable implementation of each element of the Array.
14	ToString returns a string that represents the current object. (Inherited from Object.)

For complete list of Array class properties and methods, please consult Microsoft documentation on C#.

Example

The following program demonstrates use of some of the methods of the Array class:

```
using System;
namespace ArrayApplication
{
   class MyArray
   {
      public static void Main(string[] args)
      {
         int[] list = { 34, 72, 13, 44, 25, 30, 10 };
         int[] temp = list;

         Console.Write("Original Array: ");
         foreach (int i in list)
         {
            Console.Write(i + " ");
         }
         Console.WriteLine();

         // reverse the array
         Array.Reverse(temp);
         Console.Write("Reversed Array: ");
         foreach (int i in temp)
         {
            Console.Write(i + " ");
         }
         Console.WriteLine();

         //sort the array
         Array.Sort(list);
         Console.Write("Sorted Array: ");
         foreach (int i in list)
         {
            Console.Write(i + " ");
         }
```

```
            Console.WriteLine();
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces following result:

```
Original Array: 34 72 13 44 25 30 10
Reversed Array: 10 30 25 44 13 72 34
Sorted Array: 10 13 25 30 34 44 72
```

Chapter – 12 C# - STRINGS

In C# you can use strings as array of characters, however, more common practice is to use the string keyword to declare a string variable. The string keyword is an alias for the System.String class.

Creating a String Object

You can create string object using one of the following methods:

- By assigning a string literal to a String variable
- By using a String class constructor
- By using the string concatenation operator (+)
- By retrieving a property or calling a method that returns a string
- By calling a formatting method to convert a value or object to its string representation

The following example demonstrates this:

```
using System;

namespace StringApplication
{
   class Program
   {
      public static void Main(string[] args)
      {
         //from string literal and string concatenation
         string fname, lname;
         fname = "Rowan";
         lname = "Atkinson";
         string fullname = fname +" "+ lname;
         Console.WriteLine("Full Name: {0}", fullname);
         //by using string constructor
         char[] letters = { 'H', 'e', 'l', 'l','o' };
         string greetings = new string(letters);
         Console.WriteLine("Greetings: {0}", greetings);

         //methods returning string
         string[] sarray = { "Hello", "From", "Tutorials", "Point" };
         string message = String.Join(" ", sarray);
         Console.WriteLine("Message: {0}", message);

         //formatting method to convert a value
         DateTime waiting = new DateTime(2012, 10, 10, 17, 58, 1);
         string chat = String.Format("Message sent at {0:t} on {0:D}", waiting);
         Console.WriteLine("Message: {0}", chat);
         Console.ReadKey() ;
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Full Name: Rowan Atkinson
Greetings: Hello
Message: Hello From Tutorials Point
Message: Message sent at 5:58 PM on Wednesday, October 10, 2012
```

Properties of the String Class

The String class has the following two properties:

S.N	Property Name & Description
1	**Chars** Gets the *Char* object at a specified position in the current *String* object.
2	**Length** Gets the number of characters in the current String object.

Methods of the String Class

The String class has numerous methods that help you in working with the string objects. The following table provides some of the most commonly used methods:

S.N	Method Name & Description
1	**public static int Compare(string strA, string strB)** Compares two specified string objects and returns an integer that indicates their relative position in the sort order.
2	**public static int Compare(string strA, string strB, bool ignoreCase)** Compares two specified string objects and returns an integer that indicates their relative position in the sort order. However, it ignores case if the Boolean parameter is true.
3	**public static string Concat(string str0, string str1)** Concatenates two string objects.
4	**public static string Concat(string str0, string str1, string str2)** Concatenates three string objects.
5	**public static string Concat(string str0, string str1, string str2, string str3)** Concatenates four string objects.
6	**public bool Contains(string value)** Returns a value indicating whether the specified string object occurs within this string.
7	**public static string Copy(string str)** Creates a new String object with the same value as the specified string.
8	**public void CopyTo(int sourceIndex, char[] destination, int destinationIndex, int count)** Copies a specified number of characters from a specified position of the string object to a specified position in an array of Unicode characters.
9	**public bool EndsWith(string value)** Determines whether the end of the string object matches the specified string.
10	**public bool Equals(string value)** Determines whether the current string object and the specified string object have the same

	value.
11	public static bool Equals(string a, string b) Determines whether two specified string objects have the same value.
12	public static string Format(string format, Object arg0) Replaces one or more format items in a specified string with the string representation of a specified object.
13	public int IndexOf(char value) Returns the zero-based index of the first occurrence of the specified Unicode character in the current string.
14	public int IndexOf(string value) Returns the zero-based index of the first occurrence of the specified string in this instance.
15	public int IndexOf(char value, int startIndex) Returns the zero-based index of the first occurrence of the specified Unicode character in this string, starting search at the specified character position.
16	public int IndexOf(string value, int startIndex) Returns the zero-based index of the first occurrence of the specified string in this instance, starting search at the specified character position.
17	public int IndexOfAny(char[] anyOf) Returns the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters.
18	public int IndexOfAny(char[] anyOf, int startIndex) Returns the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters, starting search at the specified character position.
19	public string Insert(int startIndex, string value) Returns a new string in which a specified string is inserted at a specified index position in the current string object.
20	public static bool IsNullOrEmpty(string value) Indicates whether the specified string is null or an Empty string.
21	public static string Join(string separator, params string[] value) Concatenates all the elements of a string array, using the specified separator between each element.
22	public static string Join(string separator, string[] value, int startIndex, int count) Concatenates the specified elements of a string array, using the specified separator between each element.
23	public int LastIndexOf(char value) Returns the zero-based index position of the last occurrence of the specified Unicode character within the current string object.
24	public int LastIndexOf(string value) Returns the zero-based index position of the last occurrence of a specified string within the current string object.
25	public string Remove(int startIndex) Removes all the characters in the current instance, beginning at a specified position and continuing through the last position, and returns the string.
26	public string Remove(int startIndex, int count) Removes the specified number of characters in the current string beginning at a specified position and returns the string.
27	public string Replace(char oldChar, char newChar) Replaces all occurrences of a specified Unicode character in the current string object with the

	specified Unicode character and returns the new string.
28	public string Replace(string oldValue, string newValue) Replaces all occurrences of a specified string in the current string object with the specified string and returns the new string.
29	public string[] Split(params char[] separator) Returns a string array that contains the substrings in the current string object, delimited by elements of a specified Unicode character array.
30	public string[] Split(char[] separator, int count) Returns a string array that contains the substrings in the current string object, delimited by elements of a specified Unicode character array. The int parameter specifies the maximum number of substrings to return.
31	public bool StartsWith(string value) Determines whether the beginning of this string instance matches the specified string.
32	public char[] ToCharArray() Returns a Unicode character array with all the characters in the current string object.
33	public char[] ToCharArray(int startIndex, int length) Returns a Unicode character array with all the characters in the current string object, starting from the specified index and up to the specified length.
34	public string ToLower() Returns a copy of this string converted to lowercase.
35	public string ToUpper Returns a copy of this string converted to uppercase.
36	public string Trim() Removes all leading and trailing white-space characters from the current String object.

The above list of methods is not exhaustive, please visit MSDN library for the complete list of methods and String class constructors.

Examples:

The following example demonstrates some of the methods mentioned above: Comparing Strings:

```
using System;
namespace StringApplication
{
  class StringProg
  {
    public static void Main(string[] args)
    {
      string str1 = "This is test";
      string str2 = "This is text";
      if (String.Compare(str1, str2) == 0)
      {
        Console.WriteLine(str1 + " and " + str2 +  " are equal.");
      }
      else
      {
        Console.WriteLine(str1 + " and " + str2 + " are not equal.");
      }
      Console.ReadKey() ;
    }
  }
```

```
}
```

When the above code is compiled and executed, it produces following result:

```
This is test and This is text are not equal.
```

String Contains String:

```
using System;

namespace StringApplication
{
  class StringProg
  {
    public static void Main(string[] args)
    {
       string str = "This is test";
       if (str.Contains("test"))
       {
         Console.WriteLine("The sequence 'test' was found.");
       }
       Console.ReadKey() ;
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
The sequence 'test' was found.
```

Getting a Substring:

```
using System;

namespace StringApplication
{
  class StringProg
  {
     public static void Main(string[] args)
     {
        string str = "Last night I dreamt of San Pedro";
        Console.WriteLine(str);
        string substr = str.Substring(23);
        Console.WriteLine(substr);
     }
     Console.ReadKey() ;
  }
}
```

When the above code is compiled and executed, it produces following result:

```
San Pedro
```

Joining Strings:

```
using System;

namespace StringApplication
{
   class StringProg
   {
      public static void Main(string[] args)
      {
         string[] starray = new string[]{"Down the way nights are dark", "And the sun shines  daily on the mountain top",
         "I took a trip on a sailing ship", "And when I reached Jamaica", "I made a stop" };
         string str = String.Join("\n", starray);
         Console.WriteLine(str);
      }
      Console.ReadKey() ;
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Down the way nights are dark
And the sun shines daily on the mountain top
I took a trip on a sailing ship
And when I reached Jamaica
I made a stop
```

Chapter – 13 C# - STRUCTURES

In C#, a structure is a value type data type. It helps you to make a single variable hold related data of various data types. The struct keyword is used for creating a structure.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member for your program.

For example, here is the way you would declare the Book structure:

```
struct Books
{
    public string title;
    public string author;
    public string subject;
    public int book_id;
};
```

The following program shows the use of the structure:

```
using System;

struct Books
{
  public string title;
  public string author;
  public string subject;
  public int book_id;
};

public class testStructure
{
  public static void Main(string[] args)
  {
   Books Book1;
   /* Declare Book1 of type Book */
   Books Book2;
   /* Declare Book2 of type Book */
   /* book 1 specification */
   Book1.title = "C Programming";
```

```
   Book1.author = "Nuha Ali";
   Book1.subject = "C Programming Tutorial";
   Book1.book_id = 6495407;
   /* book 2 specification */
   Book2.title = "Telecom Billing";
   Book2.author = "Zara Ali";
   Book2.subject =  "Telecom Billing Tutorial";
   Book2.book_id = 6495700;
   /* print Book1 info */
   Console.WriteLine( "Book 1 title : {0}", Book1.title);
   Console.WriteLine("Book 1 author : {0}", Book1.author);
   Console.WriteLine("Book 1 subject : {0}", Book1.subject);
   Console.WriteLine("Book 1 book_id :{0}", Book1.book_id);
   /* print Book2 info */
   Console.WriteLine("Book 2 title : {0}", Book2.title);
   Console.WriteLine("Book 2 author : {0}", Book2.author);
   Console.WriteLine("Book 2 subject : {0}", Book2.subject);
   Console.WriteLine("Book 2 book_id : {0}", Book2.book_id);
   Console.ReadLine();
 }
}
```

When the above code is compiled and executed, it produces following result:

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Features of C# Structures

You have already used a simple structure named Books. Structures in C# are quite different from that in traditional C or C++. The C# structures have the following features:

- Structures can have methods, fields, indexers, properties, operator methods, and events.
- Structures can have defined constructors, but not destructors. However, you cannot define a default constructor for a structure. The default constructor is automatically defined and cant be changed.
- Unlike classes, structures cannot inherit other structures or classes.
- Structures cannot be used as a base for other structures or classes.
- A structure can implement one or more interfaces.
- Structure members cannot be specified as abstract, virtual, or protected.
- When you create a struct object using the New operator, it gets created and the appropriate constructor is called.
- Unlike classes, structs can be instantiated without using the New operator.

- If the New operator is not used, the fields will remain unassigned and the object cannot be used until all the fields are initialized.

Class vs Structure

Classes and Structures have the following basic differences:

classes are reference types and structs are value types structures do not support inheritance structures cannot have default constructor

In the light of the above discussions, let us rewrite the previous example:

```
using System;

struct Books
{
   private string title;
   private string author;
   private string subject;
   private int book_id;

   public void getValues(string t, string a, string s, int id)
   {
      title = t;
      author = a;
      subject = s;
      book_id = id;
   }
   public void display()
   {
     Console.WriteLine("Title : {0}", title);
     Console.WriteLine("Author : {0}", author);
     Console.WriteLine("Subject : {0}", subject);
     Console.WriteLine("Book_id :{0}", book_id);
   }
};
public class testStructure
{
   public static void Main(string[] args)
   {
     Books Book1 = new Books(); /* Declare Book1 of type Book */
     Books Book2 = new Books(); /* Declare Book2 of type Book */
     /* book 1 specification */
     Book1.getValues("C Programming","Nuha Ali", "C Programming Tutorial",6495407);
     /* book 2 specification */
     Book2.getValues("Telecom Billing","Zara Ali", "Telecom Billing Tutorial", 6495700);
     /* print Book1 info */
     Book1.display();
     /* print Book2 info */
     Book2.display();
     Console.ReadLine();
    }
}
```

When the above code is compiled and executed, it produces following result:

```
Title : C Programming
Author : Nuha Ali
Subject : C Programming Tutorial
Book_id : 6495407
Title : Telecom Billing
Author : Zara Ali
Subject : Telecom Billing Tutorial
Book_id : 6495700
```

Chapter – 14 C# - ENUMS

An enumeration is a set of named integer constants. An enumerated type is declared using the enum keyword.

C# enumerations are value data type. In other words, enumeration contains its own values and cannot inherit or cannot pass inheritance.

Declaring *enum* Variable

The general syntax for declaring an enumeration is:

```
enum <enum_name>
{
  enumeration list
};
```

Where,

- The *enum_name* specifies the enumeration type name.
- The *enumeration list* is a comma-separated list of identifiers.

Each of the symbols in the enumeration list stands for an integer value, one greater than the symbol that precedes it. By default, the value of the first enumeration symbol is 0. For example:

```
enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };
```

Example:

The following example demonstrates use of enum variable:

```
using System;

namespace EnumApplication
{
  class EnumProgram
  {
    enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

    public static void Main(string[] args)
    {
      int WeekdayStart = (int)Days.Mon;
      int WeekdayEnd = (int)Days.Fri;
      Console.WriteLine("Monday: {0}", WeekdayStart);
      Console.WriteLine("Friday: {0}", WeekdayEnd);
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
Monday: 1
Friday: 5
```

Chapter – 15 C# - CLASSES

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object. Objects are instances of a class. The methods and variables that constitute a class are called members of the class.

Class Definition

A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. Following is the general form of a class definition:

```
<access specifier> class  class_name
{
   // member variables
   <access specifier> <data type> variable1;    <access specifier>    <data type> variable2;    ...
   <access specifier> <data type> variableN;
   // member methods
   <access specifier> <return type> method1(parameter_list)
   {
      // method body
   }
   <access specifier> <return type> method2(parameter_list)
   {
      // method body
   }    ...
   <access specifier> <return type> methodN(parameter_list)
   {
      // method body
   }
}
```

Please note that,

- Access specifiers specify the access rules for the members as well as the class itself, if not mentioned then the default access specifier for a class type is internal. Default access for the members is private.
- Data type specifies the type of variable, and return type specifies the data type of the data, the method returns, if any.
- To access the class members, you will use the dot (.) operator.
- The dot operator links the name of an object with the name of a member.

The following example illustrates the concepts discussed so far:

```
using System;

namespace BoxApplication
{
   class Box
   {
      public double length;
      // Length of a box
      public double breadth;
      // Breadth of a box
      public double height;
     // Height of a box
   }

 class Boxtester
 {
      public static void Main(string[] args)
      {
         Box Box1 = new Box();       // Declare Box1 of type Box
         Box Box2 = new Box();      // Declare Box2 of type Box
         double volume = 0.0;       // Store the volume of a box here
         // box 1 specification
         Box1.height = 5.0;
         Box1.length = 6.0;
         Box1.breadth = 7.0;
         // box 2 specification
         Box2.height = 10.0;
         Box2.length = 12.0;
         Box2.breadth = 13.0;
         // volume of box 1
         volume = Box1.height * Box1.length * Box1.breadth;
         Console.WriteLine("Volume of Box1 : {0}",  volume);
         // volume of box 2
         volume = Box2.height * Box2.length * Box2.breadth;
         Console.WriteLine("Volume of Box2 : {0}", volume);
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Member Functions and Encapsulation

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Member variables are attributes of an object (from design perspective) and they are kept private to implement encapsulation. These variables can only be accessed using the public member functions.

Let us put above concepts to set and get the value of different class members in a class:

```
using     System;

namespace BoxApplication
{
   class Box
   {
      private double breadth;  // Breadth of a box
      private double height;    // height of a box
      private double length;

     public void getLength(double len)
     {
       length = len;
     }
     public void setBreadth( double bre )
     {
        breadth = bre;
     }
     public void setHeight( double hei )
     {
        height = hei;
     }
     public double getVolume()
     {
         return length * breadth * height;
     }
   }
   class Boxtester
   {
      public static void Main(string[] args)
      {
         Box Box1 = new Box();       // Declare Box1 of type Box
         Box Box2 = new Box();       // Declare Box2 of type Box
         double volume;
         // box 1 specification
         Box1.setLength(6.0);
         Box1.setBreadth(7.0);
         Box1.setHeight(5.0);
         // box 2 specification
         Box2.setLength(12.0);
         Box2.setBreadth(13.0);
```

```
         Box2.setHeight(10.0);
         // volume of box 1
         Volume =  Box1.getVolume();
         Console.WriteLine("Volume of Box1 : {0}" ,volume);
         // volume of box 2
         volume = Box2.getVolume();
         Console.WriteLine("Volume of Box2 : {0}", volume);
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Constructors in C#

A class constructor is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type. Following example explains the concept of constructor:

```
using System;

namespace LineApplication
{
  class Line
  {
    private double length;   // Length of a line
    public Line()
    {
      Console.WriteLine("Object is being created");
    }
    public void setLength( double len )
    {
      length = len;
    }
    public double getLength()
    {
       return length;
    }
    pubic static void Main(string[] args)
    {
      Line line = new Line();
      // set line length
      line.setLength(6.0);
      Console.WriteLine("Length of line : {0}", line.getLength());
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
Object is being created
Length of line : 6
```

A default constructor does not have any parameter but if you need a constructor can have parameters. Such constructors are called parameterized constructors. This technique helps you to assign initial value to an object at the time of its creation as shown in the following example:

```
using System;
namespace LineApplication
{
  class Line
  {
    private double length;   // Length of a line
    public Line(double len)  //Parameterized constructor
    {
      Console.WriteLine("Object is being created, length = {0}", len);
      length = len;
    }

    public void setLength( double len )
    {
      length = len;
    }
    public double getLength()
    {
      return length;
    }

    static void Main(string[] args)
    {
      Line line = new Line(10.0);
      Console.WriteLine("Length of line : {0}", line.getLength());
      // set line length
      line.setLength(6.0);
      Console.WriteLine("Length of line : {0}", line.getLength());
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

Destructor in C#

A destructor is a special member function of a class that is executed whenever an object of its class goes out of scope. A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters.

Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc. Destructors cannot be inherited or overloaded.

Following example explain the concept of destructor:

```
using System;
namespace LineApplication
{
  class Line
  {
    private double length;   // Length of a line
    public Line()  // constructor
    {
      Console.WriteLine("Object is being created");
    }
    ~Line() //destructor
    {
      Console.WriteLine("Object is being deleted");
    }

    public void setLength( double len )
    {
      length = len;
    }
    public double getLength()
    {
      return length;
    }

    static void Main(string[] args)
    {
      Line line = new Line();
      // set line length
      line.setLength(6.0);
      Console.WriteLine("Length of line : {0}", line.getLength());
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
Object is being created
Length of line : 6
Object is being deleted
```

Static Members of a C# Class

We can define class members as static using the static keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

The keyword static implies that only one instance of the member exists for a class. Static variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance of it. Static variables can be initialized outside the member function or class definition. You can also initialize static variables inside the class definition.
The following example demonstrates the use of static variables:

```
using System;
namespace StaticVarApplication
{
   class StaticVar
   {
      public static int num;
      public void count()
      {
         num++;
      }
      public int getNum()
      {
         return num;
      }
   }
   class StaticTester
   {
      static void Main(string[] args)
      {
         StaticVar s1 = new StaticVar();
         StaticVar s2 = new StaticVar();
         s1.count();
         s1.count();
         s1.count();
         s2.count();
         s2.count();
         s2.count();
         Console.WriteLine("Variable num for s1: {0}", s1.getNum());
         Console.WriteLine("Variable num for s2: {0}", s2.getNum());
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Variable num for s1: 6
Variable num for s2: 6
```

You can also declare a member function as static. Such functions can access only static variables. The static functions exist even before the object is created. The following example demonstrates the use of static functions:

```
using System;
namespace StaticVarApplication
{
   class StaticVar
   {
      public static int num;
      public void count()
      {
         num++;
      }
      public static int getNum()
      {
         return num;
      }
   }
   class StaticTester
   {
      static void Main(string[] args)
      {
         StaticVar s = new StaticVar();
         s.count();
         s.count();
         s.count();
         Console.WriteLine("Variable num: {0}", StaticVar.getNum());
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Variable num: 3
```

Chapter – 16 C# - INHERITANCE

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.

The idea of inheritance implements the IS-A relationship. For example, mammal IS A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base and Derived Classes

A class can be derived from more than one class or interface, which means that it can inherit data and functions from multiple base class or interface.

The syntax used in C# for creating derived classes is as follows:

```
<acess-specifier> class <base_class>
{
   ...
}
class <derived_class> : <base_class>
{
  ...
}
```

Consider a base class Shape and its derived class Rectangle:

```
using System;
namespace InheritanceApplication
{
  class Shape
  {
    public void setWidth(int w)
    {
       width = w;
    }
    public void setHeight(int h)
    {
       height = h;
    }
    protected int width;
    protected int height;
  }
  // Derived class
  class Rectangle: Shape
  {
```

```
    public int getArea()
    {
        return (width * height);
     }
   }
  class RectangleTester
  {
     public static void Main(string[] args)
     {
       Rectangle Rect = new Rectangle();
       Rect.setWidth(5);
       Rect.setHeight(7);
       // Print the area of the object.
       Console.WriteLine("Total area: {0}",  Rect.getArea());
     }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Total area: 35
```

Base Class Initialization

The derived class inherits the base class member variables and member methods. Therefore the super class object should be created before the subclass is created. You can give instructions for superclass initialization in the member initialization list.

The following program demonstrates this:

```
using System;
namespace RectangleApplication
{
  class Rectangle
  {
    //member variables
    protected double length;
    protected double width;
    public Rectangle(double l, double w)
    {
       length = l;
       width = w;
    }
    public double GetArea()
    {
       return length * width;
    }
    public void Display()
    {
      Console.WriteLine("Length: {0}", length);
      Console.WriteLine("Width: {0}", width);
      Console.WriteLine("Area: {0}", GetArea());
    }
  }//end class Rectangle
  class Tabletop : Rectangle
  {
     private double cost;
```

```
      public Tabletop(double l, double w) : base(l, w)
      {
      }
      public double GetCost()
      {
         double cost;
         cost = GetArea() * 70;
         return cost;
      }
      public void Display()
      {
        base.Display();
        Console.WriteLine("Cost: {0}", GetCost());
      }
    }
   class ExecuteRectangle
   {
      public static void Main(string[] args)
      {
        Tabletop t = new Tabletop(4.5, 7.5);
        t.Display();
        Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Length: 4.5
Width: 7.5
Area: 33.75
Cost: 2362.5
```

Multiple Inheritance in C#

C# does not support multiple inheritance. However, you can use interfaces to implement multiple inheritance. The following program demonstrates this:

```
using System;
namespace InheritanceApplication
{
  class Shape
  {
    public void setWidth(int w)
    {
      width = w;
    }
    public void setHeight(int h)
    {
      height = h;
    }
    protected int width;
    protected int height;
  }
```

```
// Base class PaintCost
  public interface PaintCost
  {
    int getCost(int area);

  }
  // Derived class
  class Rectangle : Shape, PaintCost
  {
    public int getArea()
    {
      return (width * height);
    }
    public int getCost(int area)
    {
      return area * 70;
    }
  }
  class RectangleTester
  {
    static void Main(string[] args)
    {
      Rectangle Rect = new Rectangle();
      int area;
      Rect.setWidth(5);
      Rect.setHeight(7);
      area = Rect.getArea();
      // Print the area of the object.
      Console.WriteLine("Total area: {0}",  Rect.getArea());
      Console.WriteLine("Total paint cost: ${0}" , Rect.getCost(area));
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
Total area: 35
Total paint cost: $2450
```

Chapter – 17 C# - POLYMORPHISM

The word polymorphism means having many forms. In object oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'.

Polymorphism can be static or dynamic. In static polymorphism the response to a function is determined at the compile time. In dynamic polymorphism it is decided at run time.

Static Polymorphism

The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism.

These are:

- Function overloading
- Operator overloading

We will discuss function overloading in the next section and operator overloading will be dealt with in next chapter.

Function Overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Following is the example where same function print() is being used to print different data types:

```
using     System;

namespace PolymorphismApplication
{
  class Printdata
  {
    void print(int i)
    {
      Console.WriteLine("Printing int: {0}", i );
    }
    void print(double f)
    {
      Console.WriteLine("Printing float: {0}" , f);
    }
    void print(string s)
    {
      Console.WriteLine("Printing string: {0}", s);
    }
    public static void Main(string[] args)
    {
      Printdata p = new Printdata();
```

```
         // Call print to print integer
          p.print(5);
         // Call print to print float
         p.print(500.263);
        // Call print to print string
         p.print("Hello C++");
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Printing int: 5
Printing float: 500.263
Printing string: Hello C++
```

Dynamic Polymorphism

C# allows you to create abstract classes that are used to provide partial class implementation of an interface.

Implementation is completed when a derived class inherits from it. Abstract classes contain abstract methods which are implemented by the derived class. The derived classes have more specialized functionality.

Please note the following rules about abstract classes:

- You cannot create an instance of an abstract class
- You cannot declare an abstract method outside an abstract class
- When a class is declared sealed, it cannot be inherited, abstract classes cannot be declared sealed.

The following program demonstrates an abstract class:

```
using     System;

namespace PolymorphismApplication
{
   abstract class Shape
   {
      public abstract int area();
   }
   class Rectangle:  Shape
   {
      private int length;
      private int width;
      public Rectangle( int a=0, int b=0)
      {
         length = a;
         width = b;
      }
      public override int area ()
      {
```

```
        Console.WriteLine("Rectangle class area :");
        return (width * length);
      }
 }
   class RectangleTester
   {
     public static void Main(string[] args)
     {
       Rectangle r = new Rectangle(10, 7);
       double a = r.area();
       Console.WriteLine("Area: {0}",a);
       Console.ReadLine();
     }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Rectangle class area :
Area: 70
```

When you have a function defined in a class that you want to be implemented in an inherited class(es), you use virtual functions. The virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.

Dynamic polymorphism is implemented by abstract classes and virtual functions.

The following program demonstrates this:

```
using     System;
namespace PolymorphismApplication
{
    class Shape
    {
       protected int width, height;

       public Shape( int a=0, int b=0)
       {
          width = a;
          height = b;
       }

      public virtual int area()
      {
        Console.WriteLine("Parent class area :");
        return 0;
    }
}

  class Rectangle: Shape
  {
    public Rectangle( int a=0, int b=0): base(a, b)
    {

    }
    public override int area ()
```

```
    {
      Console.WriteLine("Rectangle class area :");
       return (width * height);
    }
}
  class Triangle: Shape
  {
    public Triangle(int a = 0, int b = 0): base(a, b)
    {

    }
    public override int area()
    {
      Console.WriteLine("Triangle class area :");
      return (width * height / 2);
    }
  }
  class Caller
  {
    public void CallArea(Shape sh)
    {
       int a;
       a = sh.area();
       Console.WriteLine("Area: {0}", a);
    }
  }
  class Tester
  {
    public static void Main(string[] args)
    {
      Caller c = new Caller();
      Rectangle r = new Rectangle(10, 7);
      Triangle t = new Triangle(10, 5);
      c.CallArea(r);
      c.CallArea(t);
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
Rectangle class area:
Area: 70
Triangle class area:
Area: 25
```

Chapter – 18 C# - INTERFACES

An interface is defined as a syntactical contract that all the classes inheriting the interface should follow. The interface defines the 'what' part of the syntactical contract and the deriving classes define the 'how' part of the syntactical contract.

Interfaces define properties, methods and events, which are the members of the interface. Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members. It often helps in providing a standard structure that the deriving classes would follow.

Abstract classes to some extent serve the same purpose, however, they are mostly used when only few methods are to be declared by the base class and the deriving class implements the functionalities.

Declaring Interfaces

Interfaces are declared using the interface keyword. It is similar to class declaration. Interface statements are public by default. Following is an example of an interface declaration:

```
public interface ITransactions
{
  // interface members    void showTransaction();    double getAmount();
}
```

Example

The following example demonstrates implementation of the above interface:

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

namespace InterfaceApplication
{
  public interface ITransactions
  {
    // interface members
    void showTransaction();
    double getAmount();
  }
  public class Transaction : ITransactions
  {
    private string tCode;
    private string date;
    private double amount;
    public Transaction()
    {
      tCode = " ";
      date = " ";
      amount = 0.0;
    }
    public Transaction(string c, string d, double a)
    {
```

```
      tCode = c;
      date = d;
      amount = a;
    }
    public double getAmount()
    {
      return amount;
    }
    public void showTransaction()
    {
      Console.WriteLine("Transaction: {0}", tCode);
      Console.WriteLine("Date: {0}", date);
      Console.WriteLine("Amount: {0}", getAmount());

    }
  }
  class Tester
  {
    static void Main(string[] args)
    {
      Transaction t1 = new Transaction("001", "8/10/2012", 78900.00);
      Transaction t2 = new Transaction("002", "9/10/2012", 451900.00);
      t1.showTransaction();
      t2.showTransaction();
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
Transaction: 001
Date: 8/10/2012
Amount: 78900
Transaction: 002
Date: 9/10/2012
Amount: 451900
```

Chapter 19 – C# NAMESPACES

A namespace is designed for providing a way to keep one set of names separate from another. The class names declared in one namespace will not conflict with the same class names declared in another.

Defining a Namespace

A namespace definition begins with the keyword namespace followed by the namespace name as follows:

```
namespace namespace_name
{
   // code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:

```
namespace_name.item_name;
```

The following program demonstrates use of namespaces:

```
using System;
namespace first_space
{
   class namespace_cl
   {
      public void func()
      {
         Console.WriteLine("Inside first_space");
      }
   }
}
namespace second_space
{
   class namespace_cl
   {
      public void func()
      {
         Console.WriteLine("Inside second_space");
      }
   }
}
class TestClass
{
   public static void Main(string[] args)
   {
      first_space.namespace_cl fc = new first_space.namespace_cl();
      second_space.namespace_cl sc = new second_space.namespace_cl();
      fc.func();
      sc.func();
```

```
      Console.ReadLine();
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Inside first_space
Inside second_space
```

The *using* Keyword

The using keyword states that the program is using the names in the given namespace. For example, we are using the System namespace in our programs. The class Console is defined there. We just write:

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

We could have written the fully qualified name as:

```
System.Console.WriteLine("Hello there");
```

You can also avoid prepending of namespaces with the using namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code:

Let us rewrite our preceding example, with using directive:

```
using System;
using first_space;
using second_space;

namespace first_space
{
   class abc
   {
      public void func()
      {
         Console.WriteLine("Inside first_space");
      }
   }
}

namespace second_space
{
   class efg
   {
      public void func()
      {
         Console.WriteLine("Inside second_space");
      }
   }
}

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

```
  {
     abc fc = new abc();
     efg sc = new efg();
     fc.func();
     sc.func();
    Console.ReadLine();
  }
}
```

When the above code is compiled and executed, it produces following result:

```
Inside first_space Inside second_space
```

Nested Namespaces

Namespaces can be nested where you can define one namespace inside another namespace as follows:

```
namespace namespace_name1
{
  // code declarations
  namespace namespace_name2
  {
    // code declarations
  }
}
```

You can access members of nested namespace by using the dot (.) operator as follows:

```
using System;
using first_space;
using first_space.second_space;

namespace first_space
{
  class abc
  {
    public void func()
    {
      Console.WriteLine("Inside first_space");
    }
  }
  namespace second_space
  {
    class efg
    {
      public void func()
      {
        Console.WriteLine("Inside second_space");
      }
    }
  }
}

class TestClass
{
```

```
   public static void Main(string[] args)
   {
      abc fc = new abc();
      efg sc = new efg();
      fc.func();
      sc.func();
      Console.ReadLine();
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Inside first_space
Inside second_space
```

Chapter – 20 C# - EXCEPTION HANDLING

An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: try, catch, finally and throw.

- try: A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.
- catch: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- finally: The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
- throw: A program throws an exception when a problem shows up. This is done using a throw keyword.

Syntax

Assuming a block will raise and exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
  // statements causing exception
}
catch( ExceptionName e1 )
{
  // error handling code
}
catch( ExceptionName e2 )
{
  // error handling code
}
catch( ExceptionName eN )
{
  // error handling code
}
finally
{
  // statements to be executed
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Exception Classes in C#

C# exceptions are represented by classes. The exception classes in C# are mainly directly or indirectly derived from the System.Exception class. Some of the exception classes derived from the System.Exception class are the System.ApplicationException and System.SystemException classes.

The System.ApplicationException class supports exceptions generated by application programs. So the exceptions defined by the programmers should derive from this class. defined by the programmers should derive from this class.

The System.SystemException class is the base class for all predefined system exception.

The following table provides some of the predefined exception classes derived from the Sytem.SystemException class:

Exception Class	Description
System.IO.IOException	Handles I/O errors.
System.IndexOutOfRangeException	Handles errors generated when a method refers to an array index out of range.
System.ArrayTypeMismatchException	Handles errors generated when type is mismatched with the array type.
System.NullReferenceException	Handles errors generated from deferencing a null object.
System.DivideByZeroException	Handles errors generated from dividing a dividend with zero.
System.InvalidCastException	Handles errors generated during typecasting.
System.OutOfMemoryException	Handles errors generated from insufficient free memory.
System.StackOverflowException	Handles errors generated from stack overflow.

Handling Exceptions

C# provides a structured solution to the exception handling problems in the form of try and catch blocks. Using these blocks the core program statements are separated from the error-handling statements.

These error handling blocks are implemented using the try, catch and finally keywords. Following is an example of throwing an exception when dividing by zero condition occurs:

```
using System;

namespace ErrorHandlingApplication
{
   class DivNumbers
   {
      int result;
      DivNumbers()
      {
         result = 0;
      }
      public void division(int num1, int num2)
      {
         try
         {
            result = num1 / num2;
         }
         catch (DivideByZeroException e)
         {
            Console.WriteLine("Exception caught: {0}", e);
         }
         finally
         {
            Console.WriteLine("Result: {0}", result);
         }
      }
     public static void Main(string[] args)
      {
         DivNumbers d = new DivNumbers();
         d.division(25, 0);
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Exception caught: System.DivideByZeroException: Attempted to divide by zero.
at ...
Result: 0
```

Creating User-Defined Exceptions

You can also define your own exception. User defined exception classes are derived from the ApplicationException class. The following example demonstrates this:

```
using System;

namespace UserDefinedException
{
  class TestTemperature
  {
    public static void Main(string[] args)
```

```
    {
      Temperature temp = new Temperature();
      try
      {
        temp.showTemp();
      }
      catch(TempIsZeroException e)
      {
        Console.WriteLine("TempIsZeroException: {0}", e.Message);
      }
      Console.ReadLine();
    }
  }
}
public class TempIsZeroException: ApplicationException
{
  public TempIsZeroException(string message): base(message)
  {
  }
}
public class Temperature
{
  int temperature = 0;
  public void showTemp()
  {
    if(temperature == 0)
    {
      throw (new TempIsZeroException("Zero Temperature found"));
    }
    else
    {
      Console.WriteLine("Temperature: {0}", temperature);
    }
  }
}
```

When the above code is compiled and executed, it produces following result:

```
TempIsZeroException: Zero Temperature found
```

Throwing Objects

You can throw an object if it is either directly or indirectly derived from the System.Exception class. You can use a throw statement in the catch block to throw the present object as:

```
Catch(Exception e)
{
  ...
  Throw e
}
```

Chapter – 21 C# - FILE I/O

A file is a collection of data stored in a disk with a specific name and a directory path. When a file is opened for reading or writing, it becomes a stream.

The stream is basically the sequence of bytes passing through the communication path. There are two main streams: the input stream and the output stream. The input stream is used for reading data from file (read operation) and the output stream is used for writing into the file (write operation).

C# I/O Classes

The System.IO namespace has various class that are used for performing various operation with files, like creating and deleting files, reading from or writing to a file, closing a file etc.

The following table shows some commonly used non-abstract classes in the System.IO namespace:

I/O Class	Description
BinaryReader	Reads primitive data from a binary stream.
BinaryWriter	Writes primitive data in binary format.
BufferedStream	A temporary storage for a stream of bytes.
Directory	Helps in manipulating a directory structure.
DirectoryInfo	Used for performing operations on directories.
DriveInfo	Provides information for the drives.
File	Helps in manipulating files.
FileInfo	Used for performing operations on files.
FileStream	Used to read from and write to any location in a file.
MemoryStream	Used for random access to streamed data stored in memory.
Path	Performs operations on path information.
StreamReader	Used for reading characters from a byte stream.
StreamWriter	Is used for writing characters to a stream.
StringReader	Is used for reading from a string buffer.
StringWriter	Is used for writing into a string buffer.

The FileStream Class

The FileStream class in the System.IO namespace helps in reading from, writing to and closing files. This class derives from the abstract class Stream.

You need to create a FileStream object to create a new file or open an existing file. The syntax for creating a FileStream object is as follows:

For example, for creating a FileStream object F for reading a file named sample.txt:

```
FileStream <object_name> = new FileStream( <file_name>,
<FileMode Enumerator>, <FileAccess Enumerator>, <FileShare Enumerator>);
```

```
FileStream F = new FileStream("sample.txt", FileMode.Open, FileAccess.Read, FileShare.Read);
```

Parameter	Description
FileMode	The FileMode enumerator defines various methods for opening files. The members of the FileMode enumerator are: Append: It opens an existing file and puts cursor at the end of file, or creates the file, if the file does not exist. Create: It creates a new file. CreateNew: It specifies to the operating system, that it should create a new file. Open: It opens an existing file. OpenOrCreate: It specifies to the operating system that it should open a file if it exists, otherwise it should create a new file. Truncate: It opens an existing file and truncates its size to zero bytes.
FileAccess	FileAccess enumerators have members: Read, ReadWrite and Write.
FileShare	FileShare enumerators have the following members: Inheritable: It allows a file handle to pass inheritance to the child processes None: It declines sharing of the current file Read: It allows opening the file for reading ReadWrite: It allows opening the file for reading and writing Write: It allows opening the file for writing

The following program demonstrates use of the FileStream class:

```
using System;
using System.IO;

namespace FileIOApplication
{
   class Program
   {
      public static void Main(string[] args)
      {
         FileStream F = new FileStream("test.dat", FileMode.OpenOrCreate, FileAccess.ReadWrite);

         for (int i = 1; i <= 20; i++)
         {
            F.WriteByte((byte)i);
         }

         F.Position = 0;

         for (int i = 0; i <= 20; i++)
         {
            Console.Write(F.ReadByte() + " ");
         }
         F.Close();
         Console.ReadLine();
      }
   }
}
```

When above code is compiled and executed, it produces following results:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 -1
```

Advanced File Operations in C#

The preceding example provides simple file operations in C#. However, to utilize the immense powers of C# System.IO classes, you need to know the commonly used properties and methods of these classes.

We will discuss these classes and the operations they perform, in the following sections. :

Topic and Description
Reading from and Writing into Text files It involves reading from and writing into text files. The StreamReader and StreamWriter class helps to accomplish it.
Reading from and Writing into Binary files It involves reading from and writing into binary files. The BinaryReader and BinaryWriter class helps to accomplish this.
Manipulating the Windows file system It gives a C# programamer the ability to browse and locate Windows files and directories.

The StreamReader and StreamWriter classes are used for reading from and writing data to text files. These classes inherit from the abstract base class Stream, which supports reading and writing bytes into a file stream.

The StreamReader Class

The StreamReader class also inherits from the abstract base class TextReader that represents a reader for reading series of characters. The following table describes some of the commonly usedmethods of the StreamReader class:

S.N	Method Name & Purpose
1	public override void Close() It closes the StreamReader object and the underlying stream, and releases any system resources associated with the reader.
2	public override int Peek() Returns the next available character but does not consume it.
3	public override int Read() Reads the next character from the input stream and advances the character position by one character.

Example:

The following example demonstrates reading a text file named Jamaica.txt. The file reads:

```
Down the way where the nights are gay
And the sun shines daily on the mountain top
I took a trip on a sailing ship
And when I reached Jamaica
I made a stop
```

```
using System;
using System.IO;

namespace FileApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         try
         {
            // Create an instance of StreamReader to read from a file.
            // The using statement also closes the StreamReader.
            using (StreamReader sr = new StreamReader("c:\\jamaica.txt"))
            {
               string line;

               // Read and display lines from the file until
               // the end of the file is reached.
```

```
            while ((line = sr.ReadLine()) != null)
            {
                Console.WriteLine(line);
            }
        }
    }
    catch (Exception e)
    {
        // Let the user know what went wrong.
        Console.WriteLine("The file could not be read:");
        Console.WriteLine(e.Message);
    }
    Console.ReadLine();
  }
 }
}
```

Guess what it displays when you compile and run the program!

The StreamWriter Class

The StreamWriter class inherits from the abstract class TextWriter that represents a writer, which can write a series of character.

The following table shows some of the most commonly used methods of this class:

S.N	Method Name & Purpose
1	public override void Close() Closes the current StreamWriter object and the underlying stream.
2	public override void Flush() Clears all buffers for the current writer and causes any buffered data to be written to the underlying stream.
3	public virtual void Write(bool value) Writes the text representation of a Boolean value to the text string or stream. (Inherited from TextWriter.)
4	public override void Write(char value) Writes a character to the stream.
5	public virtual void Write(decimal value) Writes the text representation of a decimal value to the text string or stream.
6	public virtual void Write(double value) Writes the text representation of an 8-byte floating-point value to the text string or stream.
7	public virtual void Write(int value) Writes the text representation of a 4-byte signed integer to the text string or stream.
8	public override void Write(string value) Writes a string to the stream.

9	public virtual void WriteLine() Writes a line terminator to the text string or stream.

For complete list of methods, please visit Microsoft's C# documentation.

Example:

The following example demonstrates writing text data into a file using the StreamWriter class:

```
using System;
using System.IO;

namespace FileApplication
{
   class Program
   {
      public static void Main(string[] args)
      {
         string[] names = new string[] {"Zara Ali", "Nuha Ali"};
         using (StreamWriter sw = new StreamWriter("names.txt"))
         {
            foreach (string s in names)
            {
               sw.WriteLine(s);
            }
         }

         // Read and show each line from the file.
         string line = "";
         using (StreamReader sr = new StreamReader("names.txt"))
         {
            while ((line = sr.ReadLine()) != null)
            {
               Console.WriteLine(line);
            }
         }
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Zara Ali
Nuha Ali
```

Chapter – 22 C# - PROPERTIES

Properties are named members of classes, structures, and interfaces. Member variables or methods in a class or structures are called Fields. Properties are an extension of fields and are accessed using the same syntax. They use accessors through which the values of the private fields can be read, written or manipulated.

Properties do not name the storage locations. Instead, they have accessors that read, write, or compute their values.

For example, let us have a class named Student, with private fields for age, name and code. We cannot directly access these fields from outside the class scope, but we can have properties for accessing these private fields.

Accessors

The accessor of a property contains the executable statements that helps in getting (reading or computing) or setting (writing) the property. The accessor declarations can contain a get accessor, a set accessor, or both. For example:

```
// Declare a Code property of type string:
public string Code
{
  get
  {
    return code;
  }
  set
  {
    code = value;
  }
}

// Declare a Name property of type string:
public string Name
{
  get
  {
   return name;
  }
  set
  {
   name = value;
  }
}

// Declare a Age property of type int:
public int Age
{
  get
  {
    return age;
  }
```

```
  set
  {
    age = value;
  }
}
```

Example:

The following example demonstrates use of properties:

```
using System;

class Student
{
  private string code = "N.A";
  private string name = "not known";
  private int age = 0;

  // Declare a Code property of type string:
  public string Code
  {
    get
    {
      return code;
    }
    set
    {
      code = value;
    }
  }

  // Declare a Name property of type string:
  public string Name
  {
    get
    {
      return name;
    }
    set
    {
      name = value;
    }
  }

  // Declare a Age property of type int:
  public int Age
  {
    get
    {
      return age;
    }
    set
    {
      age = value;
    }
  }
  public override string ToString()
  {
```

```
    return "Code = " + Code +", Name = " + Name + ", Age = " + Age;
  }

  public static void Main()
  {
    // Create a new Student object:
    Student s = new Student();

    // Setting code, name and the age of the student
    s.Code = "001";
    s.Name = "Zara";
    s.Age = 9;
    Console.WriteLine("Student Info: {0}", s);
    //let us increase age
    s.Age += 1;
    Console.WriteLine("Student Info: {0}", s);
    Console.ReadLine();
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Student Info: Code = 001, Name = Zara, Age = 9
Student Info: Code = 001, Name = Zara, Age = 10
```

Abstract Properties

An abstract class may have an abstract property, which should be implemented in the derived class. The following program illustrates this:

```
using System;

public abstract class Person
{
  public abstract string Name
  {
    get;
    set;
  }
  public abstract int Age
  {
    get;
    set;
  }
}
class Student : Person
{
  private string code = "N.A";
  private string name = "N.A";
  private int age = 0;

  // Declare a Code property of type string:
  public string Code
  {
    get
    {
```

```
      return code;
    }
    set
    {
      code = value;
    }
  }

  // Declare a Name property of type string:
  public override string Name
  {
    get
    {
      return name;
    }
    set
    {
      name = value;
    }
  }

  // Declare a Age property of type int:
  public override int Age
  {
    get
    {
      return age;
    }
    set
    {
      age = value;
    }
  }
  public override string ToString()
  {
    return "Code = " + Code +", Name = " + Name + ", Age = " + Age;
  }

  public static void Main()
  {
    // Create a new Student object:
    Student s = new Student();

    // Setting code, name and the age of the student
    s.Code = "001";
    s.Name = "Zara";
    s.Age = 9;
    Console.WriteLine("Student Info:- {0}", s);
    //let us increase age
    s.Age += 1;
    Console.WriteLine("Student Info:- {0}", s);
    Console.ReadLine();
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Student Info: Code = 001, Name = Zara, Age = 9
Student Info: Code = 001, Name = Zara, Age = 10
```

Chapter – 23 C# - COLLECTIONS

Collection classes are specialized classes for data storage and retrieval. These classes provide support for stacks, queues, lists, and hash tables. Most collection classes implement the same interfaces.
Collection classes serve various purposes, such as allocating memory dynamically to elements and accessing a list of items on the basis of an index etc. These classes create collections of objects of the Object class, which is the base class for all data types in C#.

Various Collection Classes and Their Usage

The following are the various commonly used classes of the System.Collection namespace.
It represents an ordered collection of an object that can be indexed individually. It is basically an alternative to an array. However unlike array you can add and remove items from a list at a specified position using an index and the array resizes itself automatically. It also allow dynamic memory allocation, adding, searching and sorting items in the list.

Methods and Properties of the ArrayList Class

The following table lists some of the commonly used properties of the ArrayList class:

Property	Description
Capacity	Gets or sets the number of elements that the ArrayList can contain.
Count	Gets the number of elements actually contained in the ArrayList.
IsFixedSize	Gets a value indicating whether the ArrayList has a fixed size.
IsReadOnly	Gets a value indicating whether the ArrayList is read-only.
Item	Gets or sets the element at the specified index.

The following table lists some of the commonly used methods of the ArrayList class:

S.N	Method Name & Purpose
1	public virtual int Add(object value); Adds an object to the end of the ArrayList.
2	public virtual void AddRange(ICollection c); Adds the elements of an ICollection to the end of the ArrayList.
3	public virtual void Clear(); Removes all elements from the ArrayList.
4	public virtual bool Contains(object item); Determines whether an element is in the ArrayList.
5	public virtual ArrayList GetRange(int index, int count); Returns an ArrayList which represents a subset of the elements in the source ArrayList.

6	public virtual int IndexOf(object); Returns the zero-based index of the first occurrence of a value in the ArrayList or in a portion of it.
7	public virtual void Insert(int index, object value); Inserts an element into the ArrayList at the specified index.
8	public virtual void InsertRange(int index, ICollection c); Inserts the elements of a collection into the ArrayList at the specified index.
9	public virtual void Remove(object obj); Removes the first occurrence of a specific object from the ArrayList.
10	public virtual void RemoveAt(int index); Removes the element at the specified index of the ArrayList.
11	public virtual void RemoveRange(int index, int count); Removes a range of elements from the ArrayList.
12	public virtual void Reverse(); Reverses the order of the elements in the ArrayList.
13	public virtual void SetRange(int index, ICollection c); Copies the elements of a collection over a range of elements in the ArrayList.
14	public virtual void Sort(); Sorts the elements in the ArrayList.
15	public virtual void TrimToSize(); Sets the capacity to the actual number of elements in the ArrayList.

Example:

The following example demonstrates the concept:

```
using System;
using System.Collections;

namespace CollectionApplication
{
   class Program
   {
      public static void Main(string[] args)
      {
         ArrayList al = new ArrayList();

         Console.WriteLine("Adding some numbers:");
         al.Add(45);
         al.Add(78);
         al.Add(33);
         al.Add(56);
         al.Add(12);
         al.Add(23);
         al.Add(9);
```

```
            Console.WriteLine("Capacity: {0} ", al.Capacity);
            Console.WriteLine("Count: {0}", al.Count);

            Console.Write("Content: ");
            foreach (int i in al)
            {
               Console.Write(i + " ");
            }
            Console.WriteLine();
            Console.Write("Sorted Content: ");
            al.Sort();
            foreach (int i in al)
            {
               Console.Write(i + " ");
            }
            Console.WriteLine();
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces following result:

```
Adding some numbers:
Capacity: 8
Count: 7
Content: 45 78 33 56 12 23 9
Content: 9 12 23 33 45 56 78
```

Chapter – 24 C# - GENERICS

Generics allow you to delay the specification of the data type of programming elements in a class or a method, until it is actually used in the program. In other words, generics allow you to write a class or method that can work with any data type.

You write the specifications for the class or the method, with substitute parameters for data types. When the compiler encounters a constructor for the class or a function call for the method, it generates code to handle the specific data type. A simple example would help understanding the concept:

```
using System;

using System.Collections.Generic;

namespace GenericApplication
{
   public class MyGenericArray<T>
   {
      private T[] array;
      public MyGenericArray(int size)
      {
         array = new T[size + 1];
      }
      public T getItem(int index)
      {
         return array[index];
      }
      public void setItem(int index, T value)
      {
         array[index] = value;
      }
   }

   class Tester
   {
      static void Main(string[] args)
      {
         //declaring an int array
         MyGenericArray<int> intArray = new MyGenericArray<int>(5);
         //setting values
         for (int c = 0; c < 5; c++)
         {
            intArray.setItem(c, c*5);
         }
         //retrieving the values
         for (int c = 0; c < 5; c++)
         {
            Console.Write(intArray.getItem(c) + " ");
         }
         Console.WriteLine();
         //declaring a character array
         MyGenericArray<char> charArray = new MyGenericArray<char>(5);
         //setting values
         for (int c = 0; c < 5; c++)
         {
            charArray.setItem(c, (char)(c+97));
```

```
        }
        //retrieving the values
        for (int c = 0; c< 5; c++)
        {
           Console.Write(charArray.getItem(c) + " ");
        }
        Console.WriteLine();
        Console.ReadLine();
     }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
0 5  10 15 20
a b  c d e
```

Features of Generics

Using generics is a technique that enriches your programs in the following ways:

- It helps you to maximize code reuse, type safety, and performance.
  - You can create generic collection classes. The .NET Framework class library contains several new generic collection classes in the *System.Collections.Generic* namespace. You may use these generic collection classes instead of the collection classes in the *System.Collections* namespace.
  - You can create your own generic interfaces, classes, methods, events and delegates.
  - You may create generic classes constrained to enable access to methods on particular data types.
- You may get information on the types used in a generic data type at run-time by means of reflection.

Generic Methods

In the previous example, we have used a generic class; we can declare a generic method with a type parameter. The following program illustrates the concept:

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

namespace GenericMethodAppl
{
   class Program
   {
      static void Swap<T>(ref T lhs, ref T rhs)
      {
         T temp;
         temp = lhs;
         lhs = rhs;
         rhs = temp;
      }
      public static void Main(string[] args)
      {
```

```
            int a, b;
            char c, d;
            a = 10;
            b = 20;
            c = 'I';
            d = 'V';

            //display values before swap:
            Console.WriteLine("Int values before calling swap:");
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.WriteLine("Char values before calling swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);

            //call swap
            Swap<int>(ref a, ref b);
            Swap<char>(ref c, ref d);

            //display values after swap:
            Console.WriteLine("Int values after calling swap:");
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.WriteLine("Char values after calling swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces following result:

```
Int values before calling swap:
a = 10, b = 20
Char values before calling swap:
c = I, d = V
Int values after calling swap:  a = 20,
b = 10
Char values after calling swap: c = V, d = I
```

Chapter – 25 C# - ANONYMOUS METHODS

We discussed that delegates are used to reference any methods that has the same signature as that of the delegate. In other words, you can call a method that can be referenced by a delegate using that delegate object.

Anonymous methods provide a technique to pass a code block as a delegate parameter. Anonymous methods are basically methods without a name, just the body.
You need not specify the return type in an anonymous method; it is inferred from the return statement inside the method body.

Syntax for Writing an Anonymous Method

Anonymous methods are declared with the creation of the delegate instance, with a delegate keyword. For example,

```
delegate void NumberChanger(int n);
...
NumberChanger nc = delegate(int x)
{
   Console.WriteLine("Anonymous Method: {0}", x);
};
```

The code block *Console.WriteLine("Anonymous Method: {0}", x);* is the body of the anonymous method.
The delegate could be called both with anonymous methods as well as named methods in the same way, i.e., by passing the method parameters to the delegate object.
For example,

```
nc(10);
```

Example:

The following example demonstrates the concept:

```
using System;

delegate void NumberChanger(int n);
namespace DelegateAppl
{
   class TestDelegate
   {
      static int num = 10;
      public static void AddNum(int p)
      {
         num += p;
         Console.WriteLine("Named Method: {0}", num);
      }

      public static void MultNum(int q)
      {
```

```
         num *= q;
         Console.WriteLine("Named Method: {0}", num);
      }
      public static int getNum()
      {
         return num;
      }

      public static void Main(string[] args)
      {
         //create delegate instances using anonymous method
         NumberChanger nc = delegate(int x)
         {
           Console.WriteLine("Anonymous Method: {0}", x);
         };

         //calling the delegate using the anonymous method
         nc(10);

         //instantiating the delegate using the named methods
         nc =  new NumberChanger(AddNum);

         //calling the delegate using the named methods
         nc(5);

         //instantiating the delegate using another named methods
         nc =  new NumberChanger(MultNum);

         //calling the delegate using the named methods
         nc(2);
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
Anonymous Method: 10
Named Method: 15
Named Method: 30
```

Chapter – 26 C# - MULTITHREADING

A thread is defined as the execution path of a program. Each thread defines a unique flow of control. If your application involves complicated and time consuming operations then it is often helpful to set different execution paths or threads, with each thread performing a particular job.
Threads are lightweight processes. One common example of use of thread is implementation of concurrent programming by modern operating systems. Use of threads saves wastage of CPU cycle and increase efficiency of an application.

So far we have written programs where a single thread runs as a single process which is the running instance of the application. However, this way the application can perform one job at a time. To make it execute more than one task at a time, it could be divided into smaller threads.

Thread Life Cycle

The life cycle of a thread starts when an object of the System.Threading.Thread class is created and ends when the thread is terminated or completes execution.

Following are the various states in the life cycle of a thread:

- The Unstarted State : it is the situation when the instance of the thread is created but the Start method has not been called.
- The Ready State : it is the situation when the thread is ready to run and waiting CPU cycle.
- The Not Runnable State : a thread is not runnable, when:
  - Sleep method has been called
  - Wait method has been called

Blocked by I/O operations

The Dead State : It is the situation when the thread has completed execution or has been aborted.

The Main Thread

In C#, the System.Threading.Thread class is used for working with threads. It allows creating and accessing individual threads in a multithreaded application. The first thread to be executed in a process is called the main thread.

When a C# program starts execution, the main thread is automatically created. The threads created using the Thread class are called the child threads of the main thread. You can access a thread using the CurrentThread property of the Thread class.

The following program demonstrates main thread execution:

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
   class MainThreadProgram
   {
      public static void Main(string[] args)
      {
         Thread th = Thread.CurrentThread;
         th.Name = "MainThread";
         Console.WriteLine("This is {0}", th.Name);
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
This is MainThread
```

Commonly Used Properties and Methods of the Thread Class

The following table shows some of the most commonly used properties of the Thread class:

Property	Description
CurrentContext	Gets the current context in which the thread is executing.
CurrentCulture	Gets or sets the culture for the current thread.
CurrentPrinciple	Gets or sets the thread's current principal (for role-based security).
CurrentThread	Gets the currently running thread.
CurrentUICulture	Gets or sets the current culture used by the Resource Manager to look up culture-specific resources at run time.
ExecutionContext	Gets an ExecutionContext object that contains information about the various contexts of the current thread.
IsAlive	Gets a value indicating the execution status of the current thread.
IsBackground	Gets or sets a value indicating whether or not a thread is a background thread.
IsThreadPoolThread	Gets a value indicating whether or not a thread belongs to the managed thread pool.
ManagedThreadId	Gets a unique identifier for the current managed thread.

Name	Gets or sets the name of the thread.
Priority	Gets or sets a value indicating the scheduling priority of a thread.
ThreadState	Gets a value containing the states of the current thread.

The following table shows some of the most commonly used methods of the Thread class:

S.N	Method Name & Description
1	public void Abort() Raises a ThreadAbortException in the thread on which it is invoked, to begin the process of terminating the thread. Calling this method usually terminates the thread.
2	public static LocalDataStoreSlot AllocateDataSlot() Allocates an unnamed data slot on all the threads. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
3	public static LocalDataStoreSlot AllocateNamedDataSlot(string name) Allocates a named data slot on all threads. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
4	public static void BeginCriticalRegion() Notifies a host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exception might jeopardize other tasks in the application domain.
5	public static void BeginThreadAffinity() Notifies a host that managed code is about to execute instructions that depend on the identity of the current physical operating system thread.
6	public static void EndCriticalRegion() Notifies a host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exception are limited to the current task.
7	public static void EndThreadAffinity() Notifies a host that managed code has finished executing instructions that depend on the identity of the current physical operating system thread.
8	public static void FreeNamedDataSlot(string name) Eliminates the association between a name and a slot, for all threads in the process. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
9	public static Object GetData(LocalDataStoreSlot slot) Retrieves the value from the specified slot on the current thread, within the current thread's current domain. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
10	public static AppDomain GetDomain() Returns the current domain in which the current thread is running.

11	public static AppDomain GetDomain() Returns a unique application domain identifier
12	public static LocalDataStoreSlot GetNamedDataSlot(string name) Looks up a named data slot. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
13	public void Interrupt() Interrupts a thread that is in the WaitSleepJoin thread state.
14	public void Join() Blocks the calling thread until a thread terminates, while continuing to perform standard COM and SendMessage pumping. This method has different overloaded forms.
15	public static void MemoryBarrier() Synchronizes memory access as follows: The processor executing the current thread cannot reorder instructions in such a way that memory accesses prior to the call to MemoryBarrier execute after memory accesses that follow the call to MemoryBarrier.
16	public static void ResetAbort() Cancels an Abort requested for the current thread.
17	public static void SetData(LocalDataStoreSlot slot, Object data) Sets the data in the specified slot on the currently running thread, for that thread's current domain. For better performance, use fields marked with the ThreadStaticAttribute attribute instead.
18	public void Start() Starts a thread.
19	public static void Sleep(int millisecondsTimeout) Makes the thread pause for a period of time.
20	public static void SpinWait(int iterations) Causes a thread to wait the number of times defined by the iterations parameter
21	public static byte VolatileRead(ref byte address) public static double VolatileRead(ref double address) public static int VolatileRead(ref int address) public static Object VolatileRead(ref Object address) Reads the value of a field. The value is the latest written by any processor in a computer, regardless of the number of processors or the state of processor cache. This method has different overloaded forms. Only some are given above.
22	public static void VolatileWrite(ref byte address, byte value) public static void VolatileWrite(ref double address, double value) public static void VolatileWrite(ref int address, int value) public static void VolatileWrite(ref Object address, Object value) Writes a value to a field immediately, so that the value is visible to all processors in the computer. This method has different overloaded forms. Only some are given above.

23	public static bool Yield() Causes the calling thread to yield execution to another thread that is ready to run on the current processor. The operating system selects the thread to yield to.

Creating Threads

Threads are created by extending the Thread class. The extended Thread class then calls the Start()method to begin the child thread execution.

The following program demonstrates the concept:

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
   class ThreadCreationProgram
   {
      public static void CallToChildThread()
      {
         Console.WriteLine("Child thread starts");
      }

      public static void Main(string[] args)
      {
         ThreadStart childref = new ThreadStart(CallToChildThread);
         Console.WriteLine("In Main: Creating the Child thread");
         Thread childThread = new Thread(childref);
         childThread.Start();
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
In Main: Creating the Child thread
Child thread starts
```

Managing Threads

The Thread class provides various methods for managing threads.
The following example demonstrates the use of the sleep() method for making a thread pause for a specific period of time.

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
   class ThreadCreationProgram
   {
      public static void CallToChildThread()
      {
         Console.WriteLine("Child thread starts");
         // the thread is paused for 5000 milliseconds
         int sleepfor = 5000;
         Console.WriteLine("Child Thread Paused for {0} seconds", sleepfor / 1000);
         Thread.Sleep(sleepfor);
         Console.WriteLine("Child thread resumes");
      }

      public static void Main(string[] args)
      {
         ThreadStart childref = new ThreadStart(CallToChildThread);
         Console.WriteLine("In Main: Creating the Child thread");
         Thread childThread = new Thread(childref);
         childThread.Start();
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
In Main: Creating the Child thread
Child thread starts
Child Thread Paused for 5 seconds
Child thread resumes
```

Destroying Threads

The Abort() method is used for destroying threads.
The runtime aborts the thread by throwing a ThreadAbortException. This exception cannot be caught, the control is sent to the *finally* block, if any.

The following program illustrates this:

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
   class ThreadCreationProgram
   {
      public static void CallToChildThread()
      {
         try
         {
            Console.WriteLine("Child thread starts");
            // do some work, like counting to 10
            for (int counter = 0; counter <= 10; counter++)
            {
               Thread.Sleep(500);
               Console.WriteLine(counter);
            }
            Console.WriteLine("Child Thread Completed");
         }
         catch (ThreadAbortException e)
         {
            Console.WriteLine("Thread Abort Exception");
         }
         finally
         {
            Console.WriteLine("Couldn't catch the Thread Exception");
         }
      }
      public static void Main(string[] args)
      {
         ThreadStart childref = new ThreadStart(CallToChildThread);
         Console.WriteLine("In Main: Creating the Child thread");
         Thread childThread = new Thread(childref);
         childThread.Start();
         //stop the main thread for some time
         Thread.Sleep(2000);
         //now abort the child
         Console.WriteLine("In Main: Aborting the Child thread");
         childThread.Abort();
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
In Main: Creating the Child thread
Child thread starts
0
1
2
In Main: Aborting the Child thread
Thread Abort Exception
Couldn't catch the Thread Exception
```

Chapter – 27 C# - Delegates and Events

A delegate is a .NET class that encapsulates a method, but not in the same way other classes encapsulate methods. A delegate actually stores the address of a method that is contained in some other class. So, a delegate is really the equivalent of a function pointer in C++. However, they are also far more than that. In this article, I explain the many uses of delegates.

Delegate is a type which holds the method(s) reference in an object. It is also referred to as a type safe function pointer.
Advantages are as follows:

1) Encapsulating the method's call from caller
2) Effective use of delegate improves the performance of application
3) Used to call a method asynchronously

Declaring and using delegates

Declaration:--
public delegate type_of_delegate delegate_name()

Example:

public delegate int mydelegate(int delvar1,int delvar2)

You can use delegates without parameters or with parameter list
You should follow the same syntax as in the method
(If you are referring to the method with two int parameters and int return type, the delegate which you are declaring should be in the same format. This is why it is referred to as type safe function pointer.)
Sample Program using Delegate

```
 public delegate double Delegate_Prod(int a,int b);
class Class1
{
   static double fn_Prodvalues(int val1,int val2)
   {
      return val1*val2;
   }
   static void Main(string[] args)
   {
      //Creating the Delegate Instance
      Delegate_Prod delObj = new Delegate_Prod(fn_Prodvalues);
      Console.Write("Please Enter Values");
      int v1 = Int32.Parse(Console.ReadLine());
      int v2 = Int32.Parse(Console.ReadLine());
      //use a delegate for processing
      double res = delObj(v1,v2);
      Console.WriteLine ("Result :"+res);
```

```
        Console.ReadLine();
    }
}
```

Explanation:--

Here I have used a small program which demonstrates the use of delegate.

The delegate "Delegate_Prod" is declared with double return type and accepts only two integer parameters.

Inside the class, the method named fn_Prodvalues is defined with double return type and two integer parameters. (The delegate and method have the same signature and parameter type.)

Inside the Main method, the delegate instance is created and the function name is passed to the delegate instance as follows:

```
Delegate_Prod delObj = new Delegate_Prod(fn_Prodvalues);
```

After this, we are accepting the two values from the user and passing those values to the delegate as we do using method:

```
delObj(v1,v2);
```

Here delegate object encapsulates the method functionalities and returns the result as we specified in the method.

Multicast Delegate

What is Multicast Delegate?

It is a delegate which holds the reference of more than one method.

Multicast delegates must contain only methods that return void, else there is a run-time exception.

Simple Program using Multicast Delegate

```
delegate void Delegate_Multicast(int x, int y);
Class Class2
{
    static void Method1(int x, int y)
    {
        Console.WriteLine("You r in Method 1");
    }

    static void Method2(int x, int y)
    {
        Console.WriteLine("You r in Method 2");
    }

    public static void <place w:st="on" />Main</place />()
    {
```

```
        Delegate_Multicast func = new Delegate_Multicast(Method1);
        func += new Delegate_Multicast(Method2);
        func(1,2);          // Method1 and Method2 are called
        func -= new Delegate_Multicast(Method1);
        func(2,3);          // Only Method2 is called
    }
}
```

Explanation:--

In the above example, you can see that two methods are defined named method1 and method2 which take two integer parameters and return type as void.

In the main method, the Delegate object is created using the following statement:

Delegate_Multicast func = new Delegate_Multicast(Method1);
Then the Delegate is added using the += operator and removed using the -= operator.

Chapter – 28 C# - New Features in .NET 4.0

The C# 3.0 version brought the following new features to the language:

- **Implicitly typed variables**
- **Implicitly typed arrays**
- **Object & collection initializers**
- **Automatic properties**
- **Anonymous types**
- **Extension methods**
- **Query expressions**
- **Lambda expressions**
- **Expression trees**

Query expressions, lambda expressions and expression trees are beyond the scope of this tutorial. They are closely connected to the LINQ.

Implicitly typed local variables & arrays

Both implicitly typed local variables & arrays are connected with the var keyword. It is an implicit data type. In some cases, we do not have to specify the type for a variable. This does not mean, that C# is partially a dynamic language. C# remains a statically and strongly typed programming language. Sometimes, when the usage of the var keyword is allowed, the compiler will find and use the type for us. In some cases, the var keyword is not allowed. It can be used only on a local variable. It cannot be applied on a field in a class scope. It must be declared and initialized in one statement. The variable cannot be initialized to null.

```
using System;

public class CSharpApp
{
   public static void Main()
   {
      int x = 34;
      var y = 32.3f;

      var name = "Jane";

      Console.WriteLine(x);
      Console.WriteLine(y.GetType());
      Console.WriteLine(name.GetType());
   }
}
```

We have a small example, where we use the var type.

```
int x = 34;
var y = 32.3f;
```

The first variable is explicitly typed, the second variable is implicitly typed. The compiler will look at the right side of the assignment and infer the type of the variable.

```
Console.WriteLine(y.GetType());
Console.WriteLine(name.GetType());
```

These two lines will check the type of the two variables.

```
34
System.Single
System.String
```

As we can see, the two variables use the familiar, built-in data types.

Implicitly typed arrays Implicitly typed arrays are arrays, in which the type of the array is inferred from the elements of the array in the array initializer by the compiler. The rules are the same as for implicitly typed local variables. Implicitly typed arrays are mostly used in query expressions together with anonymous types and object and collection initializers.

```
using System;

public class CSharpApp
{
   public static void Main()
   {
      var items = new[] { "C#", "F#", "Python", "C" };

      foreach (var item in items)
      {
         Console.WriteLine(item);
      }
   }
}
```

An example demonstrating the implicitly typed arrays.

```
var items = new[] { "C#", "F#", "Python", "C" };
```

We again use the var keyword. The square brackets on the left side are omitted.

```
foreach (var item in items)
{
   Console.WriteLine(item);
}
```

The foreach loop is used to traverse the array. Note the use of the local implicitly typed item variable.

Output.

```
C#
F#
Python
C
```

Object initializers

Object initializers give a new syntax for creating and initiating objects. Inside a pair of curly brackets {} we initiate members of a class through a series of assignments. They are separated by comma character.

```
using System;

public class Person
{
   private string _name;
   private int _age;

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

   public int Age
   {
      get { return _age; }
      set { _age = value;}
   }

   public override string ToString()
   {
      return String.Format("{0} is {1} years old", _name, _age);
   }
}

public class CSharpApp
{
   static void Main()
   {
      Person p1 = new Person();
      p1.Name = "Jane";
      p1.Age = 17;

      Person p2 = new Person { Name="Becky", Age=18 };

      Console.WriteLine(p1);
      Console.WriteLine(p2);
   }
}
```

In the above example, we have a Person class with two properties. We create two instances of this class. We use the old way and we also use the object initializer expression.

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

This is a Name property, with set a get accessors.

```
Person p1 = new Person();
p1.Name = "Jane";
p1.Age = 17;
```

This is the old way of creating an object and initiating it with values using the field notation.

```
Person p2 = new Person { Name="Becky", Age=18 };
```

This is the object initializer. Inside curly brackets, the two members are initiated.

Collection initializers

Collection initializers are a way of initiating collections, where the elements of a collection are specified inside curly brackets.

```
List <string> planets = new List<string>();

planets.Add("Mercury");
planets.Add("Venus");
planets.Add("Earth");
planets.Add("Mars");
planets.Add("Jupiter");
planets.Add("Saturn");
planets.Add("Uranus");
planets.Add("Neptune");
```

This is the classic way of initiating a collection. In the following example, we will use a collection initializer for the same generic list.

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

public class CSharpApp
{
   public static void Main()
   {
      List <string> planets = new List <string>
         {"Mercury", "Venus", "Earth", "Mars", "Jupiter",
           "Saturn", "Uranus", "Neptune"};
```

```
        foreach (string planet in planets)
        {
            Console.WriteLine(planet);
        }
    }
}
```

We create a generic list of planets.

List <string> planets = new List <string> {"Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune"};

This is the collection initializer expression. The planet names are specified between the curly brackets. We save a some typing. We do not need to call the Add() method for each item of the collection.

Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
Output.

Automatic properties

In a software project, there are lots of simple properties, that only set or get some simple values. To simplify programming and to make the code shorter, automatic properties were created. Note that we cannot use automatic properties in all cases. Only for the simple ones.

```
using System;

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

public class CSharpApp
{
    public static void Main()
    {
        Person p = new Person();
        p.Name = "Jane";
        p.Age = 17;

        Console.WriteLine("{0} is {1} years old", p.Name, p.Age);
    }
}
```

This code is much shorter. We have a person class in which we have two properties.

```
public string Name { get; set; }
public int Age { get; set; }
```

Here we have two automatic properties. There is no implementation of the accessors. And there are no member fields. The compiler will do the rest for us.

```
Person p = new Person();
p.Name = "Jane";
p.Age = 17;

Console.WriteLine("{0} is {1} years old", p.Name, p.Age);
```

We normally use the properties as usual.

Output of the example.

Anonymous types

The class Person, that we use here is said to be a type. More specifically, it is a user defined type. The type has a name and we can explicitly create an instance of it by referring to its name. Anonymous types are types, that do not have a name. They are class types that consist of one or more public read-only properties. No other kinds of class members are allowed. Anonymous types are created with the new keyword and object initializer. The compiler will infer the types of the properties itself. It will give the type a name, but it is only used by the compiler; it is not available to the programmer. In fact, at compile type, the compiler generates a type from the anonymous type expression. And at runtime, an object is created out of the type.

```
using System;

public class CSharpApp
{
   public static void Main()
   {
      var p = new {Name="Jane", Age=17};

      Console.WriteLine("{0} is {1} years old", p.Name, p.Age);
   }
}
```

An anonymous type example.

```
var p = new {Name="Jane", Age=17};
```

An anonymous object initializer declares an anonymous type and returns an instance of that type. We use the var keyword, because we do not know the type.

```
Console.WriteLine("{0} is {1} years old", p.Name, p.Age);
```

We access the properties created using the field access notation.

Output.

Extension methods

Developers often face situations, in which they would like to extend an existing type, but it is not possible. For example, the class is sealed. The extension method is a workaround for such cases. Extension methods are a special kind of a static method, but they are called as if they were instance methods on the extended type. To create an extension method, we need a static class and a static method. When we call our extention method on a class, the compiler does some behind the scenes processing; for us it appears as if we have called the extension method on the object of a type.
It is also possible, and it was a common workaround in the past, to create special utility classes for such methods. These were mostly created as static methods of static classes. Both approaches have their advantages. It is also advised to use extension methods sparingly.

```
using System;
public static class Util
{
  public static string Reverse(this string input)
  {
    char[] chars = input.ToCharArray();
    Array.Reverse(chars);
    return new String(chars);
  }
}
public class CSharpApp
{
    public static void Main()
    {
        string str1 = "Jane";
        string str2 = str1.Reverse();
        Console.WriteLine(str2);
    }
}
```

In our case, we would like to add a new method for a String class. The String class is a built-in class and it is sealed. No inheritance is possible. This is why we need an extension method.

```
public static class Util
{
  public static string Reverse(this string input)
  {
    char[] chars = input.ToCharArray();
    Array.Reverse(chars);
    return new String(chars);
  }
}
```

We have a Reverse() extension method. This method reverses characters of a string variable. The method and its class must be static. The static Reverse() method becomes an extension method, when we put the this modifier as the first parameter of the method.

```
string str1 = "Jane";
string str2 = str1.Reverse();
```

We define and initialize a string variable. We call the Reverse() method on this variable. Even though the method is static, we use the instance method call syntax.