SRINIVAS UNIVERSITY

COLLEGE OF COMPUTER & INFORMATION SCIENCE

CITY CAMPUS, PANDESHWAR, MANGALORE – 575 001

BACKGROUND STUDY MATERIAL

.NET FRAMEWORK & C#

M.C.A - I SEMESTER

Compiled by

Prof. Vaikunth Pai

Faculty, CCIS

Types of Application Architectures

Applications are developed to support organizations in their business operations. Applications accept input, process the data based on business rules, and provide data as output. The functions performed by an application can be divided into three categories: user services, business services, and data services. Each category is implemented as a layer in an application. The user services layer constitutes the front-end of a solution. It is also called a presentation layer because it provides an interactive user interface. The business services layer controls the enforcement of business rules on the data of an organization. Business rules encompass those practices and activities that define the behavior of an organization. For example, an organization may have decided that the credit limit of clients cannot exceed $200000. The business services layer performs validations pertaining to business rules. It ensures that the back-end does not receive incorrect data. The data services layer comprises the data and the functions for manipulating this data. These three layers form the base of the models or architectures used in application development.

Applications may vary from single-tier desktop applications (applications that follow the single-tier architecture) to multi-tier applications (applications that follow the two-, three-, or n-tier architecture).

Single-tier Architecture

In the case of the single-tier architecture, a single executable file handles all functions relating to the user, business, and data service layers.

Two-tier Architecture

The two-tier architecture divides an application into the following two components:

Client: Implements the user interface

Server: Stores data

Thus, in the case of the two-tier architecture, the user and data services are located separately, either on the same machine or on separate machines. For example, you might have a Visual Basic application, which provides the user interface and SQL Server 7.0, which manages data.

In the two-tier architecture, the business services layer may be implemented in one of the following ways:

1. By using fat client
2. By using fat server
3. By dividing the business services between the user services and the data services

Fat Client

In the case of fat clients, the business services layer is combined with the user services layer. Clients execute the presentation logic and enforce business rules. The server stores data and processes transactions. The fat client model is used when the server is overloaded with transaction processing activities and is not equipped to process business logic.

Fat Server

In two-tier architecture with a fat server, the business services layer is combined with the data services layer. As business services are stored on the server, most of the processing takes place on the server.

Dividing Business Services between the User and Data Services

You can also implement a two-tier model in which the business services are distributed between the user and data services. In this case, the processing of the business logic is distributed between the user and data services.

Three-Tier Architecture

In the case of the three-tier architecture, all the three service layers reside separately, either on the same machine or on different machines. The user interface interacts with the business logic. The business logic validates the data sent by the interfaces and forwards it to the database if it conforms to the requirements. The front-end only interacts with business logic, which, in turn, interacts with the database.

n-tier Architecture

An n-tier application uses business objects for handling business rules and data access. It has multiple servers handling business services. This application architecture provides various advantages over other types of application architectures. Some of the advantages include extensibility, flexibility to change, maintainability, and scalability of the application.

Most modem enterprise applications are based on the n-tier application architecture. These applications serve organization-specific requirements. However, the introduction of the Internet and its rapid growth in the recent past has led to the development of a number of new technologies. It has also led to an increase in the number of Web applications and their requirements in terms of payment handling, data access, and security. One of the most important requirements of such applications is the ability to interchange information across platforms and to benefit from the functionality provided by other applications. In other words, there should be increased interoperability and integration between various applications. In the current scenario, although applications serve organization-specific requirements, they are not interoperable. For example, a Web site may register its customers and provide services to them. However, it may not be capable of interacting with another Web site that provides complementary services to customers. For performing such tasks, applications need to provide cross-platform application integration.

Evolution of .NET

With the change in technology, the approach to application development has changed completely. Earlier, the developers had to learn different programming languages to develop different types of applications. This was a complicated task.

The .NET framework has simplified this task by providing a common platform that enables the developers to develop different types of applications easily and quickly.

Introduction to .NET Framework

Microsoft has introduced the .NET initiative with the intention of bridging the gap in interoperability between applications. It aims at integrating various programming languages and services.

The .NET initiative offers a complete suite for developing and deploying applications. This suite consists of .NET products, .NET services, and the .NET Framework.

.NET products: Microsoft introduced Visual Studio .NET, which is a tool for developing .NET applications by using programming languages such as Visual Basic, C#, and Visual C++. In addition, Microsoft also intends to introduce .NET versions of the Windows operating system. These products aim at allowing developers to create applications that are capable of interacting seamlessly with each other. To ensure interaction between different applications, all .NET products use eXtensible Markup Language (XML) for describing and exchanging data between applications.

XML is a cross-platform, hardware and software independent markup language. XML allows computers to store data in a format that can be interpreted by any other computer system. Therefore, XML can be used to transfer structured data between heterogeneous systems. XML is used as a common data interchange format in a number of applications.

.NET services: .NET delivers software as Web services. Therefore, users can subscribe to a Web service and use it as long as they need it, regardless of the hardware and software platform. Microsoft is coming up with its own set of Web services, known as My Services. These services are based on the Microsoft Passport Authentication service, the same service that is used in Hotmail. They allow the consumers of the services to access data by linking calendars, phonebooks, address books, and personal references to the Passport Authentication service. In addition to the Web services provided by Microsoft, third party products and services can also be integrated easily with the .NET environment.

The .NET Framework: is a software development environment for designing, developing, deploying, and executing Windows and Web-based applications and it is a platform designed for developing interoperable applications such as Windows, Web, and Console. The objective of the .NET framework is to bring various programming languages and services together and make it easier to develop applications that run over the Internet anywhere, any time, on any platform and on any devices.

Using the .NET framework, we can create applications that are:

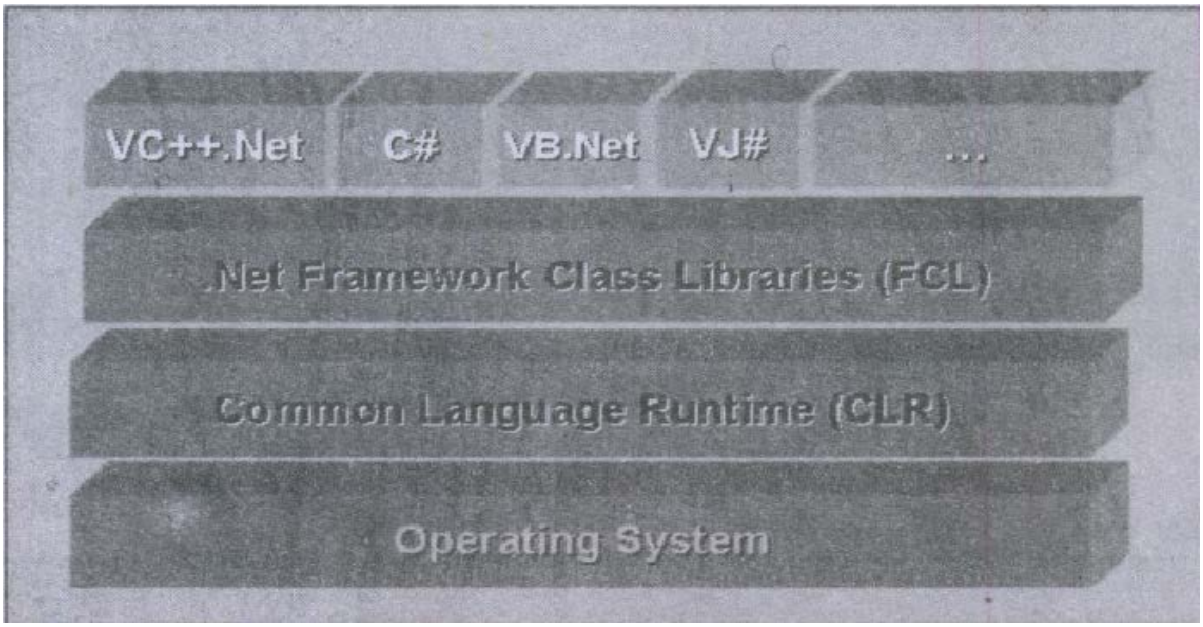
Robust: Applications that remains stable in extreme situations, such as erroneous inputs

Scalable: Applications that can be extended

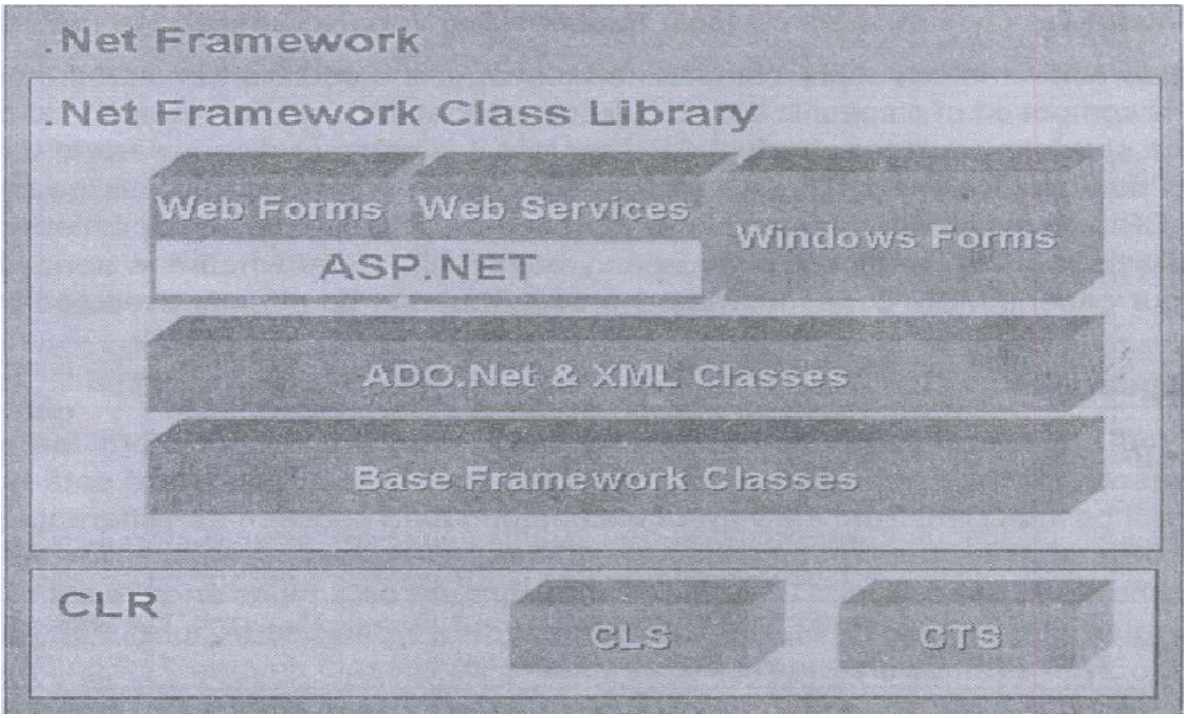
Distributed: Applications whose components run on different computers over a network

Components of the .NET Framework

The following figure shows the architecture of the .NET Framework



The .NET Framework consists of Web Forms, Windows Forms, and Console applications that pertain to the presentation layer of an application. Besides these three components, the .NET Framework consists of two other components, the .NET Framework Class Library (FCL) and Common Language Runtime (CLR). The following figure shows the components of the .NET framework and their constituents



Framework Class Library (FCL)

The class libraries are the collection of reusable, manageable classes and interfaces. These classes are organized in very well known hierarchical structure. Organization of these class libraries according to their functionality is called Namespaces. FCL is a standard library available to all languages using .NET framework. Using these libraries, you can accomplish different tasks such as database connectivity, string handling, input/output functionality, numerical functions, file handling and graphic rendering. These libraries are pre-coded solutions for the developer. We can access the classes of this library using full namespace notation. Thus .NET Framework Base Class Library comprises namespaces, which are contained within assemblies. The namespaces help us to create logical groups of related classes and interfaces, which can be used by any language supported by .NET framework. An assembly is a single deployable unit that contains all the information about the implementation of classes, structures and interfaces.

Windows Forms provides a set of classes to design forms for windows-based applications

Web forms (ASP.NET) provides a set of classes to design forms for the web pages similar to the HTML Forms and provides a set of classes to build web applications.

Database Connectivity (ADO. Net) provides classes to interact with databases.

Web Services includes a set of classes to design applications that can be accessed using a standard set of protocols.

XML Classes enables XML manipulation, searching and translations.

Common Language Runtime (CLR)

Every programming language generally has a compiler and a runtime environment of its own. The compiler converts your code into executable code that can be run by the users. The runtime environment provides the operating system services to your executable code. These services include memory management and file I/O. Before .Net came into existence, each language came with its own run time environment. For example, Visual Basic came with a DLL called MSVBVM60.DLL. Similarly, Visual C++ came with a DLL called MSVCRT.DLL. Thus there was a need of a runtime environment that enables you to execute the code irrespective of the language used.

One of the primary goals of .NET framework is to combine the runtime environments so that the developers can work with a single set of runtime services. Thus, the .NET Framework provides a Common Language Runtime (CLR) that offers capabilities such as memory management, security and robust error-handling to any language that works with the .NET Framework. All the code in .NET is managed by the CLR and is therefore called as 'managed code'. The managed code contains the information about the code such as classes, methods and variables defined in the code.

When you compile the code written in any .NET compatible language, the code is converted into Microsoft Intermediate Language or MSIL. MSIL is a CPU-independent instruction set that indicate how your code should be executed. MSIL is not a specific instruction set developed for physical CPU. This means that the code can be executed from any platform that supports the .NET CLR. However, when you run the code for the first time, the MSIL code is turned into CPU-specific code. This process is called as "just-in-time" compilation, or JIT. The "just-in-time" part of the name here reflects the fact that MSIL code is only compiled as and when it is needed. The main job of a JIT compiler is to translate your generic MSIL code into machine code that can be directly executed by your CPU that is JIT compiler compiles the MSIL code into a native code that is specific to the OS and machine architecture being targeted.

Common Language Specification (CLS) and Common Type System (CTS)

The one of the goals of .NET is to provide interoperability between applications. In order to create interoperable applications, you will need a set of standard data types that would be used across applications. This will be important from the view of exchanging data between the different applications. Additionally you will also need a set of guidelines for creating the user-defined classes and objects for the .NET Framework. The Common Type System commonly referred to as CTS contains the standard data types and the set of guidelines that defines the rules by which all types are declared, defined and managed, regardless of source language. This means that type int in C++ language is the same as the type int in C# language. Considering the main aim of providing interoperability, CTS includes only those data types and features that are compatible across the languages supported by the .NET Framework. The CTS is designed to be rich and flexible enough to support a wide variety of

source languages, and is the basis for cross-language integration, type safety, and managed execution services.

For example, consider that you have created some part of an application using C++. Later, in order to provide an interactive user interface to your application, you decide to recreate the entire application using ASP.NET. The problem that arises here is that, C++ and ASP.NET are not interoperable. As a result, you have to recreate all the classes that you may have created in C++, once again in ASP.NET. This is exactly where CTS feature of the .NET Framework plays an important role. If you have created a class using any language under the .NET Framework, CTS enables you to reuse that same class in any other language supported by the .NET framework.

In addition to CTS, Common Language Specification (CLS) is one more feature that ensures language interoperability in .NET Framework. Since different languages support different set of constructs (types), such as C# supports decimal type and C++ does not support decimal type, they become a bottleneck to carry on with cross-language integration or language interoperability. To solve this, Framework defines Common Language Specification (CLS), as a set of rules that any .NET language should follow in order to create applications that are interoperable with other languages. However, one important thing to remember here is to achieve interoperability across languages, you can only use objects with features listed in the CLS.

For example one of the data types uint32 (32-bit unsigned integer), supported by C# is not compliant with CLS. Visual Basic .NET does not support this data type. Thus, in this case, if you are creating an application having a class created using C# that uses data type uint32, this class might not be interoperable with Visual Basic .NET. This means that in order to implement interoperability, the languages have to use CLS-compliant code in applications.

CLS, CTS and MSIL work together closely to implement language interoperability.

The process of compilation and execution of a .NET application:

1. A program written in any .NET compatible language is compiled with their respective compilers (CSC.exe or VBC.exe) to yield .exe or .dll file containing IL code that is instead of compiling the source code into machine-level code, the compiler translates it into Microsoft Intermediate Language (MSIL) or Intermediate Language (IL). No matter which language has been used to develop the application, it always gets translated into IL. This ensures language interoperability.

2. In addition to translating the code into IL, the compiler also produces metadata about the program during the process of compilation. Metadata contains the description of the program, such as the classes and interfaces, the dependencies, and the versions of the components used in the program.

3. The IL and the metadata are linked in an assembly.

4. The compiler creates the .EXE or .DLL file.

5. When you execute the .EXE or .DLL file, the code (converted into IL) and all the other relevant information from the base class library is sent to the class loader. The class loader loads the code in the memory.

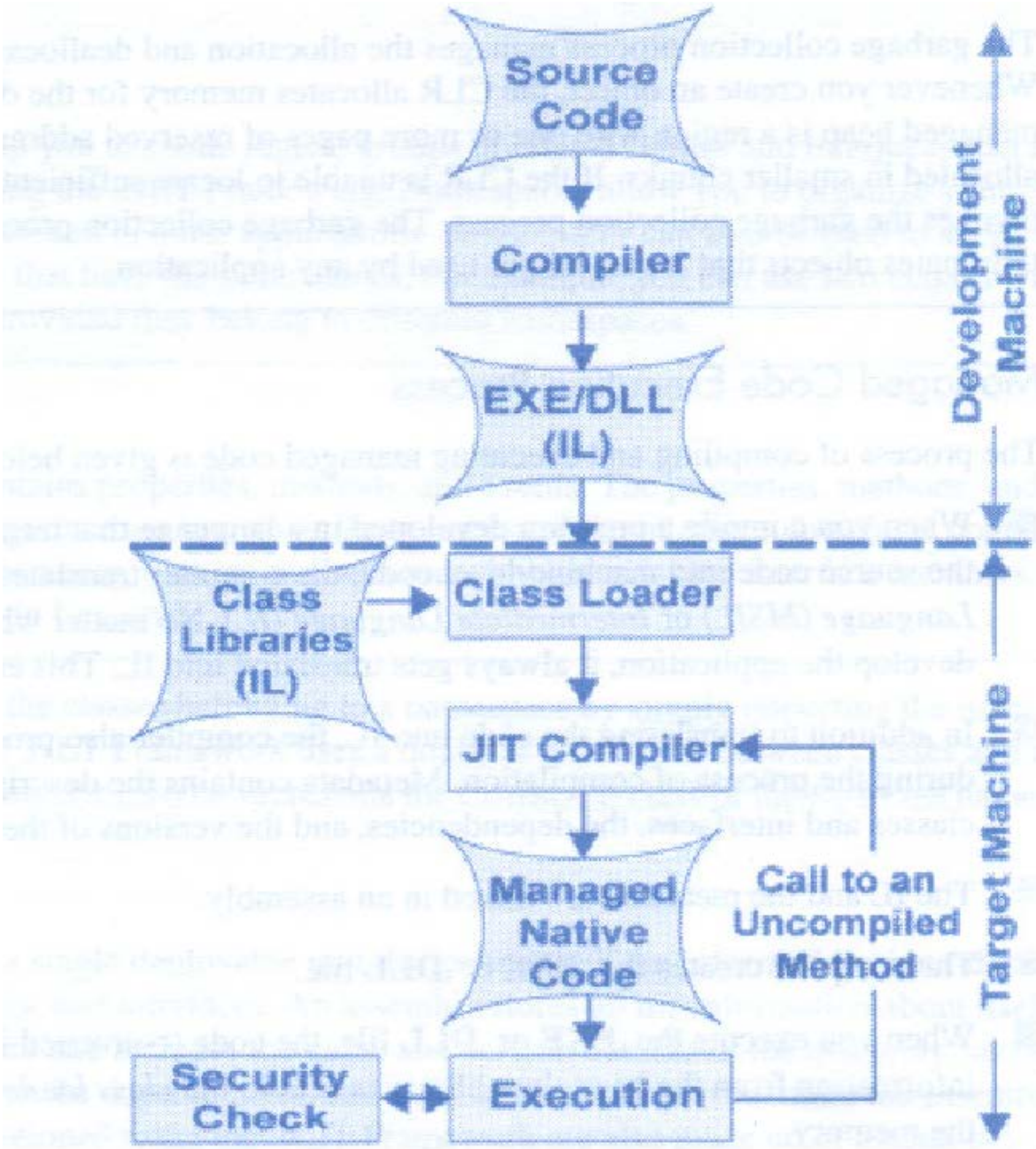
6. Before the code can be executed, the .NET Framework needs to convert the IL into native or CPU-specific code. The Just-In-Time (JIT) compiler translates the code from IL into managed native code. The CLR supplies a JIT compiler for each supported CPU architecture. During the process of compilation, the JIT compiler compiles only the code that is required during execution instead of compiling the complete IL code. When an uncompiled method is invoked during execution, the JIT compiler converts the IL for that method into native code. This process saves the time and memory required to convert the complete IL into native code.

7. During JIT compilation, the code is also checked for type safety. Type safety ensures that objects are always accessed in a compatible way. Therefore, if you try to pass an 8-byte value to a method that accepts a 4-byte value as its parameter, the CLR will detect and trap such an attempt. Type safety also ensures that objects are safely inaccessible from each other and are therefore safe from any unintended or malicious corruption.

8. After translating the IL into native code, the converted code is sent to the .NET runtime manager.

9. The .NET runtime manager executes the code. While executing the code, a security check is performed to ensure that the code has the appropriate permissions for accessing the available resources.

The following diagram depicts the process of compilation and execution of a .NET application:



Types of application interface

1. Character-based: Known as Character User Interface(CUI)
2. Graphics-based: Known as Graphical User Interface(GUI)

Limitations of CUI

1. Input needs to be entered in a particular sequence
2. Input needs to be entered through keyboard
3. Images and pictures cannot be used
4. Interactivity is low

Advantages of GUI

1. Input can be provided through keyboard or mouse
2. Input need not be provided in a sequence
3. Graphics can be used
4. Appearance is attractive, interactive and easy to use

Types of applications that can be developed using .NET framework are

1. **Console based Application** - Creating a Command Line Application and provides a non-graphical text user interface that can be run from the command line. A console application has a character user interface.
2. **Windows based Application** - Creating an application with the GUI interface to the user.

   Limitations of Windows based Applications are:

   - ➔ Cannot be accessed over the web
   - ➔ Need to be installed on each computer on which they have to be executed

3. **Web based Application** –Creating an application using Web pages and can be accessed from any computer that contains a web browser and has access to the internet. Web pages are used to create user interface in web application.

   Advantages:

   - ➔ Can be accessed from anywhere and at anytime
   - ➔ Need not be installed on each computer that accesses these applications
   - ➔ Can be accessed from any computer that contains a web browser and has access to the internet

Web pages can be of the following types:

- ➔ Static web pages – they do not respond dynamically to the actions performed by a user
- ➔ Dynamic web pages - respond dynamically to the actions performed by a user. A web application that contains dynamic web pages is known as a dynamic web application. Example - Google

4. Web Services - Web Services are software components that perform a task in a distributed manner

Advantages of the .NET Framework

1. Consistent programming model
2. Multi-platform applications
3. Multi-language integration
4. Automatic resource management
5. Ease of deployment

1. **Consistent programming model:** The .NET Framework provides a common object-oriented programming model across languages. This object model can be used in code to perform several tasks, such as reading from and writing to files, connecting to databases, and retrieving data.
2. **Multi-platform applications:** There are several versions of Windows most of which run on x86 CPUs. Some versions, such as Windows CE and 64-bit Windows, run on non-x86 CPUs as well. A .NET application can execute on any architecture that is supported by the CLR. In future, a CLR version could even be built for non-Windows platforms.
3. Multi-language integration: .NET allows multiple languages to be integrated. For example, it is possible to create a class in C# that derives from a class implemented in Visual Basic. To enable objects to interact with each other regardless of the language used to develop them, a set of language features has been defined in Common Language Specification (CLS). This specification includes the basic language features required by many applications. The CLS enhances language interoperability. The CLS also establishes certain requirements, which help you to determine whether your managed code conforms to the CLS. Most of the classes defined in the .NET Framework class library are CLS-compliant.
4. Automatic resource management: While creating an application, a programmer may be required to write code for managing resources such as files, memory, network connections, and database resources. If a programmer does not free these resources, the application may not execute properly. The CLR automatically tracks resource usage and relieves a programmer of the task of manual resource management.
5. Ease of deployment: One of the goals of the .NET Framework is to simplify application deployment. .NET applications can be deployed simply by copying files to the target computer. Deployment of components has also been simplified. Till now, Microsoft's Component Object Model (COM) has been used for creating components. COM suffers from various problems relating to deployment. For example, every COM component needs to be registered before it can be used in an application. Moreover, two versions of a component cannot run side-by-side. In such

a case, if you install a new application that installs the newer version of the component, the newly installed application may function normally. However, the existing applications that depends on the earlier version of the component may stop functioning. As against this, the .NET Framework provides zero-impact deployment. Installation of new applications or components does not have an adverse effect on the existing applications. In the .NET Framework, applications are deployed in the form of assemblies. An assembly stores metadata. Therefore, registry entries are not required for storing information about components and applications. In addition, assemblies also store information about the versions of components used by an application. Therefore, the problems relating to versioning are also eliminated in the .NET Framework.

Visual Studio .NET

The Visual Studio .NET integrated development environment (IDE) provides you with a common interface for developing windows and web applications. The IDE provides us with a centralized location for designing the user interface for the application, writing code and compiling and debugging the application.

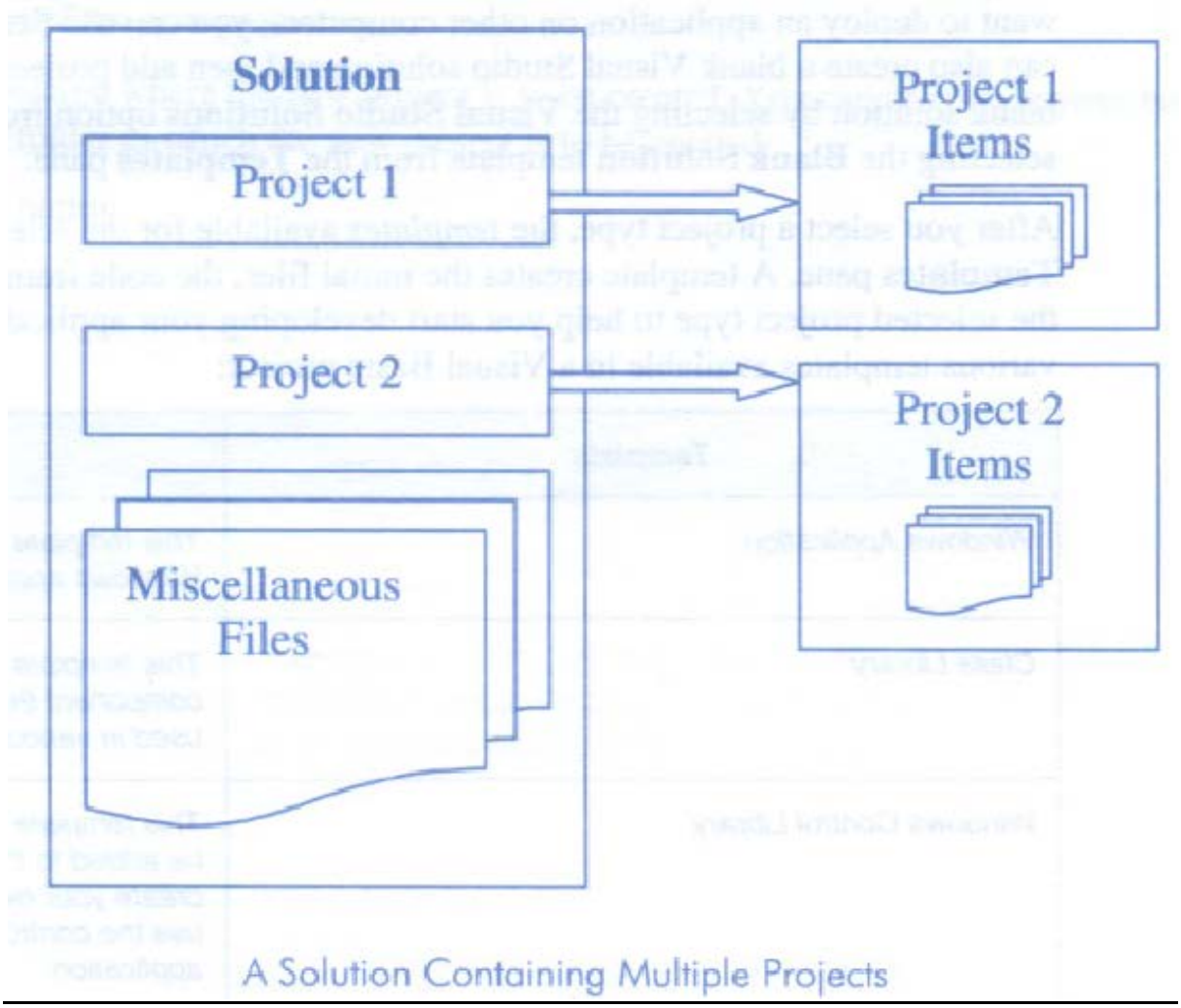
The IDE consists of the following components:

Projects and Solutions

In Visual Studio .NET, an application can be made up of one or more items, such as files and folders. To organize these items efficiently, Visual Studio .NET has provided two types of containers: projects and solutions.

A project typically contains items that are interrelated. For example, you may create a project that contains items for connecting to the Sales database and allowing users to manipulate the data in the database. A project allows you to manage, build, and debug the items that make up an application. When you build a project, it usually results in the creation of an executable file (.exe) or a dynamic link library (.dll). These files that are created as a result of building the project are called the project output.

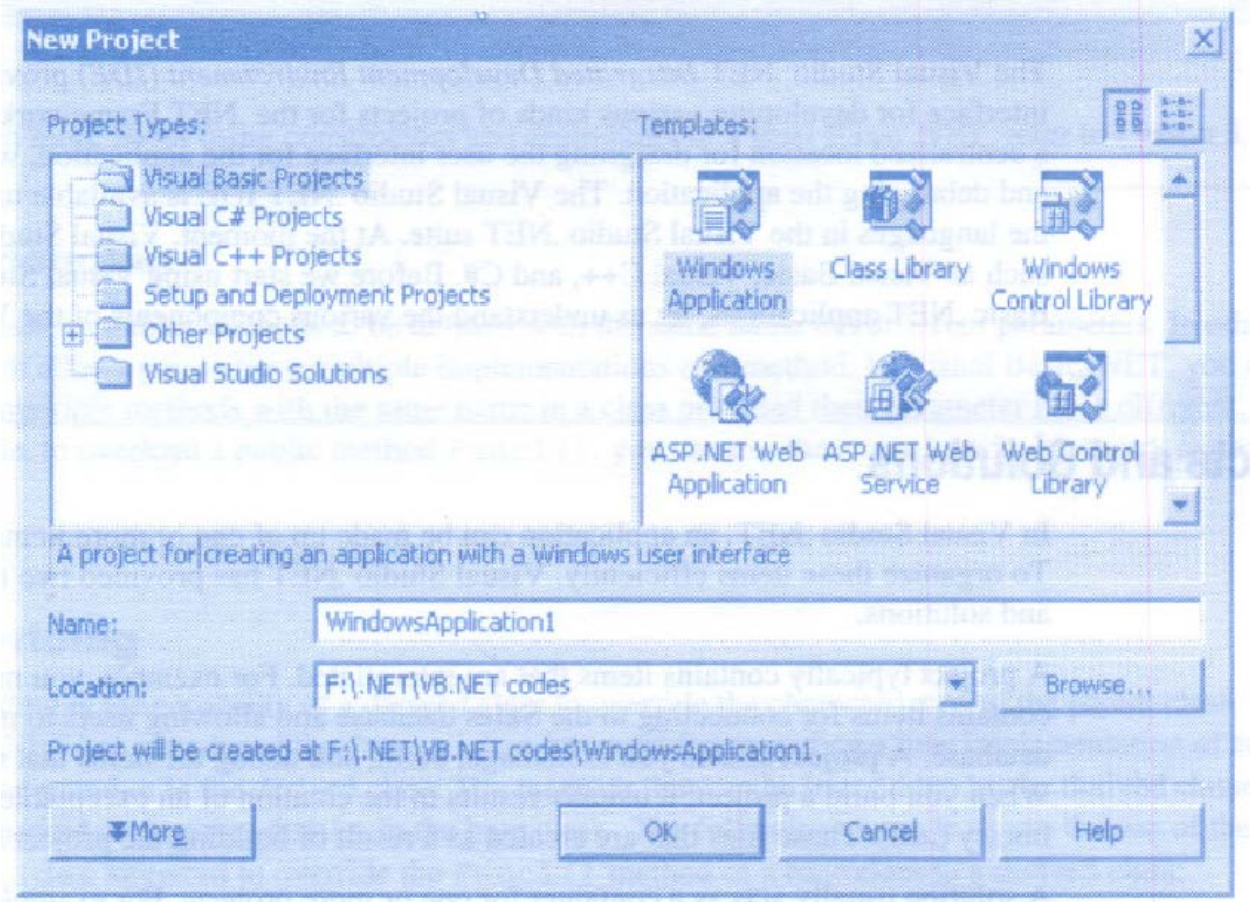
A solution usually acts as a container for one or more projects. For example, you may create a solution containing two projects, one for manipulation of data and the other for generation of reports for the Sales division of an organization. Thus, a solution allows you to work on multiple projects within the same instance of the Visual Studio .NET IDE. A solution also allows you to specify settings and options that apply to multiple projects. The following diagram depicts a solution containing multiple projects:



A Solution Containing Multiple Projects

Creating a Project in Visual Studio .NET

You can create a new project in Visual Studio .NET by clicking the New Project button on the Visual Studio Start Page or by clicking the File -> New -> Project option. When you perform one of these actions, the New Project dialog box is displayed, as shown in the following figure:



The New Project Dialog Box

In the **New Project** dialog box, the **Project Types** pane displays various categories of projects that can be created. You can create a Visual Basic project, a Visual C# project, or a Visual C++ project. If you want to deploy an application on other computers, you can use **Setup and Deployment Projects**. You can also create a blank Visual Studio solution and then add projects to the solution. You can create a blank solution by selecting the **Visual Studio Solutions** option from the Project **Types** pane and then selecting the **Blank Solution** template from the **Templates** pane.

After you select a project type, the templates available for the selected project type are displayed in the **Templates** pane. A template creates the initial files, the code framework, and the property settings for the selected project type to help you start developing your application. The following table lists the various templates available in a Visual Basic project:

Template	Description
Windows Application	This template is used to create the traditional stand-alone Windows application.
Class Library	This template is used to create a class or a reusable component that exposes some functionality that can used in various

	projects.
Windows Control Library	This template is used to create a custom control that can be added to the user interface. For example, you can create your own control to display the current time and use the control while designing the user interface for your application.
ASP.NET Web Application	This template is used to create a Web application. To be able to create an ASP.NET Web application, you must have Internet Information Services (IIS) 5.0 installed on your computer.
ASP.NET Web service	This template is used to create a Web service, which is a component that can be made available to other applications via the Web. A Web service uses standards such as HTTP and XML to communicate with client applications.
Web Control Library	This template is used to create a custom control that can be used in Web applications.
Console Application	This template is used to create a console application that can be run from the command line. A console application has a character user interface.
Windows Service	This template is used to create an application that does not have a user interface. A Windows Service is a program that runs in the background of Windows and can start automatically when the computer is started. For example, you can create a Windows Service that monitors the performance of the system
Empty Project	This template is used to create an empty project. If you want to add any file or component, you need to do so manually.
Empty Web Project	This template is used to create an empty

	Web project.
New Project in Existing Folder	This template is used to add a blank project to an existing application folder. You can create a blank project and then add existing files to the project.

In order to create a Visual Basic Windows application, you need to perform the following steps:

1. In the **New Project** dialog box, select **Visual Basic Projects** from the **Project Types** pane and **Windows Application** from the **Templates** pane.

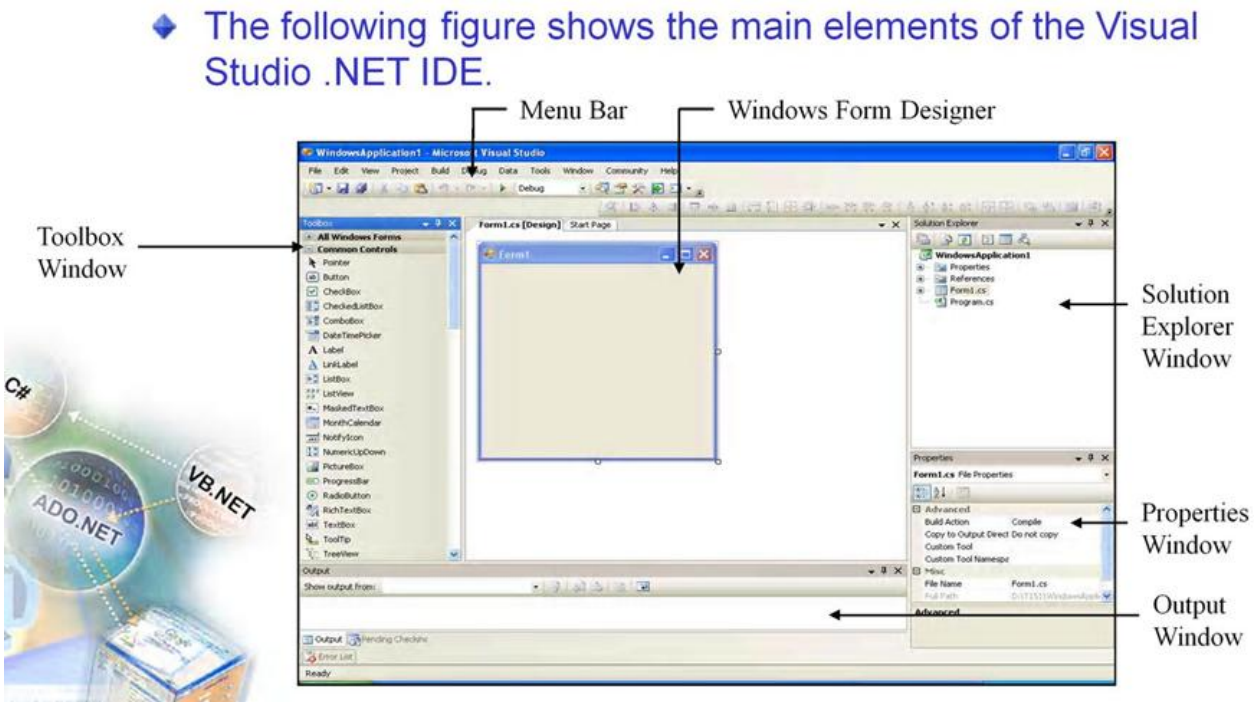
2. Type the name of the new project.

3. Specify the location where the new project is to be created. You can use the **Browse** button to browse to the folder in which the new project is to be created.

4. Click the **OK** button.

User Interface Elements of Visual Studio .NET IDE

When you create a new Windows Application project in Visual Studio .NET, a window similar to the one shown in the following figure is displayed:



The main window contains various standard interface elements found in the Windows environment, such as the menu bar and the toolbar. In addition to these standard interface elements, the Visual Studio .NET IDE contains other elements. These include the Windows Forms Designer, Solution Explorer, Properties window, Toolbox, Output window, Task List window, Server Explorer, Dynamic Help window, Class View window, and the Code and Text Editor window.

1. Windows forms designer: It allows you to design the user interface for an application and add controls to a form, arrange them and add code to perform some action. It provides you with a rapid development solution for our application.

2. Properties Window: It lists the properties associated with each and every control.

3. Solution Explorer window: This displays the solution name, project name, form name, module name and all other files associated with the project.

4. Tool Box: It is provided with all types of controls. They can be easily dragged and dropped in the form. The toolbox has been divided into different categories of controls like Windows forms, data, Components, Containers, Crystal Reports, Menus and Toolbars.

5. Task list window: It displays a list of errors along with the source (the file and the line number) of the error. It helps you identify and locate problems that are detected automatically as you edit or compile code. You can locate the source of a particular error by simply double-clicking the error in the Task List window.

6. Server Explorer window: It is a very useful tool for showing database connectivity and displaying the tables, views and procedures associated with that connection.

7. Dynamic help window: It provides you with context sensitive help.

8. Class view window: It displays the classes, methods and properties associated with a particular file. They are displayed in a hierarchical tree-view depicting the containership of these items.

9. Code and Text editor window: It allows you to enter and edit code and text.

10. The Output Window: When you compile an application, this window displays the current status and after the compilation process is complete, it specifies the number of errors that occurred during compilation.

Programming with C#

C# (pronounced as 'C sharp') is a computer-programming language developed by Microsoft Corporation, USA. C# is a fully object-oriented language like Java and is the first Component-oriented language. It has familiar C, C++ and Java syntax and aims at ease and high productivity of visual basic, the raw power of C++ and the internet platform neutrality of Java. It has been designed to support the key features of .NET Framework, the new development platform of Microsoft for building component-based software solutions. It is a simple, efficient, productive and type-safe language derived from the popular C and C++ languages. It is easy to learn and powerful enough to develop Web-based applications.

Features of C#

1. Simple - C# simplifies C++ by eliminating irksome operators such as ->, :: and pointers. C# treats integer and Boolean data types as two entirely different types. This means that the use of = in place of = = in if statements will be caught by the compiler.

2. Consistent - C# supports an unified type system which eliminates the problem of varying ranges of integer types. All types are treated as objects and developers can extend the type system simply and easily.

3. Modern - C# is called a modem language due to a number of features it supports. It supports.

    - Automatic garbage collection.
    - Rich intrinsic model for error handling
    - Decimal data type for financial applications
    - Modem approach to debugging and
    - Robust security model

4. Object-Oriented - C# is truly object-oriented. It supports all the three tenets of object-oriented systems, namely.

    - Encapsulation
    - Inheritance and
    - Polymorphism

In C#, everything is an object. There are no more global functions, variables and constants.

5. Type-safe - Type-safety promotes robust programs. C# incorporates a number of type-safe measures.

- All dynamically allocated objects and arrays are initialized to zero
- Use of any uninitialized variables produces an error message by the compiler
- Access to arrays are range-checked and warned if it goes out-of-bounds
- C# does not permit unsafe casts
- C# enforces overflow checking in arithmetic operations
- Reference parameters that are passed are type-safe
- C# supports automatic garbage collection

6. Versionable – Making new versions of software modules work with the existing applications is known as versioning. C# provides support for versioning with the help of new and override keywords. With this support, a programmer can guarantee that his new class library will maintain binary compatibility with the existing client applications.
7. Compatible - C# enforces the .NET common language specifications and therefore allows inter-operation with other .NET languages. C# also permits interoperation with C-style APIs.
8. Flexible - Although C# does not support pointers, we may declare certain classes and methods as 'unsafe' and then use pointers to manipulate them. However, these codes will not be type-safe.
9. Inter-operability - C# provides support for using COM objects, no matter what language was used to author them. C# also supports a special feature that enables a program to call out any native API.

A Simple C# Program

```
class SampleOne
{
public static void Main( )
{
System.Console.Writeln(“Welcome to C#”);
}
}
```

Class Declaration

The first line

class SampleOne

declares a class, which is an object-oriented construct. As stated earlier, C# is a true object-oriented language and therefore, everything must be placed inside a class. class is a keyword and declares that a new class definition follows. SampleOne is a C# identifier that specifies the name of the class to be defined.

The Braces

C# is a block-structured language, meaning code blocks are always enclosed by braces { and }. Therefore, every class definition in C# begins with an opening brace '{' and ends with a corresponding closing brace '}' that appears in the last line of the program. This is similar to class constructs of Java and C++.

The Main Method

The third line

Public static void Main( )

defines a method named Main. Every C# executable program must include the Main( ) method in one of the classes. This is the' starting point' for executing the program. A C# application can have any number of classes but only one class can have the Main method to initiate the execution.

This line contains a number of keywords: public, static and void. This is very similar to the main of C++and Java. In contrast to Java and C++, Main has a capital, not lowercase M. The meaning and purpose of these keywords are given below:

public - The keyword public is an access modifier that tells the C# compiler that the Main method is accessible by anyone

static - The keyword static declares that the Main method is a global one and can be called without creating an instance of the class. The compiler stores the address of the method as the entry point and uses this information to begin execution before any objects are created.

void - The keyword void is a type modifier that states that the Main method does not return any value (but simply prints some text to the screen).

The Output Line

The only executable statement in the program is

System.Console.WriteLine("Welcome to C#");

This is output statement and similar to the printf ( ) of C or cout<< of C++. Since C# is a pure object-oriented language, every method should be part of an object. The WriteLine method is a static method of the Console class, which is located in the namespace System. This line prints the string

Welcome to C#

on the screen. The method WriteLine always appends a new-line character to the end of the string. Every C# statement must end with a semicolon.

Executing the Program

We can develop console applications by using:

- a simple DOS editor (or notepad) and a C# compiler (csc.exe), or
- full-pledged development environment by installing Visual tudio.NET 2003 in our system

To build a console application, we require is a simple notepad or MS-DOS editor to write our C# program and a C# compiler (csc.exe) to compile the program. This compiler comes free in the .NET Framework SDK (Standard Development Kit) package.

Once we install the .NET Framework in our system, by default the csc.exe is installed in a folder like: C:\WINDOWS\Microsoft.NET\Framework\vx.y.zzzz, where x, y, zzzz are numbers specific to our system. For instance v1.1.4322, and hence, the entire path to access csc.exe in our system is at: C:\WINDOWS\Microsoft.NET\Framework\ vl.l.4322.

Steps to develop and execute C# console application program:

Step 1: Open start -> All programs -> Accessories -> Command Prompt and click the mouse left button.

Step 2: Change the directory to our working directory, i.e. C:\MyWorkspace as shown in Figure.

Step 3: Set the path of our C# compiler by typing the following at command prompt:

Path = %path %; C:\WINDOWS\Microsoft.NET\Framework\vx.y.zzzz <Enter>

Alternatively, we may set the path of csc.exe in our autoexec.bat file and restart our system.

Note 1: Here we should replace these x, y, and zzzz letters with numbers specific to our C# compiler version. In ours, it is v1.1.4322

Note 2: The last word <Enter> means press Enter button on the keyboard.

Step 4: At the DOS prompt, run a DOS editor command

C:\MyWorkspace edit <Enter>

Step 5: Now write our program code

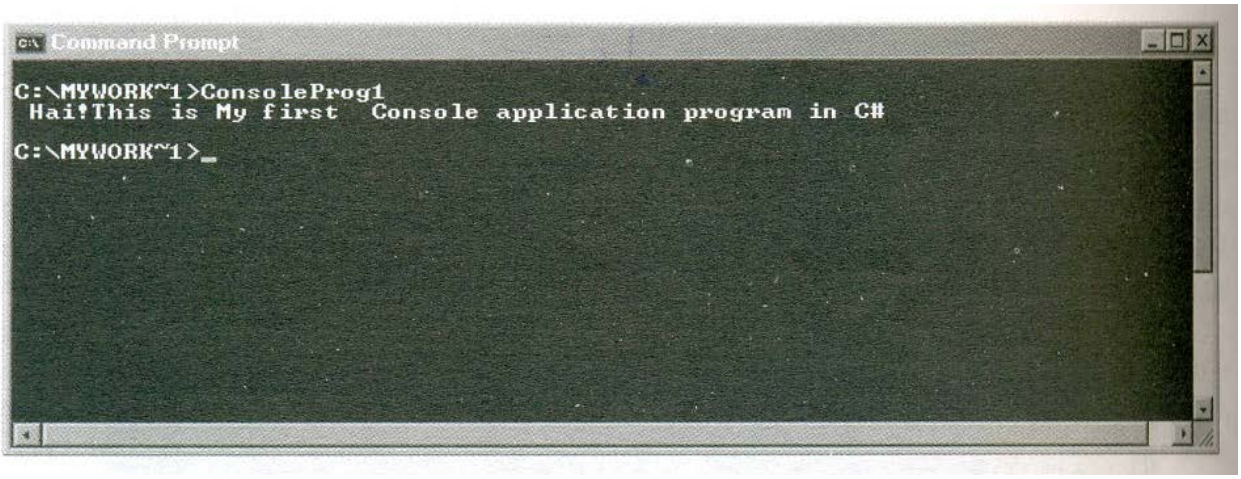
Step 6: Save this program in our MyWorkspace folder as ConsoleProg1.cs. Now, exit the editor and come back to DOS prompt.

Step 7: Type csc ConsoleApp1.cs <Enter>

Step 8: The C# compiler compiles ConsoleAppl.cs as shown in Figure.

Step 9: To run console application, type ConsoleProg1 <Enter>

The output of the application 'This is my first Console application program in C#' is as shown in Figure

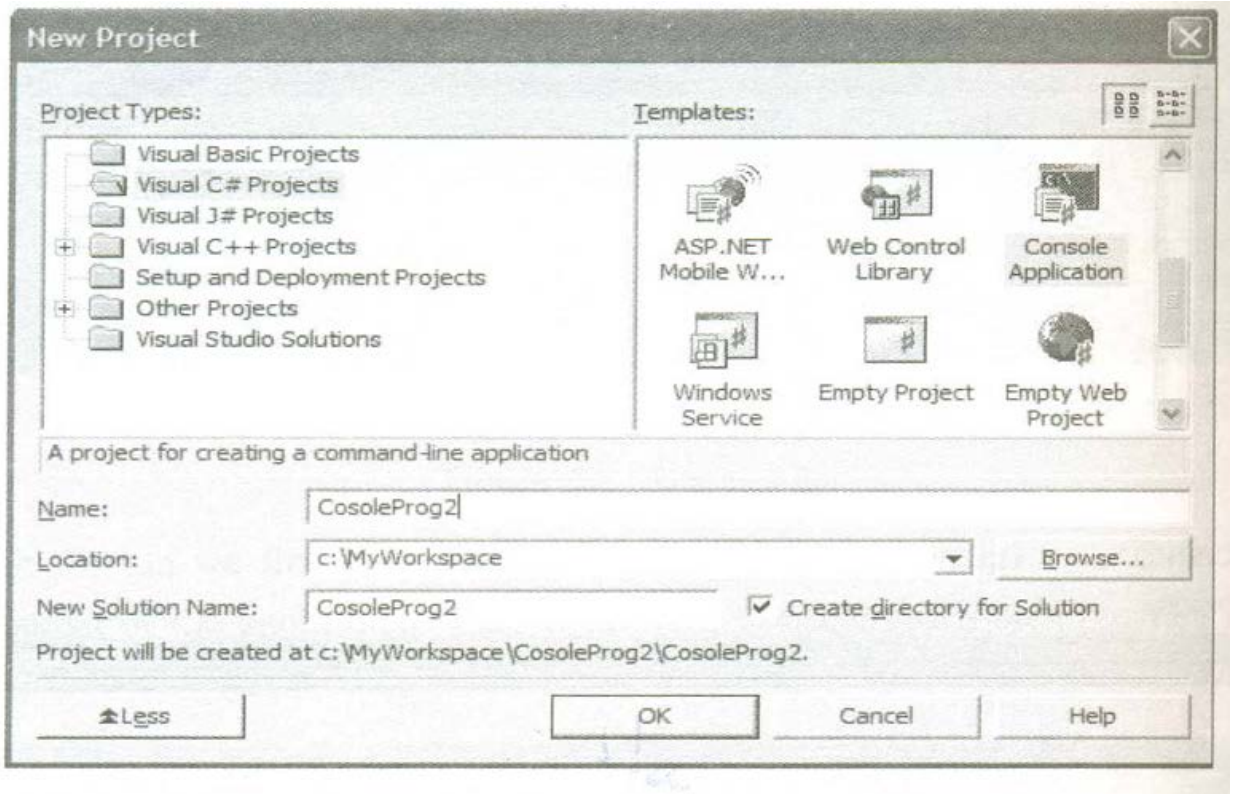


Console Application Using AppWizard

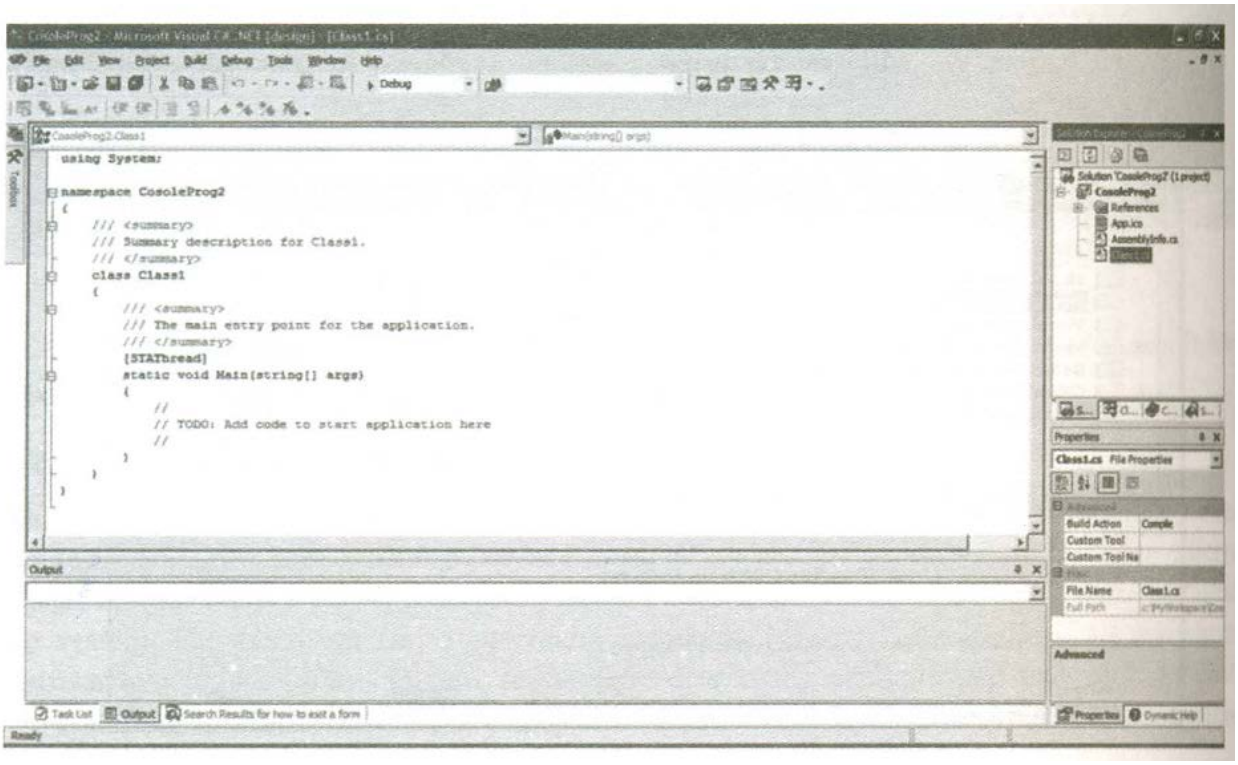
Step 1: Open start! All Programs -> Microsoft Visual Studio.NET 2003 -> Microsoft VisualStudio.NET 2003 and click the mouse left button. The MDE (Microsoft Development Environment) opens up.

Step 2: Next, we will go to File -> New -> Project on the MDE screen and click mouse left button. We will see the project options

Step 3: Now do the following settings. From the left pane, under Project types, select VisualC# Projects, and from the right pane, under Templates, select Console Application and at the lower pane, type Name: as ConsoleProg2 and type the Location: as C:\MyWorkspace. The resulting screenshot for these settings is given in Figure



Step 4: Now click OK button. A template for our ConsoleProg2 project appears as shown in Figure



Step 5: Now all that we need to do is filling in with the code that we desire to run. Insert the following code

Console.Write ("Hai");

Console.WriteLine ("This is My First Console Application Program in C#");

Step 6: Now open Build options and click 'Build Solution' or alternatively we can press ctrl, shift, and B buttons on keyboard simultaneously. If the compilation is successful, we will see the following message in the lower pane of the screen.

Build: 1 Succeeded, 0 failed, 0 skipped

Step 7: To run this program, either press ctrl button plus function button F5 simultaneously or open the Debug options, and click on 'Start Without Debugging' option.

Namespaces

Let us consider the output statement

System.Console.WriteLine ( );

In this, we noted that System is the namespace (scope) in which the Console class is located. A class in a namespace can be accessed using the dot operator. Namespaces are the way C# to create logical groups of related classes and interfaces.

C# supports a feature known as 'using' directive that can be used to import the namespace System into the program. Once a namespace is imported, we can use the elements of that namespace without using the namespace as prefix.

A program using namespaces and comments

```
/*
This program uses namespaces and comment lines
*/
using System; // System is a namspace
class SampleTwo
{
    // Main method begins
    public static void Main( )
    {
        Console.WriteLine("Hello!");
    }
    // Main method ends
}
```

Note that the first statement in the program is

using System;

This tells the compiler to look in the System library for unresolved class names. Note that we have not used the System prefix to the Console class in the output line.

When the compiler parses the Console.WriteLine method, it will understand that the method is undefined. However, it will then search through the namespaces specified in using directives and, upon finding the method in the System namespace, will compile the code without any complaint.

Adding Comments

Comments play a very important role in the maintenance of programs. They are used to enhance readability and understanding of code. C# permits two types of comments, namely,

- Single-line comments
- Multiline comments

Single-line comments begin with a double backs lash (//) symbol and terminate at the end of the line. Everything after the lion a line is considered a comment.

If we want to use multiple lines for a comment, we must use the second type known as multi-line comment. This comment starts with the /*characters and terminates with */ as shown in the beginning of the program.

Main returning a value

Main( ) can also return a value if it is declared as int type instead of void. When the return type is int, we must include a return statement at the end of the method

Main returning a value

```
// Main returning a value
using System;
class SampleThree
{
     public static int Main( )
     {
          Console.WriteLine ("Hello!");
          return 0;      // Return statement
     }
}
```

Using aliases for namespace classes

We can avoid the prefix System to the Console class by implementing the using System; statement.

System is a namespace and Console is a class. The using directive can be applied only to namespaces and cannot be applied to classes. Therefore the statement

Using System.Console;

is illegal. However, we can overcome this problem by using aliases for namespace classes. This takes the form

using alias-name = class-name;

Program illustrates the use of aliases for classes

Program 3.4

Use of aliases for classes

```
using A = System.Console; //A is alias for System.Console
class SampleFour
{
    public static void Main( )
    {
        A.WriteLine("Hello!");
    }
}
```

Passing string objects to writeline method

So far, we have seen only constant string output to the Console. We can store string values in string objects and use these objects as parameters to the WriteLine method. We can use string data type to create a string variable and assign a string constant to it as below:

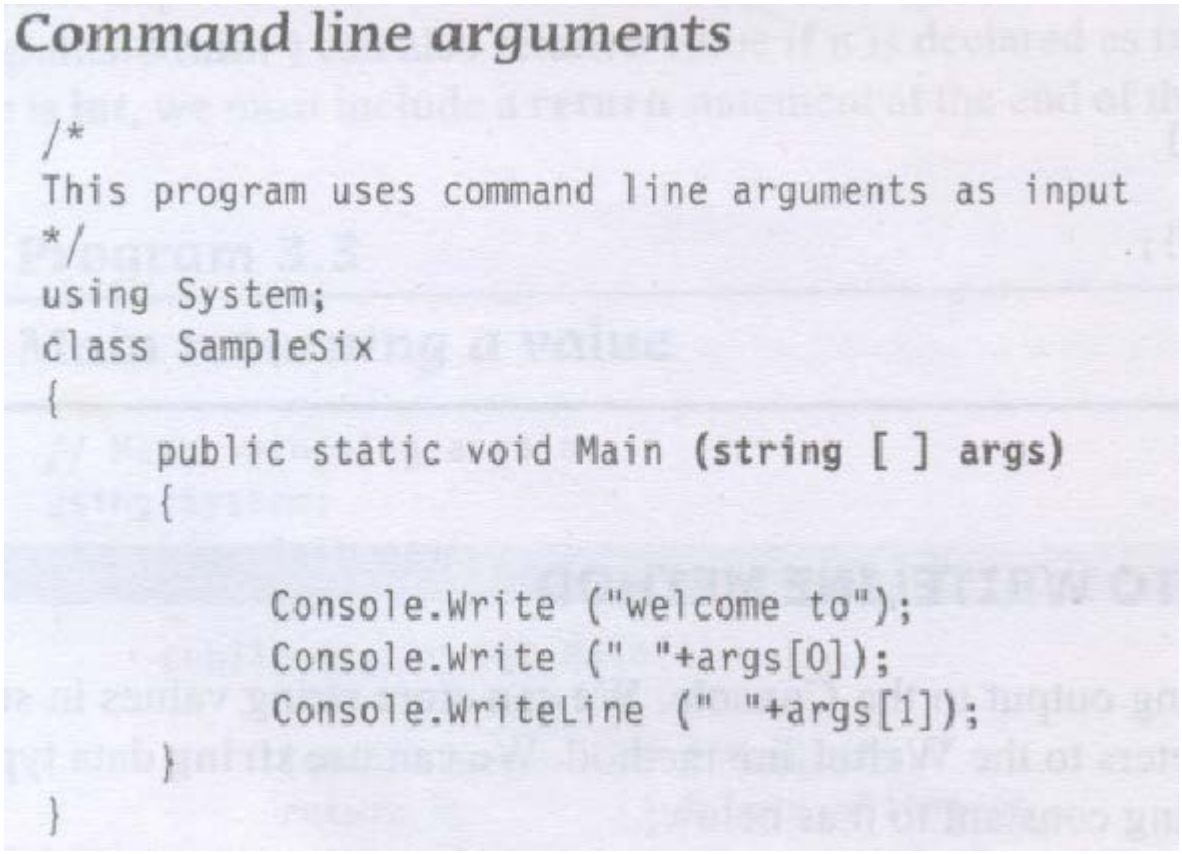
string s = "abc" ;

The content of s may be printed out using the WriteLine method.

System.Console.WriteLine(s); //s is string object

Command line arguments

Command line arguments are used to provide input at the time of execution. They are the parameters supplied to the Main method at the time invoking it for execution.



Command line arguments

```
/*
This program uses command line arguments as input
*/
using System;
class SampleSix
{
    public static void Main (string [ ] args)
    {

        Console.Write ("welcome to");
        Console.Write (" "+args[0]);
        Console.WriteLine (" "+args[1]);
    }
}
```

Main is declared with a parameter args. The parameter args is declared as an array of strings (known as string objects). Any arguments provided in the command line (at the time of execution) are passed to the array args as its elements. We can access the array elements by using a subscript like args[0], args[1] and so on.

For example, consider the command line

Samp1eSi x C sharp

This command line contains two arguments which are assigned to the array args as follows:

C ------→ args [0]

sharp --------→ args[1]

Difference between the Write and WriteLine ( ) method is that Write () does not cause a line break and therefore the next output statement is printed on the same line.

Main with a class

A program with two classes

```
class TestClass    // class definition
{
    public void fun( )
    {
        System.Console.WriteLine("C# is modern");
    }
}

class SampleSeven
{
    public static void Main( )
    {
        TestClass test = new TestClass( );  // creating test object
        test.fun( );  // calling fun ( ) function
    }
}
```

This program has two class declarations, one for the TestClass and another for the Main method. TestClass contains only one method to print a string "C# is modern". The Main method SampleSeven class creates an object of TestClass and uses it to invoke the method fun() contained in TestClass as follows:

TestClass test = new TestClass ( );

test. fun ( );

The object test is used to invoke the method fun ( ) of TestClass with the help of the dot operator. Execution of Program will produce the following output:

C# is modern

Providing interactive input

So far we have seen two approaches for giving values to string objects:

- Using an assignment statement
- Through command line arguments.

It is also possible to give values to string variables interactively through the keyboard at the time of execution.

Interactive console input

```
using System;
class SampleEight
{
     public static void Main( )
     {
          Console.Write("Enter your name: ");
          string name = Console.ReadLine( );
          Console.WriteLine("Hello " + name);
     }
}
```

The method

Console.ReadLine( );

causes the execution to wait for the user to enter his name. The moment the user types his name and presses the 'Enter' key, the input is read into the string variable name.

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

Example

n= Convert.ToInt32(Console.ReadLine( ));

USING MATHEMATICAL METHOD

Using mathematical functions

```
using System;
class SampleNine
{
    public static void Main( )
    {

        double x = 5.0; //Declaration and initialization
        double y;       // Simple declaration
        y = Math.Sqrt(x);
        Console.WriteLine ( " y = " + y) ;
    }
}
```

The statement

y =Math.Sqrt(x);

invokes the method Sqrt( ) of Math class which is a part of System namespace.

Multiple main methods

C# enables us to define more than one class with the Main method. Since Main is the entry point for program execution, there are now more than one entry points. In fact, there should be only one. This problem can be resolved by specifying which Main is to be used to the compiler at the time of compilation as shown below:

csc filename.cs / main:classname

Filename is the name of the file where the code is stored and class name is the name of the class containing the Main which we would like to be the entry point.

Program shows a simple program that has two classes containing Main methods. We may save it in a file called multimain.cs and compile it using the switch

/main:ClassA or /main:ClassB

as required.

Application with multiple Main methods

```
using System;
class ClassA
{
     public static void Main( )
     {
          Console.WriteLine("Class A");
     }
}

class ClassB
{
     public static void Main( )
     {
          Console.WriteLine("Class B");
     }
}
```

Compile time errors

A program may contain two types of errors:

- Syntax errors
- Logic errors

While syntax errors will be caught by the compiler, logic errors should be eliminated by testing the program logic carefully.

Fixing syntax errors

```
// Program with syntax errors
using Systom;  // Error here
class SampleTen
{

    public static void main( )//Error here
    {
        Console.WriteLine("Hello, Errors")      //Error here
    }
}
```

When above program is compiled, will display the following output:

Errors.cs(2.7): error cs0234: The type or namespace name 'System' does not exist in the class or namespace

The compiler could not locate a namespace named 'System' and therefore produces an error message and then stops compiling. The error message contains the following information:

- Name of the file being compiled (Errors.cs)
- Line number and column position of the error (2.7)
- Error code as defined by the compiler (CS0234)
- Short description of the error

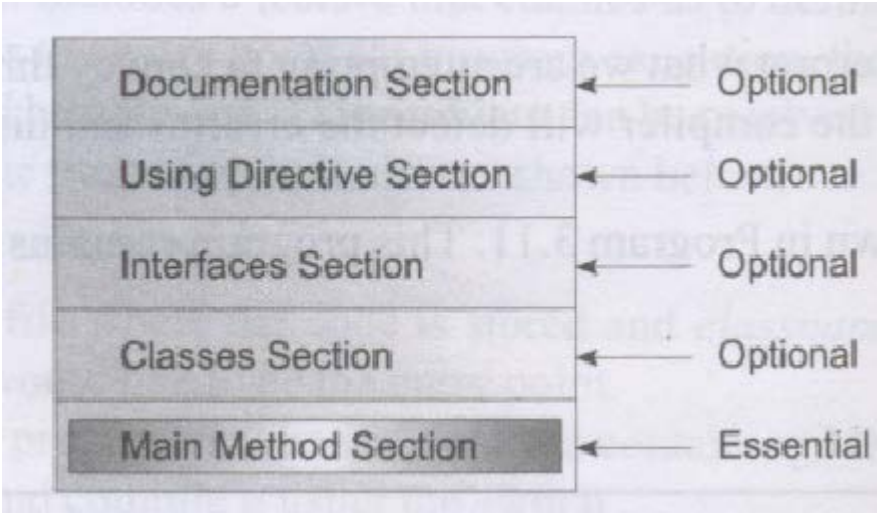
Here, the error is that System has been spelled wrongly. Once we correct this, the compiler will report the details of other errors in the same format.

Program structure

An executable C# program may contain a number coding blocks as shown in Figure. The documentation section consists of a set of comments giving the name of the program, the author, date and other details, which the programmer (or other users) may like to use at a later stage. Comments must explain the

- Why and what of classes, and the
- How of algorithms

This would greatly help maintaining the program.



The using directive section will include all those namespaces that contain classes required by the application. using directives tell the compiler to look in the namespace specified for these unresolved classes.

An interface is similar to a class but contains only abstract members. Interfaces are used when we want to implement the concept of multiple inheritance in a program.

A C# program may contain multiple class definitions. Classes are the primary and essential elements of a C# program. These classes are used to map the objects of real-world problems. The number of classes depends on the complexity of the problem.

Since every C# application program requires a Main method as its starting point, the class containing the Main is the essential part of the program. A simple C# program may contain only this part. The Main method creates objects of various classes and establishes communications between them. On reaching the end of Main, the program terminates and the control passes back to the operating system.

Literals, Variables and Data Types

A C# program is a collection of tokens, comments and white spaces. The smallest individual units in a program are known as tokens. C# includes the following five types of tokens:

• Identifiers

• Keywords

• Literals

• Operators

• Punctuators

Identifiers are used for naming variables, classes, methods, labels, namespaces, interfaces, etc. C# identifiers enforce the following rules:

• They can have alphabets, digits and underscore characters

• They must not begin with a digit

• Upper case and lower case letters are distinct

• Keywords in stand-alone mode cannot be used as identifiers

C# permits the use of keywords as identifiers when they are prefixed with the '@' character

Keywords are explicitly reserved identifiers and cannot be used as variable names except when they are prefaced by the @ character. Table lists all the C# keywords.

abstract	event	namespace	static
as	explicit	new	string
base	extern	null	struct
bool	false	object	switch
break	finally	operator	this
byte	fixed	out	throw
case	float	override	true
catch	for	params	try
char	foreach	private	typeof
checked	get	protected	unit
class	goto	public	ulong
const	if	readonly	unchecked
continue	implicit	ref	unsafe
decimal	in	return	ushort
default	int	sbyte	using
delegate	interface	sealed	value
do	internal	set	virtual
double	is	short	void
else	lock	sizeof	while
enum	long	stackable	

Literals are value constants assigned to variables in a program. C# supports types of literals

An **Integer literal** refers to a sequence of digits. There are two types of integers, namely, decimal integers and hexadecimal integers.

Real literals are numbers containing fractions. Such numbers are called real (or floating point) numbers.

Boolean Literals

There are two Boolean literal valuws

• true

• false

They are used as values of relational expressions.

Single Character Literals

A single-character literal (or simply character constant) contains a single character enclosed within a pair of single quote marks.

String literals

A string literal is a sequence of characters enclosed between double quotes. The characters may be alphabets, digits, special characters and blank spaces.

Backslash Character literal

C# supports some special backslash character constants that are used in output methods. For example, the symbol \n stands for a new-line character. A list of such backs lash character literals is given below

Table 4.2 Backslash character literals

Constant	*Meaning*
'\a'	alert
'\b'	back space
'\f'	form feed
'\n'	new-line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\''	single quote
'\"'	double quote
'\\'	backslash
'\0'	null

Operators are symbols used in expressions to describe operations involving one or more operands.

C# operators can be classified as below:

1. Arithmetic operators

2. Relational operators

3. Logical operators

4. Assignment operators

5. Increment and decrement operators

6. Conditional operators.

7. Bitwise operators

8. Special operators

Arithmetic operators

These operators are used to perform arithmetic operations.

Operator	*Meaning*
+	Addition or unary plus
–	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

Relational operators

These are used to compare two quantities and depending on their relation, take certain decisions.

Operator	*Meaning*
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

Logical operators

These are used when we want to form compound conditions by combining relations.

Operator	*Meaning*
&&	logical AND
||	logical OR
!	logical NOT
&	bitwise logical AND
|	bitwise logical OR
^	bitwise logical exclusive OR

Assignment operators

This is used to assign the value of an expression to a variable. The assignment operator is '='.

Increment and decrement operators

The operator ++ adds I to the operand while - - subtracts 1. Both are unary operators

Conditional operators

The character pair ? : is a ternary operator available in C#. This operator is used to construct conditional expressions of the form

exp1 ? exp2 : exp3

where exp1, exp2 and exp3 are expressions.

Bitwise operators

C# supports operators that may be used for manipulation of data at bit level. These operators may be used for testing the bits or shifting them to the right or left. Bitwise operators may not be applied for floating-point data.

Operator	*Meaning*
&	bitwise logical AND
|	bitwise logical OR
^	bitwise logical XOR
~	one's complement
<<	shift left
>>	shift right

Special operators

C# supports the following special operators.

is	(relational operator)
as	(relational operator)
typeof	(type operator)
sizeof	(size operator)
new	(object creator)
.(dot)	(member-access operator)

Operator precedence and associativity

Each operator in C# has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. The operators at the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from left to right or from right to left, depending on the level. This is known as the associativity property of an operator.

Operator Symbol	*Associative*	*Description*	*Precedence*
++	Left to right	Postfix increment	1
—	Left to right	Postfix decrement	
( )	Left to right	Function call	
[ ]	Left to right	Array indexing	
.	Left to right	Class or structure member access operator	
new	Right to left	Object creation	
sizeof ( )	Right to left	Size information (in bytes)	
typeof	Right to left	Type information	
checked, unchecked	Right to left	Overflow exception control	
++	Right to left	Prefix increment	2
—	Right to left	Prefix decrement	
!	Right to left	Logical NOT	
~	Right to left	Bitwise NOT	
-	Right to left	Unary minus	
+	Right to left	Unary plus	

Operator Symbol	*Associative*	*Description*	*Precedence*
*	Left to right	Multiply	3
/	Left to right	Divide	
%	Left to right	Modulus	
+	Left to right	Add	4
–	Left to right	Subtract	
>>	Left to right	Right shift	5
<<	Left to right	Left shift	
<=	Left to right	Less than or equal	6
>=	Left to right	Greater than or equal	
>	Left to right	Greater than	
<	Left to right	Less than	
is	Left to right	Type information	
= =	Left to right	Equal	7
!=	Left to right	Not equal	
&	Right to left	Address	8
^	Left to right	Bitwise exclusive OR	9
|	Left to right	Bitwise OR	10
&&	Left to right	Logical AND	11
||	Left to right	Logical OR	12
?:	Right to left	Conditional	13
*=, /=, %=, +=, –=, <<=, >>=, &=, ^=, |== =	Right to left	Compound assignments	14
,	Left to right	Comma	15

Punctuators are symbols used for grouping and separating code. Punctuators (also known as separators) in C# include:

• Parentheses ( )

• Braces { }

• Brackets [ ]

• Semicolon;

• Colon:

• Comma,

• Period.

Variables

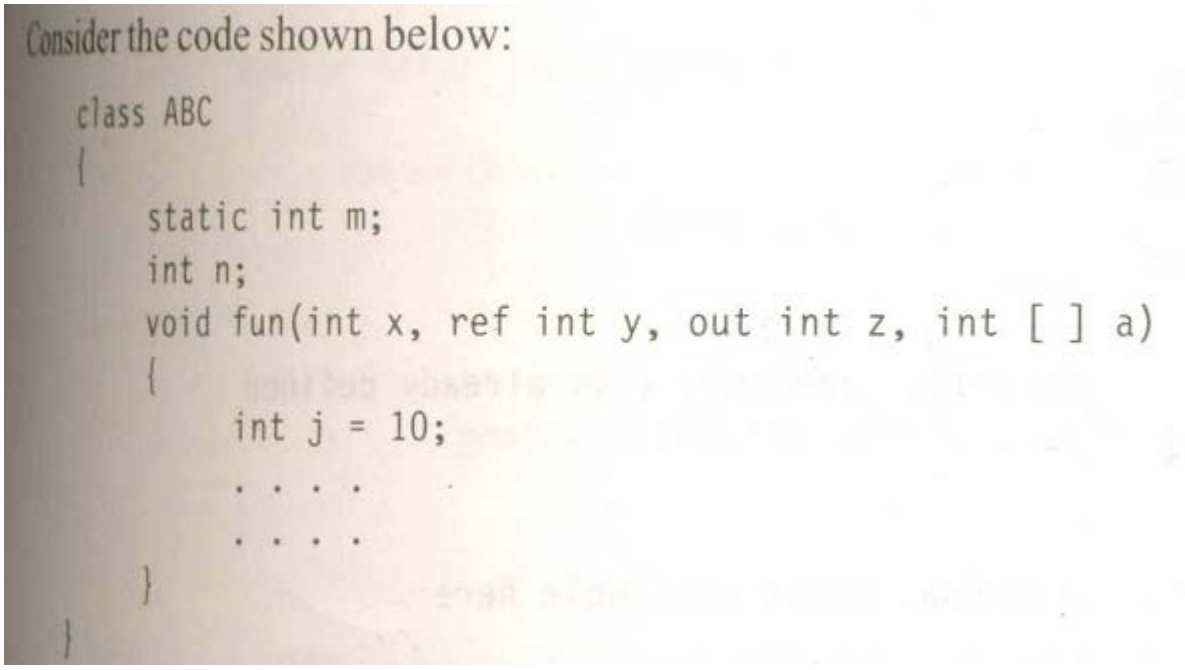
A variable is an identifier that denotes a storage location used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during the execution of the program. Every variable has a type that determines what values can be stored in the variable.

Scope of variables

The scope of a variable is the region of code within which the variable can be accessed. This depends on the type of the variable and place of its declaration. C# defines several categories of variables. They include:

- Static variables
- Instance variables
- Array variables
- Value variables
- Reference variables
- Output variables
- Local variables

Consider the code shown below:



```
class ABC
{
    static int m;
    int n;
    void fun(int x, ref int y, out int z, int [ ] a)
    {
        int j = 10;
        . . . .
        . . . .
    }
}
```

This code contains the following variables:

- m as a static variable
- n as an instance variable
- x as a value parameter
- y as a reference parameter
- z as an output parameter
- a[0] as an array element
- j as a local variable

Static and instance variables are declared at the class level and are known as fields or field variables. The scope of these variables begins at the place of their declaration and ends when the Main method terminates.

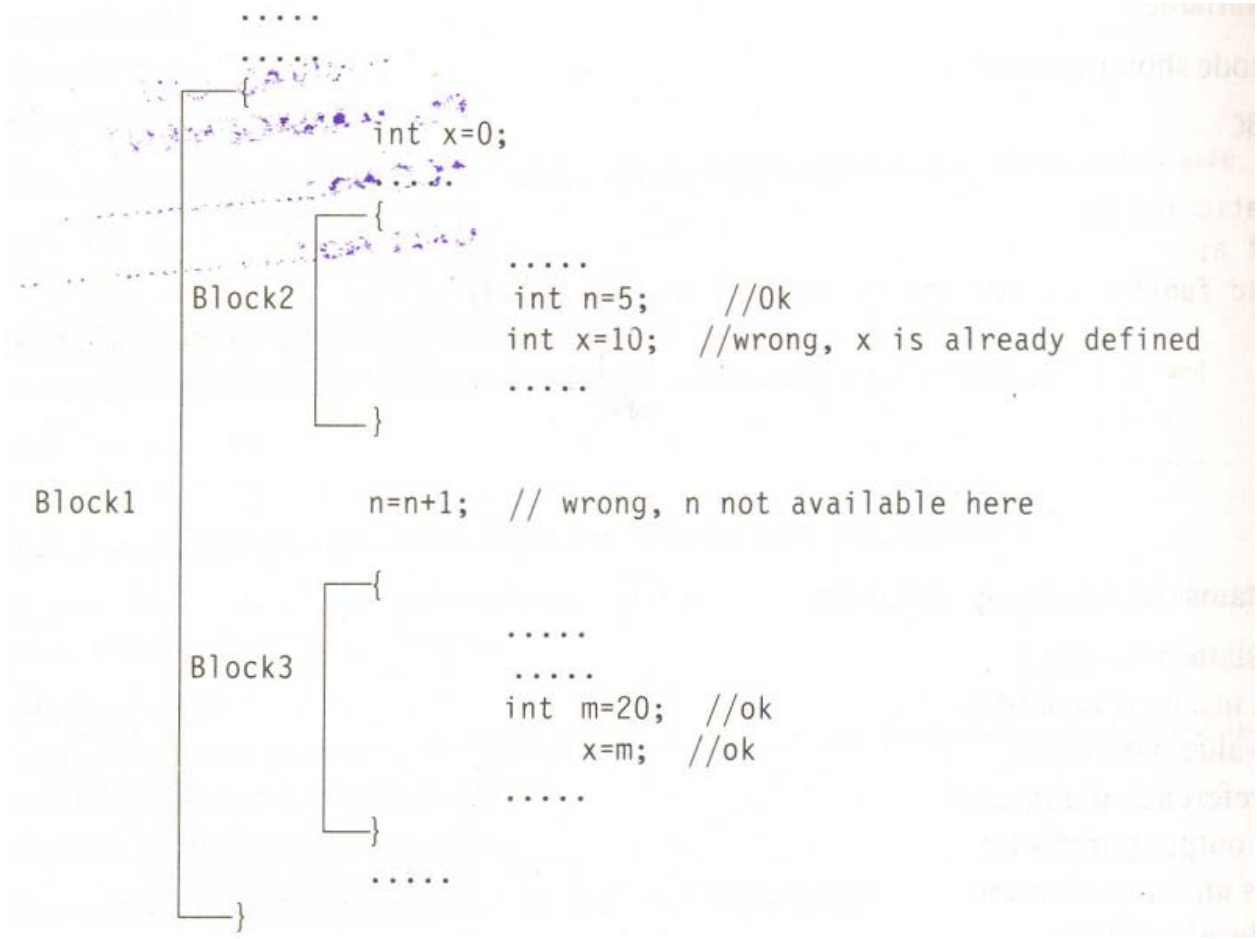
The variables x, y and z are parameters of the method fun( ). The value parameter x will exist till the end of the method. The reference and output parameters (y and z) do not create new storage locations. Instead, they represent the same storage locations as the variables that are passed as arguments. The scope of these variables is always the same as the underlying-variables.

The elements of an array such as a[0] come into existence when an array instance is created, and cease to exist when there are no references to that array instance. Arrays declared at class level behave like fields.

Variables declared inside methods are called local variables. They are called so because they are not available for use outside the method definition. Local variables can also be

declared inside program blocks that are defined between an opening brace { and a closing brace}. The scope of a local variable starts immediately after its identifier in the declaration and extends up to the end of the block containing the declaration. Within the scope of a local variable, it is an error to declare another local variable with the same name.

Consider a code segment containing nested blocks as shown below. Each block can contain its own set of local variable declarations. We cannot, however, declare a variable to have the same name as one in the outer block. The variable x declared in Blockl is available in all the three blocks. However, the variable n declared in Block2 is available only in Block2 because it goes out of scope at the end of block2. Similarly, the variable m is accessible only in Block3. Note that we cannot declare the variable name x again in either Block2 or Block3 .



Data Types

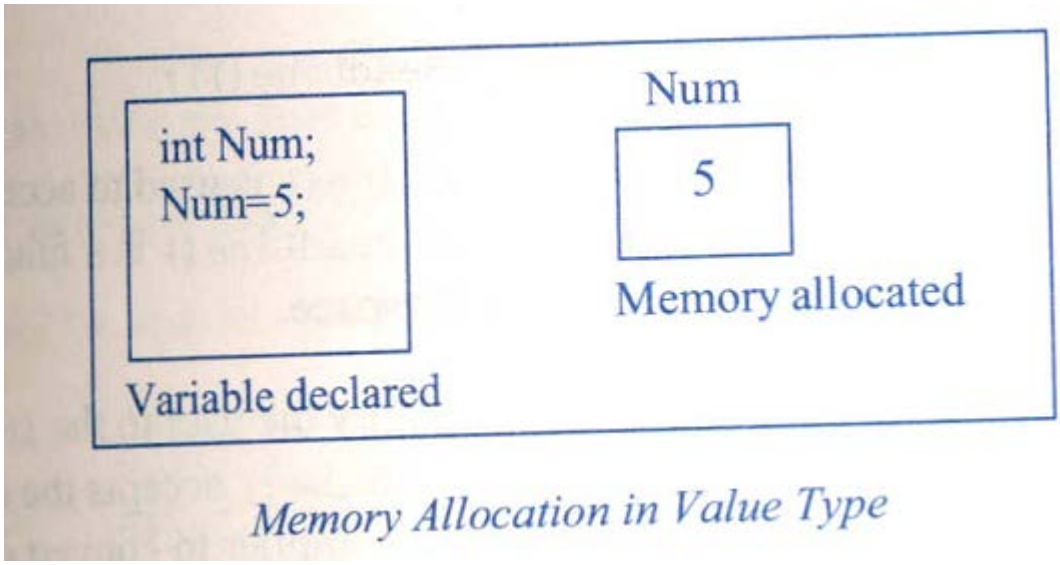
Every variable in C# is associated with a data type. Data types specify the size and type of values that can be stored.

The types in C# are primarily divided into two categories:

• **Value types:** They directly contain data. Some examples of the value types are char, int, and float, which can be used for storing alphabets, integers, and floating point numbers,

respectively. When you declare an int variable, the system allocates memory to store the value.

The following figure shows the memory allocation of an int variable.

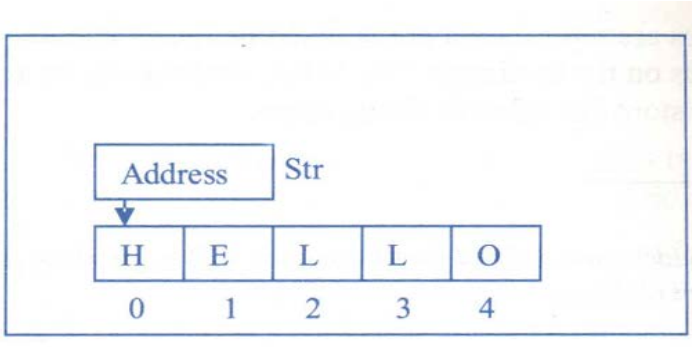


Memory Allocation in Value Type

• **Reference types Reference types:** The reference types do not maintain data but they contain a reference or pointer to the data, which are stored in memory. Variables of reference type are always allocated on heap memory. A heap consists of memory available to the program at runtime. The allotment is done automatically during the execution time

Using more than one variable, you can use the reference types to refer to a memory location. This means that if the value in the memory location is modified by one of the variables, the other variables automatically reflect the changed value. The example of a reference is string data type.

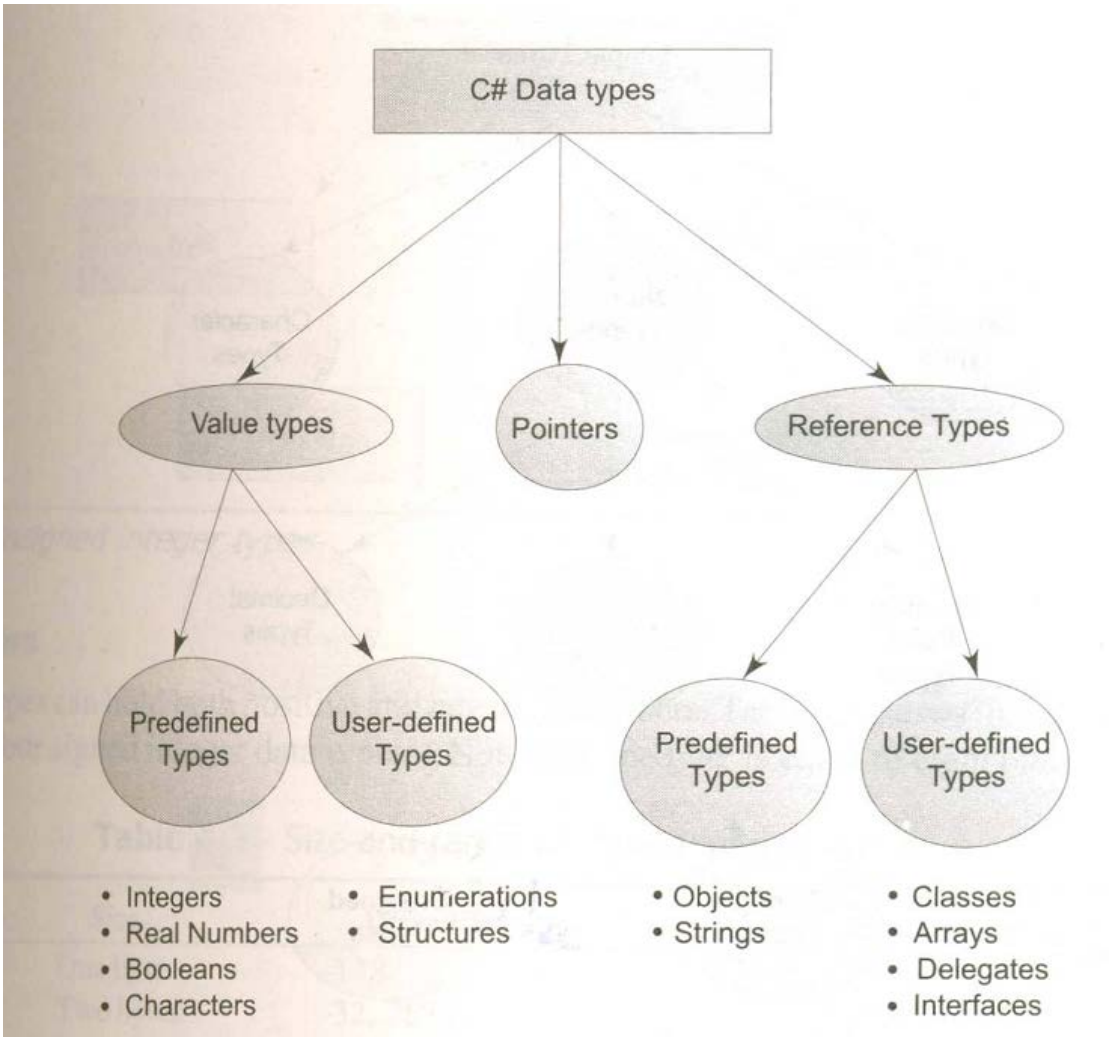
The following figure shows the memory allocation of a string value "HELLO" in a variable named Str.



Memory Allocation of the String Type Variable

In the preceding figure, the value of the variable `Str` is stored in memory and the address of this memory is stored at another location.

Value types and reference types are further classified as predefined and user-defined types as shown below



Value types

The value types of C# can be grouped into two namely,

• User-defined types and

• Pre defined types (or simple types)

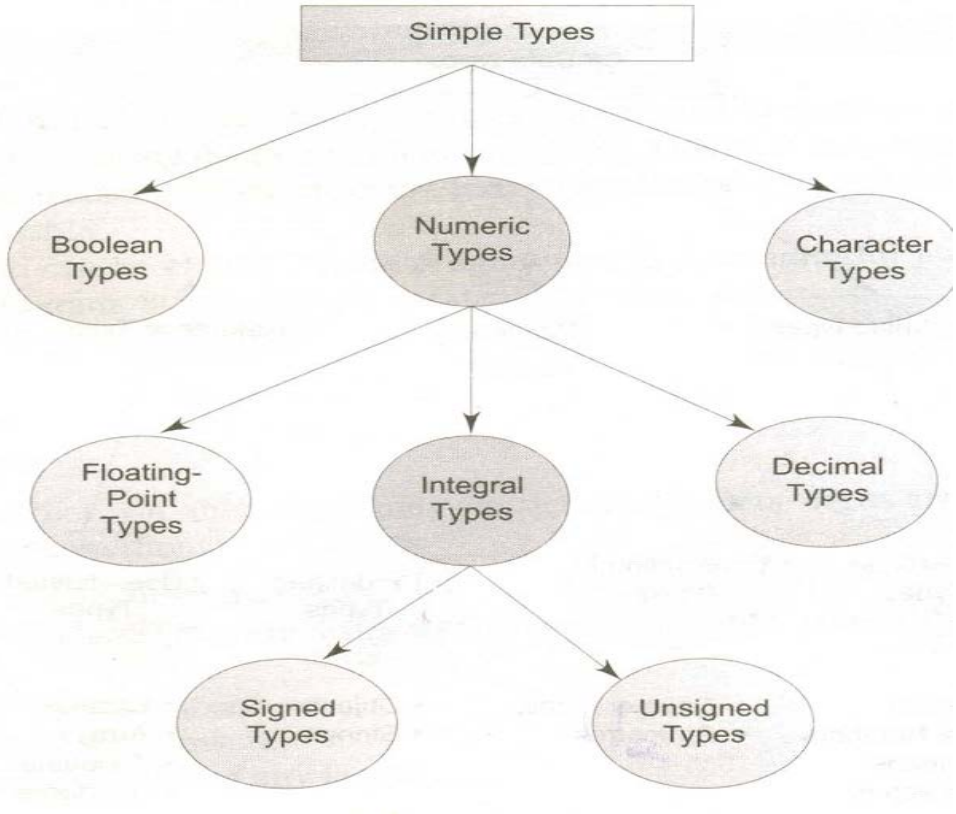
Predefined value types which are also known as simple types (or primitive types) are further subdivided into

• Numeric types

• Boolean types, and

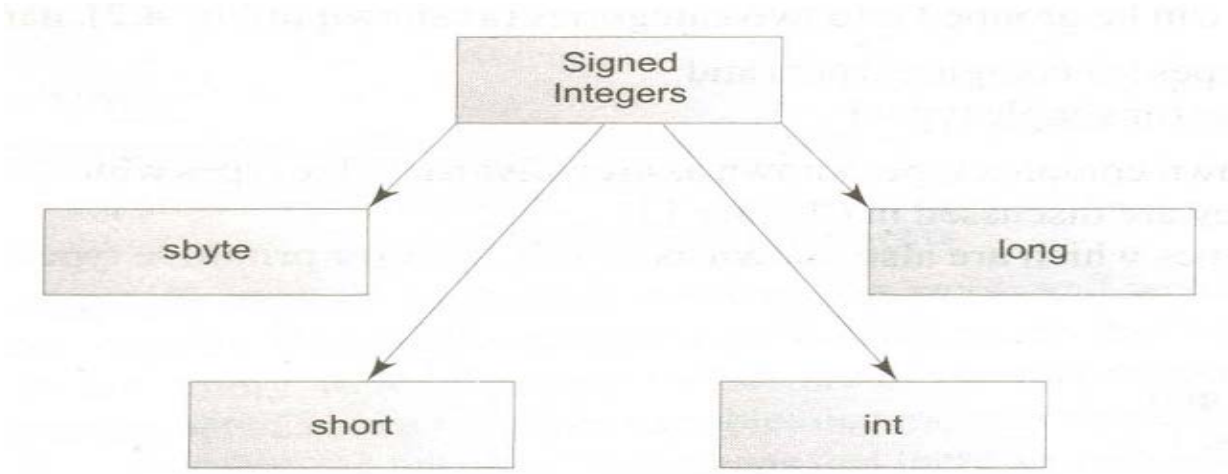
• Character types.

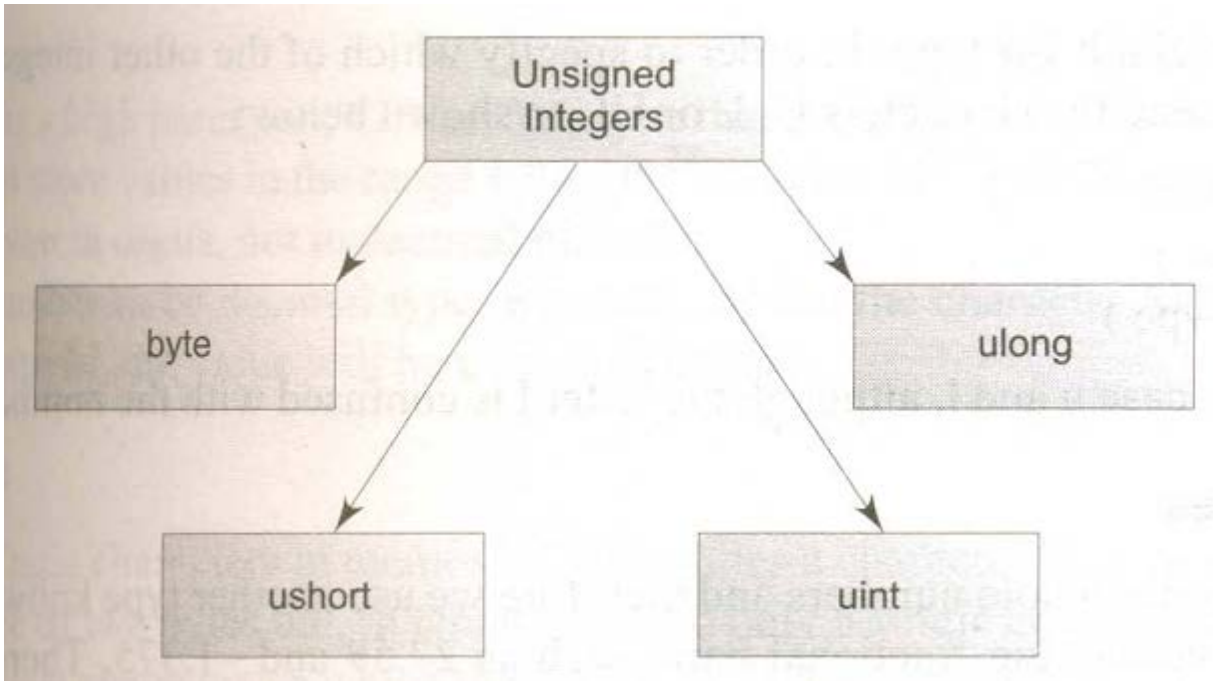
Numeric types include integral types, floating-point types and decimal types.



Integral Types

Integral types can hold whole numbers such as 123, -96 and 5639. The size of the values stored depends on the integral data type we choose. C# supports the concept of unsigned therefore it supports eight types of integers as shown below





Signed Integers

Signed integer types can hold both positive and negative numbers. The following table shows the memory size and range of all the four signed integer data types.

Table 4.3 Size and range of signed integer types

Type	*Size*	*Minimum value*	*Maximum value*
sbyte	One byte	–128	127
short	Two bytes	–32, 768	32,767
int	Four bytes	–2,147,483,648	2,147,483,647
long	Eight bytes	–9,223,372,036,854,775,808	9,223,372,036,854,775,807

Unsigned Integers

We can increase the size of the positive value stored in an integer type by making it 'unsigned'. For example a 16-bit short integer can store values in the range -32,768 to 32,767. However, by making it ushort It can handle values in the range 0 to 65,535. Table shows the size and range of all the four unsigned integer data types.

Table 4.4 Size and range of unsigned integer types

Type	*Size*	*Minimum value*	*Maximum value*
byte	One byte	0	255
ushort	Two bytes	0	65,535
uint	Four bytes	0	4,294,967,295
ulong	Eight bytes	0	18,446,744,073,709,551,615

Floating-Point Types

Integer types can hold only whole numbers and therefore we use another type known as floating type to hold numbers containing fractional parts such as 27.59 and -1.375. There are two floating point storage in C# as shown in Fig.

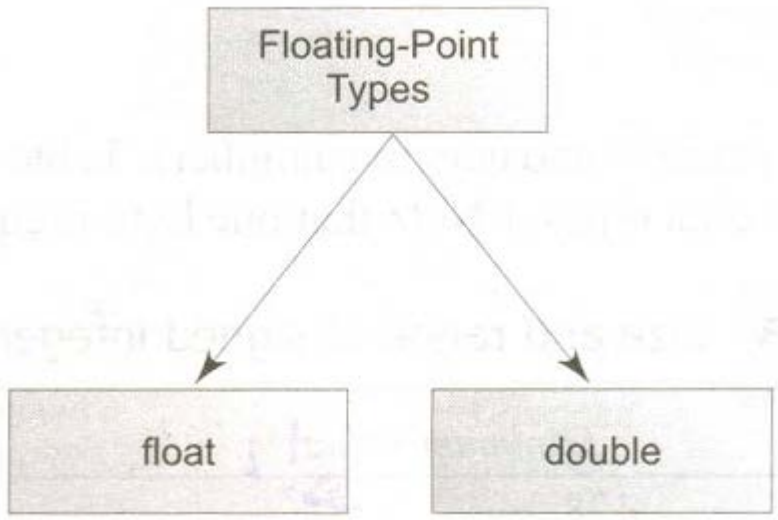


Table 4.5 Size and range of floating-point types

Type	*Size*	*Minimum value*	*Maximum value*
float	4 bytes	1.5×10^{-45}	3.4×10^{38}
double	8 bytes	5.0×10^{-324}	1.7×10^{308}

Decimal Type

The decimal type is a high precision, 128-bit data type that is designed for use in financial calculations.

To specify a number to be decimal type, we must append the character M (or m) to the value, like 123.45M. if we omit the value will be treated as double.

Example: decimal balance=2000533M;

Character Type

In order to store single characters in memory, C# provides a character data type called char. The char takes two bytes to represent single character. It has been designed to hold a 16-bit Unicodecharacter, in which the 8-bit ASCII code is a subset.

Boolean Type

Boolean type is used when we want to test a particular condition during the execution of the program. There are only two values that a Boolean type can take: true or false. Boolean type is denoted by the keyword bool and uses only one bit of storage.

Userdefined value types are further subdivided into

enum data type – it is a set of named constants that allow the user to assign symbolic names to numbers.

Syntax: enum variablename { symbolic_constant1, symbolic_constant2,. . .. }

Example: enum directions { North, South, East, West };

// by default values are 0,1,2,3

directions d = directions.North;

enum color {red=1, blue, green}; // by default values are 1,2,3

enum color {red=5, blue, green}; // by default values are 5,6,7

struct data type -A struct is user defined data type that consists of a collection of similar or dissimilar data types and methods to operate on these data types.

Note:

- ✓ struct cannot inherit from other struct except that they can derive from interfaces
- ✓ We cannot declare a default constructor for a struct, and a constructor must have parameters.
- ✓ The constructor is called only if we create our struct using new.

We cannot create a field/variable and initialize at the same time (for example, struct xyz{ public int j =250;} is illegal.

Syntax:

```
struct VariableName{
        modifier DataType identifier;
        modifier DataType identifier;  . .
                    . . .
        modifier struct_identifier ( ) // constructor
        {
            // one or more than one statements

        }   //end of the constructor

        modifier return_type method_name( )
        {
            // statements
        }//end of method

} //end of struct
```

Example:

```
struct Point
{
        int x_coordinate;
        int y_coordinate;

        public Point( int a, int b)
                        // constructor to set default values
        {
            x_coordinate = a;
                    // set default coordinates of the point
            y_coordinate = b;
        }//end of Point constructor

        public void MovePointTo(int i, int j)
        {
            x_coordinates = i;
            y_coordinates = j;
                    // move the point to new coordinates
        }//end of MovePointTo
```

Reference types

The reference types can also be divided into two groups:

- ✓ Predefined (or simple) types
- ✓ User-defined (or complex) types

Predefined reference types include two data types:

Object type – This data type can be used to declare a variable that would hold any type of value. It is treated as a “universal” data type.

Example:

object num=100;

object reat=123.66;

object str=”sample”;

String type - This data type can be used to declare a variable that would hold any string value.

Example: string str =”srinivas”;

Here str refers to a memory location where the string “srinivas” is stored.

But when string variable is declared without initialization, it gives a compile time error.

Example: string str;

System.Console.WriteLine(str);

To avoid this error, we can initialize string variable with null keyword.

String filename=”c:\\csharp\\file1.cs”;

Here each backslash has to precede by an escape sequence character ‘\’.

To avoid this extra escape character ‘\’, we can prefix the string with @ symbol.

String filename=@”c:\csharp\file1.cs”;

char ch=”Srinivas”[2];

Here ch hold third character in srinivas that is i.

String methods

string objects are immutable, meaning that we cannot modify the characters contained in them. The string is an alias for the predefined System.String class in the Common Language Runtime (CLR), there are many built-in operations available that work with strings. All operations produce a modified version of the string rather than modifying the string on which the method is called.

Table 10.1 String class methods

Method	*Operation*
Compare ( )	Compares two strings
CompareTo ( )	Compares the current instance with another instance
ConCat ( )	Concatenates two or more strings
Copy( )	Creates a new string by copying another
CopyTo ( )	Copies a specified number of characters to an array of Unicode characters
EndsWith ( )	Determines whether a substring exists at the end of the string
Equals( )	Determines if two strings are equal
IndexOf ( )	Returns the position of the first occurrence of a substring
Insert ( )	Returns a new string with a substring inserted at a specified location
Join ( )	Joins an array of strings together
LastIndexOf ( )	Returns the position of the last occurrence of a substring
PadLeft ( )	Left-aligns the strings in a field
PadRight ( )	Right-aligns the string in a field
Remove ( )	Deletes characters from the string
Replace ( )	Replaces all instances of a character with a new character
Split ( )	Creates an array of strings by splitting the string at any occurrence of one or more characters
StartsWith ( )	Determines whether a substring exists at the beginning of the string
Substring ( )	Extracts a substring
ToLower ( )	Returns a lower-case version of the string
ToUpper ( )	Returns an upper-case version of the string
Trim ( )	Removes white space from the string
TrimEnd ( )	Removes a string of characters from the end of the string

Copying Strings

We can also create new copies of existing strings. This can be accomplished in two ways:

• Using the overloaded = operator

• Using the static Copy method

Example:

string s2 = s1; //assigning

string s2 = string.Copy(s1); //copying

Both these statements would accomplish the same thing, namely, copying the contents of sl into s2.

Concatenating Strings

We may also create new strings by concatenating existing strings. There are a couple of ways to accomplish this.

• Using the overloaded + operator

• Using the static Concat method

Examples:

string s3 = s1 + s2; // s1 and s2 exist already

string s3 = string.Concat(s1, s2)

If s1 = 'abc' and s2 = 'xyz', then both the statements will store the string 'abcxyz' in s3. Note that the contents of s2 is simply appended to the contents of s2 and the result is stored in s3.

Reading from the Keyboard

It is possible to read a string value interactively from the keyboard and assign it to a string object.

string s = Console.ReadLine( );

On reaching this statement, the computer will wait for a string of characters to be entered from the keyboard. When the 'return key' is pressed, the string will be read and assigned to the string object s.

Inserting strings

The statement

string s2 = s1.Insert(3, "r");

is executed, the string variable s2 contains the string "Learn". The string "r" is inserted in s1 after characters. Similarly, the string "er" is inserted at the end of the string. Finally, the variable s3 contain the value "Learner".

Note that we are not modifying the contents of a given string variable. Rather, we are assigning the modified value to a new string variable. For instance,

sl = s1. Insert(3, "r");

is illegal. We are trying to modify the immutable string object s1.

Comparing strings

String class supports overloaded methods and operators to compare whether two strings are equal or not.

They are:

• Overloaded Compare( ) method

• Overloaded Equals( ) method

• Overloaded == operator

Compare Method

There are two versions of overloaded static Compare method. The first one takes two strings as parameters and compares them. Example:

int n = string.Compare(sl,s2);

This performs a case-sensitive comparison and returns different integer values for different conditions as under:

• Zero integer, if s1 is equal to s2

• A positive integer (1), if s1 is greater than s2

• A negative integer (-1), if s1 is less than s2

For example, if sl = "abc" and s2 = "ABC", then n will be assigned a value of -1. Remember, a lowercase letter has a smaller ASCII value than an uppercase letter.

We can use such comparison statements in if statements like:

if (string.Compare(s1, s2) == 0)

Console.WriteLine("They are equal");

The second version of Compare takes an additional bool type parameter to decide whether case should be ignored or not. If the bool parameter is true, case is ignored. Example:

int n = string.Compare(s1, s2, true);

This statement compares the strings s1 and s2 ignoring the case and therefore returns zero when s1 = "abc" and s2 = "ABC", meaning they are equal.

Equals Method

The string class supports an overloaded Equals method for testing the equality of strings. There are again two versions of Equals method. They are implemented as follows:

bool b1 = s2.Equals(s1);

bool b2 = string.Equals (s2, s1);

These methods return a Boolean value true if s1 and s2 are equal, otherwise false.

The = = Operator

A simple and natural way of testing the equality of strings is by using the overloaded == operator.

Example:

bool b3 = (s1 == s2); //b3 is true if they are equal

We very often use such statements in decision statements, like

```
if (s1 ==  s2)

Console.WriteLine("They are qual");
```

Finding substrings

It is possible to extract substrings from a given string using the overloaded Substring method available in String class. There are two version of Substring:

- s.Substring(n)
- s.Substring(n1, n2)

The first one extracts a substring starting from the nth position to the last character of the string contained in s. The second one extracts a substring from s beginning at nl position and ending at n2 position. Examples:

```
string sl =  "NEW YORK";

string s2 =  sl.Substring(5);

string s3 = sl.Substring(O,3);

string s4 = sl.Substring(5,8);
```

When executed, the string variables will contain the following substrings:

s2: YORK

s3: NEW

s4: YORK

Mutable strings

Mutable strings that are modifiable can be created using the StringBuilder class. Examples:

```
StringBuilder strl = new StringBuilder("abc");

StringBuilder str2 = new StringBuilder ( );
```

The string object strl is created with an initial size of three characters and str2 is created as an empty string. They can grow dynamically as more characters are added to them. They can grow either un-bounded or up to a configurable maximum. Mutable strings are also known as dynamic strings.

The StringBuilder class supports many methods that are useful for manipulating dynamic strings.

Table 10.2 Some useful stringBuilder methods

Method	*Operation*
Append ( )	Appends a string
AppendFormat ( )	Appends strings using a specific format
EnsureCapacity ( )	Ensures sufficient size
Insert ( )	Inserts a string at a specified position
Remove ( )	Removes the specified characters
Replace ( )	Replaces all instances of a character with a specified one

C# also supports some special functions known as *properties*. They are listed in Table 10.3

Table 10.3 StringBuilder properties

Property	*Purpose*
Capacity	To retrieve or set the number of characters the object can hold
Length	To retrieve or set the length
MaxCapacity	To retrieve the maximum capacity of the object
[ ]	To get or set a character at a specified position

Using StringBuilder methods

```
using System.Text;    //For using StringBuilder
using System;
class StringBuilderMethod
{
    public static void Main( )
    {
        StringBuilder s = new StringBuilder ("Object ");
        Console.WriteLine("Original string   : " + s);
        Console.WriteLine("Length            : " + s.Length);

        //Appending a string
        s.Append("language ");
        Console.WriteLine("String now        : " + s);

        //Inserting a string
        s.Insert (8,"oriented ");
        Console.WriteLine("Modified string   : " + s);

        //Setting a character
        int n = s.Length;
            s[n] = '!' ;
        Console.WriteLine("Final string      : " + s);
    }
}
```

Look at the output produced by Program 10.2:

```
Original string    :    Object
Length             :    7
String now         :    Object language
Modified string    :    Object oriented language
Final string       :    Object oriented language!
```

Array of strings

We can also create and use arrays that contain strings. The statement

string [ ] itemArray = new string [3];

will create an itemArray of size 3 to hold three strings. We can assign the strings to the itemArray element by element using three different statements, or more efficiently using a for loop. We could also provide an array with a list of initial values in curly braces:

string [ ] itemArray = ("Java", "C++", "Csharp");

The size of the array is determined by the number of elements in the initialization list. The size of the array, once created, cannot be changed.

Regular expressions

Regular expressions provide a powerful tool for searching and manipulating a large text. A regular expression may be applied to a text to accomplish tasks such as:

• To locate substrings and return them

• To modify one or more substrings and return them

• To identify substrings that begin with or end with a pattern of characters

• To find all words that begin with a group of characters and end with some other characters

• To find all the occurrences of a substring pattern

A regular expression (also known as a pattern string) is a string containing two types of characters.

• Literals

• Metacharacters

Literals are characters that we wish to search and match in the text. Metacharacters are special characters that give commands to the regular expression parser. Examples of regular expressions are:

Expression	*Meaning*
"\bm"	Any word beginning with m
"er\b"	Any word ending with er.
"\BX\B"	Any X in the middle of a word
"\bm\S*er\b"	Any word beginning with m and ending with er.
" |,"	Any word separated by a space or a comma

The .NET Framework provides support for regular expression matching and replacement. The namespace System.Text.RegularExpressions supports a number of classes that can be used for searching, matching and modifying a text document. The important classes are:

• Regex

• MatchCollection

• Match

Program 10.4

Use of regular expressions

```
using System;
using System.Text;    //for StringBuilder class
using System.Text.RgularExpressions;  //for Regex class

Class RegexTest

{
     public static void Main ( )
     {

          string str;
          str = "Amar, Akbar, Antony are friends!";

          Regex reg = new Regex (" |, ");
          StringBuilder sb = new StringBuilder( );
          int count = 1;

          foreach(string sub in reg.Split(str))
          {
                sb.AppendFormat("{0}: {1}\n", count++, sub);
          }
          Console.WriteLine(sb);
     }
}
```

Program 10.4 would produce the following output:

```
1: Amar
2: Akbar
3: Antony
4: are
5: friends!
```

The program creates a regular expression object **reg** using the class **Regex** and stores the regular expression

```
" |,"
```

in it. This pattern is used to search the string

"Amar, Akbar, Antony are friends!"

with the help of **Split( )** method of **Regex** class. The call

```
reg.Split(str)
```

in the **foreach** statement splits the string wherever a comma or space appears and returns the substring to **sub**. These substrings are appended to the **StringBuilder** string **sb** using the **AppendFormat** method of **StringBuilder.** The output shows that the given string has been split into separate words using the separators 'space' and 'comma'.

User defined reference types refer to those types which we define using predefined types. They include

Arrays – An array is a collection of similar data types stored in adjacent memory locations. Each element in an array is accessed through indices. By default, the first element is indexed 0, the next element 1, and so on. The C# provides three different kinds of arrays: Single-dimensional arrays, multi-dimensional arrays, and variable-size arrays

Single-dimensional arrays

A single-dimensional array is a collection of data of the same type.

Like other variables, arrays must be declared and created in the computer memory before they are used. Creation of an array involves three steps:

I. Declaring the array

2. Creating memory locations

3. Putting values into the memory locations.

Declaration of Arrays

Arrays in C# are declared as follows:

type[ ] arrayname;

Examples:

int[ ] counter;

Remember, we do not enter the size of the arrays in the declaration.

Creation of Arrays

After declaring an array, we need to create it in the memory. C# allows us to create arrays using new operator only, as shown below:

arrayname = new type[size];

Examples:

number = new int[5]; //create a 5 element int array

average = new float[10]; // create a 10 element float array

It is also possible to combine the two steps, declaration and creation, into one as shown below:

int[ ] number = new int[5]; // declare and create 5 element int array

Initialization of Arrays

The final step is to put values into the array created. This process is known as initialization. This is done using the array subscripts as shown below.

arrayname[subscript] = value;

number [0] = 35;

C# creates arrays starting with a subscript of 0 and ends with a value one less than the size specified.

We can also initialize arrays automatically in the same way as the ordinary variables when they are declared, as shown below:

type [ ] arrayname = {list of values};

The array initializer is a list of values separated by commas and defined on both ends by curly braces. Note that no size is given. The compiler allocates enough space for all the elements specified in the list.

Example:

int[ ] number = {35, 40, 20, 57, 19};

The preceding line is equivalent to:

int [ ] number = new int [5] { 35, 40, 20, 57, 19 };

This combines all the three steps, namely declaration, creation and initialization.

Array Length

We can access the length of the array a using a.Length. Example:

int aSize = a.Length;

Multi-dimensional arrays

Two or more than two-dimensional arrays are known as multi-dimensional arrays. The first index indicates the rows and the second index gives the columns in a two-dimensional array

Syntax:

```
DataType[ , ] VariableName ; // two dimensional array
DataType[, ,] VariableName; // three dimensional array
```

Examples:

```
int [ , ] myArray ; // declaring myArray as reference to two
dimensional integer array.

int[ , ] oddArray = { { 3 ,5, 7, 9},

{ 11, 13, 15, 17}
}; // a two dimensional array of size 2 X 4 elements
```

We can access the oddArray elements using *for* loops.

```
for( int i = 0; i<4; i++)
    for(int j=0 ; j<4; j++)
    System.Console.WriteLine("oddArray element [{0},{1}] is {2}", i,j,
                                                    yourArray[i,j]);
```

Variable-Size Arrays

C# treats multidimensional arrays as 'arrays of arrays'. It is possible to declare a 2D array as follows:

```
int[ ][ ] x= new int[3][];    //three rows array
x[0]  = new int[2];           //first row has two elements
x[1]  = new int[4];           //second row has four elements
x[2]  = new int[3];           //third row has three elements
```

These statements create a two-dimensional array having different lengths for each row as shown in Fig. 9.3. Variable-size arrays are called *jagged* arrays.

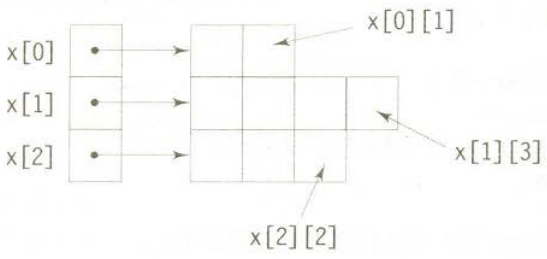


Table 9.1 Some commonly used methods of System.Array class

Method/Property	*Purpose*
Clear ( )	Sets a range of elements to empty values
CopyTo ( )	Copies elements from the source array into the destination array
GetLength ( )	Gives the number of elements in a given dimension of the array
GetValue ( )	Gets the value for a given index in the array
Length	Gives the length of an array
SetValue ( )	Sets the value for a given index in the array
Reverse ( )	Reverses the contents of a one-dimensional array
Sort ( )	Sorts the elements in a one-dimensional array

Sorting and reversing an array

```
using System;
class SortReverse
{
   public static void Main( )
   {
        //creating an array
        int [ ] x = { 30, 10, 80, 90, 20 };

        Console.WriteLine ("Array before sorting");
        foreach (int i in x)
             Console.Write(" " + i );
        Console.WriteLine ( );

        //Sorting the array elements
        Array.Sort(x);

        Console.WriteLine("Array after Sorting");
        foreach ( int i in x )
             Console.Write(" " + i);
        Console.WriteLine ( );
}
```

ArrayList class

System.Collections namespace defines a class known as ArrayList that can store a dynamically sized array of objects. The ArrayList class includes a number of methods to support operations such as sorting, removing and enumerating its contents.

An array list is very similar to an array, except that it has the ability to grow dynamically. We can create an array list by indicating the initial capacity we want. Example:

ArrayList cities = new ArrayList (30);

It creates cities with a capacity to store thirty objects. If we do not specify the size, it defaults to sixteen. That is,

ArrayList cities = new ArrayList ( );

will create a cities list with the capacity to store sixteen objects. We can now add elements to the list using the Add () method:

cities.Add ("Bombay");

cities.Add ("Anand");

We can also remove an element:

cities.RemoveAt (1);

This will remove the object in position 1. We can modify the capacity of the list using the property Capacity:

cities.Capacity = 20;

We may obtain the actual number of objects present in the list using the property Count as follows:

int n = cities.Count;

An array list can be really useful if we need to create an array of objects but we do not know in advance how big the array would be. Further, an array list can contain any object reference.

Table 9.2 lists some of the most important methods and properties supported by the **ArrayList** class.

Table 9.2 Some important ArrayList methods and properties

Methods/Property	*Purpose*
Add ( )	Adds an object to a list
Clear ( )	Removes all the elements from the list
Contains ( )	Determines if an element is in the list
CopyTo ( )	Copies a list to another
Insert ( )	Inserts an element into the list
Remove ( )	Removes the first occurrence of an element
RemoveAt ( )	Removes the element at the specified place
RemoveRange ( )	Removes a range of elements
Sort ( )	Sorts the elements
Capacity	Gets or sets the number of elements in the list
Count	Gets the number of elements currently in the list.

Using ArrayList class

```
using System;
using System.Collections;
class City
{
    public static void Main( )
    {
        ArrayList n = new ArrayList ( );
        n.Add ("Madras");
        n.Add ("Bombay");
        n.Add ("Anand");
        n.Add ("Calcutta");
        n.Add ("Delhi");
        Console.WriteLine("Capacity = "+ n.Capacity);
        Console.WriteLine("Elements present = " + n.Count);

        n.Sort( );
        for (int i = 0, i < n.Count; i ++)
        {
            Console.WriteLine(n[i]);
        }
        Console.WriteLine( );

            n.RemoveAt(4);
            for (int i = 0 ; i < n.Count ; i ++)
            {
                Console.WriteLine(n[i]);
            }
    }
}
```

Type Conversion

The process of converting data of one type to another is called type conversion.

In C#, type conversions take place in two ways

- Implicit conversions
- Explicit conversions

Implicit conversions - Implicit conversion is a process of performing automatic conversion between primitive data types without the knowledge of the user, and without any loss of data during conversion.

Explicit conversions - Explicit conversion allows conversion from any numeric type to any other (for which no implicit conversion is available) by using two mechanisms:

1. Value casting or Typecasting

2. Boxing and Unboxing

Value casting, also known as Typecasting (or simply 'casting'), can be used to change the data of one primitive type to another compatible primitive type. Suppose we have a long data type value, and an expression is expecting an int data type value, then we could typecast a long value into int value as follows.

```
long numberl = 23844L;

int number2 = (int) numberl; // casting long data type value to int data type value.
```

Boxing and Unboxing

The boxing mechanism allows us to convert a value type to a object type. And the reverse process of converting a object type to a value type is called unboxing.

Boxing example:

```
int xyz = 4657;

object abc = xyz; // Boxing. Here variable xyz is converted into an object variable abc.
```

Unboxing example:

```
xyz = (int) abc; // Unboxing. Here object variable abc is cast back to int variable xyz.
```

Decision Making and Branching

C# supports following Decision Making statements

1. The if-else statement

2. The switch statement

3. Ternary operator (?:)

1. The if-else statement

The if-else statement can be implemented in any of the following four ways:

• Using only if statement

• Using the if-else statements

• Using nested if-else statements

• Using else-if ladder.

(i) Using only ***if*** statement:

Syntax:

```
    if ( condition is true)    // do this single statement;
OR  if ( condition is true)
    {
      // do this block of statements;
    }
```

(ii) Using the ***if-else*** statement (Listing 5.1): The ***if-else*** statement allows the program to branch to one of the immediate next statements depending on whether the condition is true or false (see Figure 5.4).

```
if ( condition is true)
    {
    // do this block;
    }
    else
    {
    // do this block;
    }
```

(iii) Using **nested *if-else*** statements: Nested *if-else* statements will be required when a series of decisions are involved (see Figure 5.5).

Syntax:

```
if ( condition_1 is true)
    if (condition_2 is true)
        // do the statement1
    else
        // do this statement2
    else
        // do the statement3
    . . .
```

(iv) Using ***else-if* ladder** (Listing 5.2): A multi-path decision can be resolved using chain of *if-else* statements such that, with each *else* an *if* is associated (see Figure 5.6).

Syntax:

```
If ( condition_1 is true)
    // do statement_1
else if(condition_2 is true)
    // do statement_2
else if(condition_3 is true)
    // do statement_3
    . . .
    . . .
else if(condition_N is true)

    // do statement_N
    else
    // do default statement
```

Switch Statement

Syntax:

```
switch (expression)
{
        case value_1:
                // do block_1 statements
                break;
  case value_2:
                // do block_2 statements
                 break;
              . . .
              . . .
  case value_N:
                // do block_N statements
                break;
  default::
                // do block_1 statements
                 break;
    }
```

5.1.3 Ternary Operator (?:)

Ternary opration (?:) uses three operands. Here, the condition is tested for true or false. If true, then the expression immediately after **'?'** symbol will be executed. If the condition is false, then the expression immediately after **':'** symbol will be executed.

Syntax:

```
(Condition) ? (Expression1) : (Expression2);
```

This is equivalent to:

```
if (Condition is true)
     Expression1;
else
     Expression2;
```

Decision Making and Looping

5.2.1 The *do-while* Loop Statement

The do-while statement (Listing 5.4) is useful when at least one iteration is required to be executed before the test condition is resolved. The *do* loop continues execution until the test condition is false.

Syntax:

```
do{
        // block of statements
} while (test condition is true);
```

5.2.2 The *for* Statement

The *for* statement (Listing 5.5) is useful in circumstances where the number of iterations is known beforehand. The *for* loop checks the test condition at entry level itself, and then starts executing as long as the test condition is true. The *for* loop does three things:

- Initializing expression
- Executing block of statements if the test condition is true
- Incrementing the iteration number

Syntax:

```
for( initialization; test condition ; increment)
 {
            // block of statements
 }
```

5.2.3 The *foreach* Loop Statement

The *foreach* loop statement (Listing 5.6) is used to enumerate the contents of a collection (such as an array, enum, etc.).

Syntax:

```
foreach (type in expression)
// do statement;
```

Example:

```
int [] myArray = new int[ ] { 10,20,30,40,50};
foreach ( object k in myArray)
System.Console.WriteLine("The element in myArray is: {0}", k);
      // Displays 10,20,30,40,50.
```

```
//Illustration of foreach construct
namespace Chapter5_05
{
   using System;
   class ForConstruct
   {
         public static void Main()
         {
          string [] Gods = { "Krishna", "Christ", "Mohammed", "Buddha"};
          Console.Write("For each God:\n");

          foreach (object x in Gods)
          Console.WriteLine("{0}\t", x);
          Console.WriteLine("there is a sanctum sanctorum");

         }//end of Main
     }
}//end of Chapter5_06 namespace
```

5.2.4 The *while* Statement

The ***while*** statement (Listing 5.7) checks the test condition at the entry point itself and then executes statements if test condition is true. Else, the while loop exits.

Syntax:

```
// initialization
while (test condition is true)
// do statement
```

Flow Control Statements

We can control the flow of execution in loops by using control statements known as jump statements: break, continue and goto.

The break statement

The break statement is used to terminate the current enclosing loop.

Syntax: break;

The continue statement

The continue statement stops the current iteration and simply returns control back to the top of the loop.

Syntax: continue;

The goto statement

The goto statement can be used to jump from inside loop to outside loop.

Syntax: goto label;

Methods in C#

The code designed to work on the data is known as methods.

Declaring methods

Methods are declared inside the body of a class, normally after the declaration of data fields. The general form of a method declaration is

modifiers type methodname (formal-parameter-list)

{

    Method_body

}

Method declaration has five parts:

- Name of the method (methodname)
- Type of value the method returns (type)
- List of parameters (formal-parameter-list)
- Body of the method
- Method modifiers (modifier)

The methodname is a valid C# identifier. The type specifies the type of value the method will return. This can be a simple data type such as int as well as any class type. If the method does not return anything, we specify a return type of void. Note that we cannot omit the return type altogether.

The formal-parameter-list is always enclosed in parentheses. This list contains variable names and types of all the values we want to give to the method as input. The parameters are separated by commas. In the case where no input data are required, the declaration must still include an empty set of parantheses() after the method name. Examples are:

```
int Fun1 ( int m, float x, float y )  //three parameters

void Display ( ) //no parameters
```

The body enclosed in curly braces actually describes the operations to be performed on the data. For example, the code segment below computes the product of two integer values and returns the result.

```
int Product ( int x, int y )

{

int m = x * y;

return(m);

}
```

The modifiers specify keywords that decide the nature of accessibility and the mode of application of the method. A method can take one or more of the modifiers listed in Table

Modifier	**Description**
public	The method can be accessed from anywhere, including outside the class
protected	The method is available both within the class and within the derived class
internal	The method is available only within the file
private	The method is available only within the class
static	To declare as class method
abstract	A abstract method which defines the signature of the method, but doesn't provide an implementation
virtual	The method can be overridden by a derived class
override	The method overrides an inherited virtual or abstract method
new	The method hides an inherited method with the same signature
sealed	The method overrides an inherited virtual method, but cannot be overridden by any classes which inherit from this class.

Invoking methods

Once methods have been defined, they must be activated for operations. The process of activating a method is known as invoking or calling. The invoking is done using the dot operator as shown below:

objectname.methodname(actual-parameter-list);

Here, objectname is the name of the object on which we are calling the method methodname. The actual-parameter-list is a comma separated list of 'actual values' (or

expressions) that must match in type, order and number with the formal parameter list of the methodname declared in the class.

Method parameters

The invocation involves not only passing the values into the method but also getting back the results from the method. For managing the process of passing values and getting back the results, C# employs four kinds of parameters.

• Value parameters

• Reference parameters

• Output parameters

• Parameter arrays

Value parameters are used for passing parameters into methods by value. On the other hand reference parameters are used to pass parameters into methods by reference. Output parameters, as the name implies, are used to pass results back from a method. Parameter arrays are used in a method definition to enable it to receive variable number of arguments when called.

Pass by value

By default, method parameters are passed by value. That is, a parameter declared with no modifier is passed by value and is called a value parameter. When a method is invoked, the values of actual parameters are assigned to the corresponding formal parameters. The values of the value parameters can be changed within the method. The value of the actual parameter that is passed by value to a method is not changed by any changes made to the corresponding formal parameter within the body of the method. This is because the methods refer to only copies of those variables when they are passed by value.

Pass by reference

Default behaviour of methods in C# is pass by value. We can, however, force the value parameters to be passed by reference. To do this, we use the ref keyword. A parameter declared with the ref modifier is a reference parameter. Example:

void Modify (ref int x)

Here, x is declared as a reference parameter.

Unlike a value parameter, a reference parameter does not create a new storage location. Instead, it presents the same storage location as the actual parameter used in the method invocation. Example:

```
void Modify ( ref int x )

{

x+=10;

}

int m = 5;

Modify ( ref m ); // pass by reference
```

Reference parameters are used in situations where we would like to change the values of variables in the calling method. When we pass arguments by reference, the 'formal' arguments in the called method become aliases to the 'actual" arguments in the calling method. This means that when the method is working with its own arguments, it is actually working with the original data.

Swapping values using ref parameters

```
using System;
class PassByRef
{
    static void Swap ( ref int x, ref int y )
    {
        int temp = x;

        x = y;
        y = temp;
    }
    public static void Main( )
    {
        int m = 100;
        int n = 200;
        Console.WriteLine("Before Swapping:");
        Console.WriteLine("m = " + m);
        Console.WriteLine("n = " + n);

        Swap( ref m , ref n );

        Console.WriteLine("After Swpaping:");
        Console.WriteLine("m = " + m);
        Console.WriteLine("n = " + n);
    }
}
```

The output parameters

Output parameters are used to pass results back to the calling method. This is achieved by declaring the parameters with an out keyword. Similar to a reference parameter, an output parameter does not create a new storage location. Instead, it becomes an alias to the parameter in the calling method. When a formal parameter is declared as out, the corresponding actual parameter in the calling method must also be declared as out. Example:

```
void Output ( out int x )

{

x = 100;

}

int m; //m is uninitialized

Output ( out m ); //value of m is set
```

Note that the actual parameter m is not assigned any values before it is passed as output parameter. Since the parameters x and m refer to the same storage location, m takes the value that is assigned to x.

Variable argument lists

In C#, we can define methods that can handle variable number of arguments using what are known as parameter arrays. Parameter arrays are declared using the keyword params. Example:

```
void Functionl (Params int [] x )

{

--------

}
```

Here, x has been declared as a parameter array. Note that parameter arrays must be one-dimensional arrays. A parameter may be a part of a formal parameter list and in such cases, it must be the last parameter.

The method Functionl defined above can be invoked in two ways:

• Using int type an-ay as a value parameter. Example: Functionl(a);

  Here, a is an array of type int

• Using zero or more int type arguments for the parameter array. Example: Function (10, 20);

The second invocation creates an int type array with two elements 10 and 20 and passes the newly created array as the actual argument to the method.

Concept of variable arguments

```
using System;
class Params
{
    static void Parray ( params int [ ] arr )
    {
        Console.Write("Array elements are:");
        foreach (int i in arr)
            Console.Write(" " + i);
        Console.WriteLine( );
    }
    public static void Main( )
    {
        int [ ] x = { 11, 22, 33 } ;
        Parray ( x) ;            // call 1
        Parray ( ) ;             // call 2
        Parray ( 100, 200 ) ;    // call 3
    }
}
```

We can also use parameter arrays of type **object**. Example:

```
public static void Main( )
{
    Oarray ( 10, 20, "abc" ) ;
}
static void Oarray ( params object [ ] x )
{
    foreach ( object i in x )
    {
        Console.WriteLine( i );
    }
}
```

Methods overloading

C# allows us to create more than one method with the same name, but with the different parameter lists and different definitions. This is called method overloading. Method

overloading is used when methods are required to perform similar tasks but using different input parameters.

Overloaded methods must differ in number and/or type of parameters they take. This enables the compiler to decide which one of the definitions to execute depending on the type and number of arguments in the method call.

Using the concept of method overloading, we can design a family of methods with one name but different argument lists. For example, an overloaded add( ) method handles different types of data as shown below:

```
// Method definitions
int add ( int a, int b ) { … }          //Method1
int add ( int a, int b, int c ) { … }   //Method2
double add ( float x, float y ) { … }   //Method3
double add ( int p, float q ) { … }     //Method4
double add (float p, int q ) { … }      //Method5

//Method calls
int m = add ( 5, 10 ) ;                 //calls method1
double x = add ( 15, 5.0F ) ;           //calls method4
double x = add ( 1.0F, 2.0 F );         //calls method3
int m = add ( 5, 10, 15 ) ;             //calls method2
double x = add ( 2.0F, 10 ) ;           //calls method5
```

Classes and Objects

C# is a true object-oriented language and therefore the underlying structure of all C# programs is classes. Anything we wish to represent in a C# program must be encapsulated in a class that defines the state and behaviour of the basic program components known as objects. Classes create objects and objects use methods to communicate between them.

Classes provide a convenient approach for packing together a group of logically related data items and functions that work on them. In C#, the data items are called fields and the functions are called methods.

Defining a class

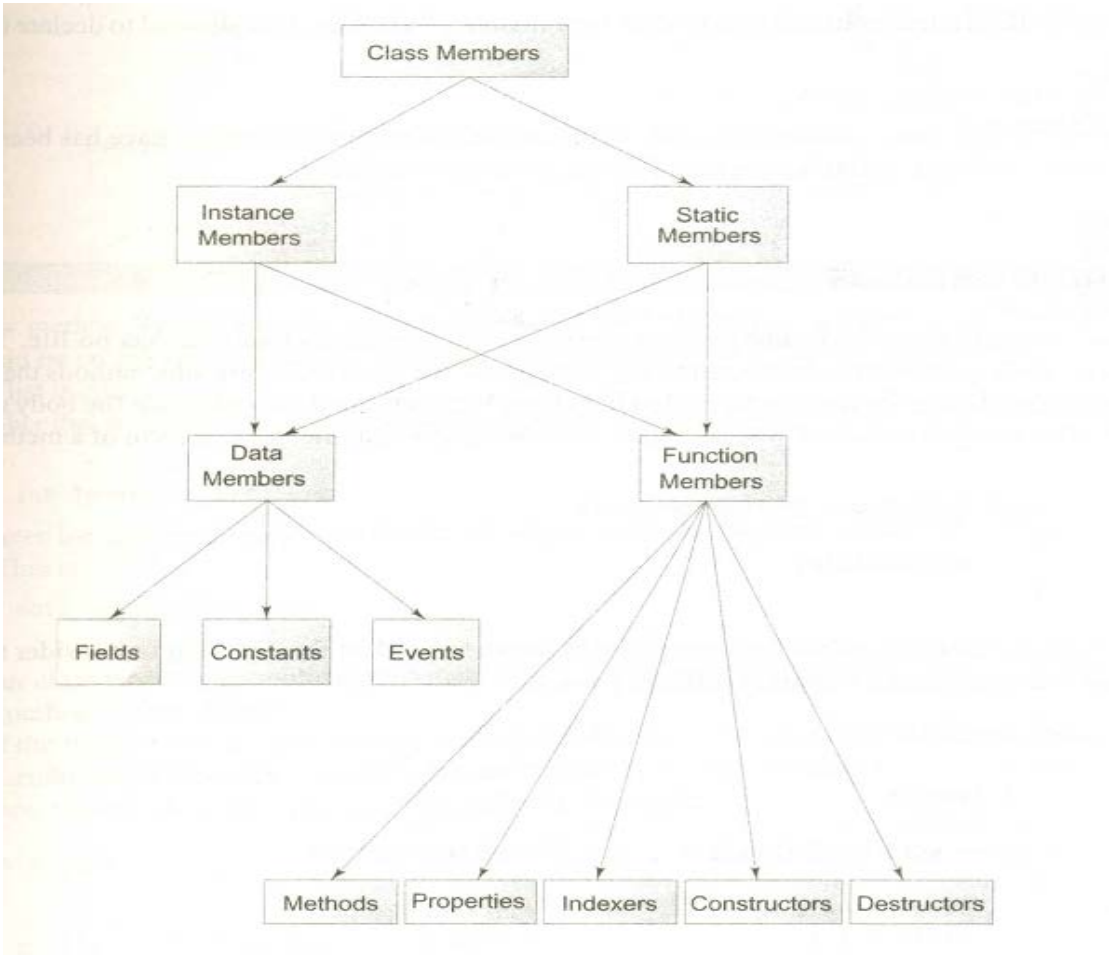
A class is a user-defined data type with a template that serves to define its properties. Once the class type has been defined, we can create 'variables' of that type using declarations that are similar to the basic type declarations. In C#, these variables are termed as instances of classes, which the actual objects. The basic form of a class definition is:

```
class classname
{
    [ variables declaration; ]
    [ methods declaration; ]
}
```

class is a keyword and *classname* is any valid C# identifier. Everything inside the square brackets is optional. This means that the following would be a valid class definition:

```
class Empty     //class name is Empty
{
}
```

Categories of Class members



Adding variables

Data is encapsulated in a class by placing data fields inside the body of the class definition. These variables are called instance variables because they are created whenever an object of the class is instantiated. We can declare the instance variables exactly the same way as we declare local variables.

Example:

```
class Rectangle
{
    int length; //instance variable
    int width;  //instance variable
}
```

Adding methods

A class with only data fields and without methods that operate on that data has no life. The objects created by such a class cannot respond to any messages. We must therefore add methods that are necessary for manipulating the data contained in the class. Methods are declared inside the body of the class, usually after the declaration of instance variables. The general form of a method declaration is

```
type methodname (parameter-list)
{
    method-body;
}
```

The *body* actually describes the operations to be performed on the data. Let us consider the **Rectangle** class again and add a method **GetData ( )** to it.

```
class Rectangle
{
    int length;
    int width;
    public void GetData(int x , int y)//mutator method
    {
        length = x ;
        width = y ;
    }
}
```

Member class modifiers

One of the goals of object-oriented programming is 'data hiding'. That is, a class may be designed to hide its members from outside accessibility. C# provides a set of 'access modifiers' that can be used with the members of a class to control their visibility to outside users.

Table 12.1 C# access modifiers

Modifier	*Accessibility Control*
private	Member is accessible only within the class containing the member
public	Member is accessible from anywhere outside the class as well. It is also accessible in derived classes.
protected	Member is visible only to its own class and its derived classes.
internal	Member is available within the assembly or component that is being created but not to the clients of that component
protected internal	Available in the containing program or assembly and in the derived classes.

In C#, all members have **private** access by default. If we want a member to have any other visibility range, then we must specify a suitable access modifier to it individually. Example.

```
class Visibility
{
    public int x;
    internal int y;
    protected double d;
    float p;  //private by default
}
```

Note we cannot declare more than one member under a visibility modifier. For instance, the code

```
public: //allowed in C++
int x;
int y;
```

is illegal in C#.

Creating Objects

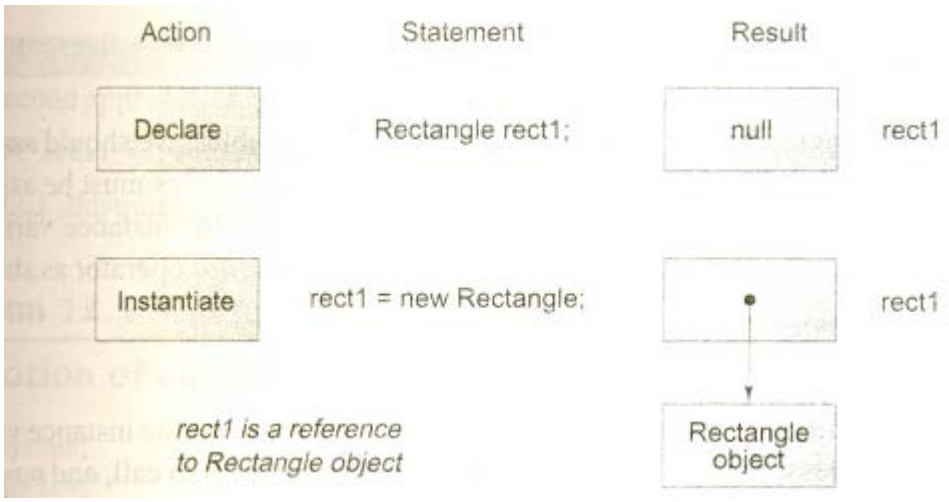
An object in C# is essentially a block of memory that contains space to store all the instance variables. Creating an object is also referred to as instantiating an object.

Object in C# are created using the new operator. The new operator creates an object of the specified class and returns a reference to that object. Here is an example of creating an object of type Rectangle.

Rectangle rect1; // declare

rectl = new Rectangle( ); // instantiate

The first statement declares a variable to hold the object reference and the second one actually assigns the object reference to the variable. The variable rect1 is now an object of the Rectangle class.



Accessing class members

All variables must be assigned values before they are used. Since we are outside the class, we cannot access the instance variables and the methods directly. To do this, we must use the concerned object and the dot operator as shown below:

objectname.variable name;

objectname .methodname (parameter-list) ;

Here objectname is the name of the object, variablename is the name of the instance variable inside the object that we wish to access, methodname is the method that we wish to call, and parameter-list is a comma separated list of 'actual values' (or expressions) that must match in type and number with the parameter list of the methodname declared in the class. The instance variables and methods of the Rectangle class may be accessed and assigned values as follows:

```
rect1.length = 15;
rect1.width  = 10;
rect2.length = 20;
rect2.width  = 12;
```

```
Rectangle rect1 = new Rectangle( );
rect1.GetData(15,10);      // calling the method
```

Application of classes and objects

```
using System;
class Rectangle
{
  public int length, width;                    //   Declaration of variables

  public void GetData(int x, int y)            //   Definition of method
  {
    length = x;
    width  = y;
  }

  public int RectArea()                        //   Definition of another method
  {
    int area = length * width;
    return (area);
  }
}
```

```
class RectArea                                 //   class with main method
{
  public static void Main( )
  {
    int area1,area2;                           //   Local variables
    Rectangle rect1 = new Rectangle();         //   Creating objects
    Rectangle rect2 = new Rectangle();

    rect1.length = 15;                         //   Accessing variables
    rect1.width  = 10;
    area1 = rect1.length * rect1.width;

    rect2.GetData(20,12);                      //   Accessing methods
    area2 = rect2.RectArea();

    Console.WriteLine("Area1 = " + area1);
    Console.WriteLine("Area2 = " + area2);
  }
}
```

Constructors

C# supports a special type of method called a constructor that enables an object to initialize itself when it is created.

Constructors have the same name as the class itself. Secondly, they do not specify a return type, not even void. This is because they do not return any value.

```
class Rectangle
{
        public int length ;
        public int width ;
        public Rectangle(int x, int y) // Constructor method
        {
                length = x ;
                width  = y ;
        }
        public int RectArea( )
        {
            return(length * width);
        }
}
```

Constructors are usually public because they are provided to create objects. However, they can also be declared as private or protected. In such cases, the objects of that class cannot be created and also the class cannot be used as a base class for inheritance.

Overloaded constructors

It is possible to create methods that have the same name, but different parameter lists and different definitions. This is called method overloading. Method overloading is used when objects are required to perform similar tasks but using different input parameters. When we call a method in an object, C# matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute. This process is known as polymorphism. We can extend the concept of method overloading to provide more than one constructor to a class.

To create an overloaded constructor method, all we have to do is to provide several different constructor definitions with different parameter lists. The difference may be in either the number or type of arguments. That is, each parameter list should be unique. Here is an example of creating an overloaded constructor

```
class Room
{
      public double length ;
      public double breadth ;

      public Room(double x, double y)          // constructor1
      {
            length = x ;
            breadth = y ;
      }

      public Room(double x)                    // constructor2
      {
            length = breadth = x ;
      }

      public int Area( )
      {
            return (length * breadth) ;
      }
}
```

Here we are overloading the constructor method Room( ). An object representing a rectangular room will be created as

Room rooml = new Room(25.0,15.0); //using constructor!

On the other hand, if the room is square, then we may create the corresponding object as

Room room2 = new Room(20.0); // using constructor2

Static members

A class contains two sections. One declares variables and the other declares methods. These variables and methods are called instance variables and instance methods. This is because everytime the class is instantiated, a new copy of each is created. They are accessed using the objects (with dot operator).

Let us assume that we want to define a member that is common to all the objects and accessed without using a particular object. That is, the member belongs to the class as a whole rather than the objects created from the class. Such members can be defined as follows:

static int count ;

static int max(int x, int y);

The members that are declared static as shown above are called static members. Since these members are associated with the class itself rather than with individual objects, the static variables and static methods are often referred to as class variables and class methods.

Static variables are used when we want to have a variable common to all instances of a class. Like static variables, static methods can be called without using the objects. They are also available for use by other classes. Methods that are of general utility but do not directly affect an instance of that class are usually declared as class methods. C# class libraries contain a large number of class methods. For example, the Math class of C# System namespace defines many static methods to perform math operations that can be used in any program.

double x = Math.Sqrt(25.0);

The method Sqrt is a class method (or static method) defined in Math class.

Defining and using static members

```
using System;
class Mathoperation
{
    public static float mul(float x, float y);
    {
        return x*y;
    }
    public static float divide(float x, float y)
    {
        return x/y ;
    }
}

class MathApplication
{
    public void static Main( )
    {
        float a = MathOperation.mul(4.0F,5.0F) ;
        float b = MathOperation.divide(a,2.0F) ;
        Console.WriteLine("b = "+ b) ;
    }
}
```

Note that the static methods are called using class names. In fact, no objects have been created for use. Static methods have several restrictions:

• They can only call other static methods.

• They can only access static data.

• They cannot refer to this

Static constructors

A static constructor is called before any object of the class is created. This is useful to do any housekeeping work that needs to be done once. It is usually used to assign initial values to static data members.

A static constructor is declared by prefixing a static keyword to the constructor definition. It cannot have any parameters. Example:

```
class Abc
{
     static Abc ( ) //No parameters
     {
          . . . .   //set values for static members here
     }
     . . . .
}
```

Note that there is no access modifier on static constructors. It cannot take any. A class can have only one static constructor.

Private constructors

C# does not have global variables or constants. All declarations must be contained in a class. In many situations, we may wish to define some utility classes that contain only static members. Such classes are never required to instantiate objects. Creating objects using such classes may be prevented by adding a private constructor to the class.

Copy constructors

A copy constructor creates an object by copying variables from another object. For example, we may wish to pass an Item object to the Item constructor so that the new Item object has the same values as the old one.

Since C# does not provide a copy constructor. we must provide it ourselves if we wish to add this feature to the class. A copy constructor is defined as follows:

```
public Item (Item item)
{
    code = item.code;
    price = item.price;
}
```

The copy constructor is invoked by instantiating an object of type **Item** and passing it the object to be copied. Example:

```
Item item2 = new Item (item1);
```

Now, **item2** is a copy of **item1**.

Destructors

A destructor is opposite to a constructor. It is a method called when an object is no more required. The name of the destructor is the same as the class name and is preceded by a tilde (~). Like constructors, a destructor has no return type. Example:

```
class Fun
{
    . . . .
    . . . .
    ~ Fun ( ) //No arguments
    {
    . . . .
    }
}
```

Note that the destructor takes no arguments.

The this reference

C# supports the keyword this which is a reference to the object that called the method. The this reference is available within all the member methods and always refers to the current instance. It is normally used to distinguish between local and instance variables that have the same name.

```
class Integers
{
    int x;
    int y;
    public void SetXY(int x, int y(
    {
        this.x = x;
        this.y = y;
    }
    . . . .
    . . . .
}
```

In the assignment statements, this.x and this.y refer to the class members named x and y whereas

Simple x and y refer to the parameters of the SetXY ( ) method.

Constant Members

C# permits declaration of data fields of a class as constants. This can be done using the modifier const.

Example:

public canst int size = 100;

The member size is assigned the value 100 at compile time and cannot be changed later. Any attempt to assign a value to it will result in compilation error. It also means its value must be set when it is defined. For instance,

public const int size;

is wrong and will cause a compilation error. The const members are implicitly static. The value is by definition constant and therefore only one copy of it is stored which is common for all objects of the class. This implies that we must access the const members using the class name as we do with the static members.

Although const members are implicitly static, we cannot declare them explicitly using static. For example, the statement

public static const int size = 100;

is wrong and will produce compile-time error.

Read-only members

There are situations where we would like to decide the values of a constant member at run-time. We may also like to have different constant values for different objects of the class. To overcome these shortcomings, C# provides another modifier known as readonly to be used with data members. This modifier is designed to set the value of the member using a constructor method, but cannot be modified later. The readonly members may be declared as either static fields or instance fields. When they are declared as instance fields, they can take different values with different objects. Consider the code below:

```
class Numbers
{
      public readonly int m;
      public static readonly int n;
      public Numbers ( int x )
      {
            m = x;
      }
      static Numbers ( )
      {
            n = 100;
      }
}
```

The value for m is provided at the time of creation of an object using the constructor with parameter x. This value will remain constant for that object. Remember, the variable n is assigned a value of 100, even before the creation of any objects of Numbers.

Properties

One of the design goals of object-oriented systems is not to permit any direct access to data members because of the implications of integrity. It is normal practice to provide special methods known as accessor methods to have access to data members. We must use only these methods to set or retrieve the values of these members.

Accessing private data using accessor methods

```
using System;
class Number
{
    private int number;
    public void SetNumber( int x )    //accessor method
    {
        number = x;                   //private number accessible
    }
    public int GetNumber( )           //accessor method
    {
        return number;
    }
}

class NumberTest
{
    public static void Main ( )
    {
        Number n = new Number ( );
        n.SetNumber (100);    // set value
        Console.WriteLine("Number = " + n.GerNumber( ));    // get value

        // n.number; //Error! Cannot access private data
    }
```

The drawback with this type of accessor method is users have to remember that they have to use accessor methods to work with data members.

To overcome this problem, C# provides a mechanism known as properties that has the same capabilities as accessor methods but simple to use. Using a property, a programmer can get access to data members as though they are public fields.

Properties are class members, consists of two accessor methods get and set to modify the field value. The advantage of using a property is it allows a programmer to validate the client's request for any change in the value of a class field and may allow or reject such a request.

Implementing a property

```
using System;
class Number
{
      private int number;
      public int Anumber   // property
      {

            get
            {
                  return number;
            }
            set
            {
                  number = value;
            }
      }
}
class PropertyTest
{

       public void static Main ( )
       {
       Number n = new Number ( );
       n.Anumber = 100;
       int m = n.Anumber;
       Console.WriteLine("Number = " + m);
       }
}
```

Indexers

An indexer is a class member, which allows us to access member of a class as if it were an array.

```
public double this[ int idx ]
{
      get
      {
            //Return desired data
      }
      set
      {
            //Set desired data
      }
}
```

'this' keyword refers to the object instance. The return type determines what will be returned. The parameter inside the square brackets is used as the index.

Implementation of an indexer

```
using System;
using System.Collections;
class List
{
   ArrayList array = new ArrayList ( );
   public object this[ int index ]
   {
    get
    {
          if(index < 0  index >= array.Count)
          {
                return null;
          }
          else
          {
                return (array [index] );
          }
    }
    set
    {
          array[index] = value;
    }
   }
}

class IndexerTest
{
   public static void Main ( )
   {
    List list = new List ( );
    list [0] = "123";
    list [1] = "abc";
    list [2] = "xyz";
    for (int i = 0, i < list.Count; i++)
          Console.WriteLine( list[i]);
   }
}
```

Inheritance and Polymorphism

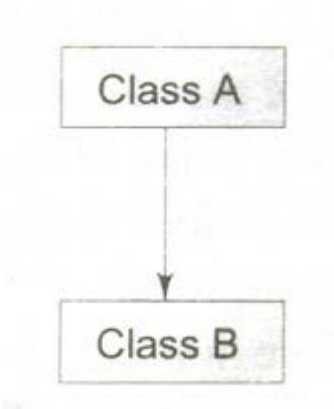
C# classes can be reused in several ways. Reusability is achieved by designing new classes, reusing all or some of the properties of existing ones. The mechanism of designing or constructing one class from another is called inheritance. This may be achieved in two different forms.

• Classical form

• Containment form

Classical form of inheritance

Inheritance represents a kind of relationship between two classes. Let us consider two classes A and B. We can create a class hierarchy such that B is derived from A as shown in figure.



Class A, the initial class that is used as the basis for the derived class is referred to as the base class, parent class or super class. Class B, the derived class, is referred to as derived class, child class or sub class. A derived class is a completely new class that incorporates all the data and methods of its base class. It can also have its own data and method members that are unique to itself. That is, it can enhance the content and behaviour of the base class.

We can now create objects of classes A and B independently. Example:

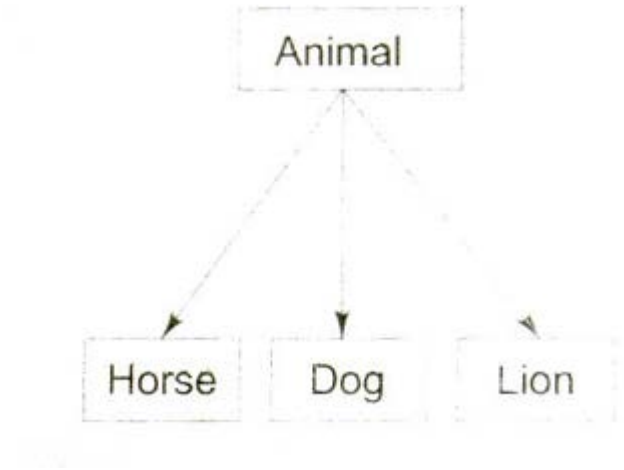
A a; // a is object of A

B b; // b is object of B

In such cases, we say that the object b is a type of a. Such relationship between a and b is referred to as 'is-a' relationship. Examples of is-a relationship are:

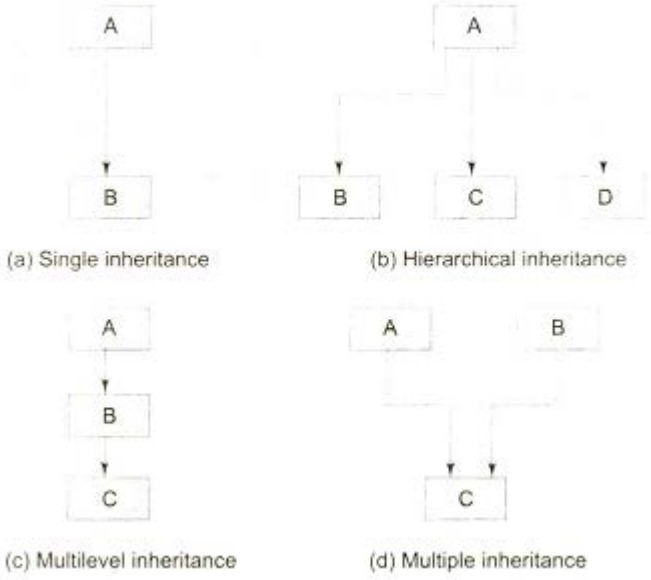
- Dog is-a type of animal
- Manager is-a type of employee
- Ford is-a type of car

The is-a relationship is illustrated in Fig.



Different types of relationship

- Single inheritance (only one base class)
- Multiple inheritance (several base classes)
- Hierarchical inheritance (one base class, many subclasses)
- Multilevel inheritance (derived from a derived class)



C# does not directly implement multiple inheritance. However, this concept is implemented using secondary inheritance path in the form of interfaces.

Containment Inheritance

We can also define another form of inheritance relationship known as containership between c1assA and B. Example:

```
class A
{
     . . . .
}
class B
{
     . . . .
     A a; // a is contained in b
}
B b;
. . . .
```

In such cases, we say that the object a is contained in the object b. This relationship between a and b is referred to as 'has-a' relationship. The outer class B which contains the inner class A is termed the 'parent' class and the contained class A is termed a 'child' class. Examples are:

- Car has-a radio
- House has-a store room
- City has-a road

The has-a relationship is illustrated in Fig.13.4.

Defining a subclass

A subclass is defined as follows:

```
Class subclass-name : baseclass-name

{
     variables declaration ;
     methods declaration ;

}
```

The definition is very similar to a normal class definition except for the use of colon : and baseclass-name. The colon signifies that the properties of the baseclass are extended to the subclass-name. When implemented the subclass will contain its own members as well those of the baseclass. This kind of situation occurs when we want to add more properties to an existing class without actually modifying it.

For example:

```
class Cylinder : Circle
{
     // Add additional fields
     // and methods here
}
```

Circle is the name of an existing class containing value of the radius and **Cylinder** is the name of derived class which could declare the length of the cylinder as an additional data member and a method to compute its volume.

Illustration of a simple inheritance

```
using System;
Class Item
{
    public void Company ( )        // base class
    {
        Console.WriteLine("Item Code = XXX");
    }
}

class Fan : Item                   // derived class
{
    public void Model ( )
    {
        Console.WriteLine("Fan Model : Classic");
    }
}

class SimpleInheritance
{
    public static void Main( )
    {
        Item item = new Item( ) ;
        Fan fan = new Fan( ) ;
        item.Company( ) ;
        fan.Company( ) ;
        fan.Model( ) ;
    }
}
```

Visibility Control

Class visibility is used to decide which parts of the system can create class objects.

A C# class can have one of the two visibility modifiers: public or internal. If we do not explicitly mark the visibility modifier of a class, it is implicitly set to 'internal'; that is, by default all classes are internal. Internal classes are accessible within the same program assembly and are not accessible from outside the assembly.

Classes marked public are accessible everywhere, both within and outside the program assembly.

Although classes are normally marked either public or internal, a class may also be marked private when it is declared as a member of another class. In such cases, the class behaves like a member of the enclosing class and therefore the modifiers applicable to class members are also applicable to it.

Class Members Visibility

As mentioned in the previous chapter, a class member can have any one of the five visibility modifiers:

- public
- protected
- private
- internal
- protected internal

Table 13.1 Visibility of class members

Keyword	*Visibility*			
	Containing classes	*Derived classes*	*Containing program*	*Anywhere outside the containing program*
Private	✓			
protected	✓	✓		
Internal	✓		✓	
protected internal	✓	✓	✓	
Public	✓	✓	✓	✓

It is important to remember that the accessibility domain of a member is never larger than that of the class containing it. Following table gives the visibility domain of members under different combinations of class access specifiers.

Table 13.2 Accessibility domain of class members

Member modifier	*Modifier of the containing class*		
	public	*internal*	*private*
public	Everywhere	only program	only class
internal	only program	only program	only class
private	only class	only class	only class

Defining subclass constructors

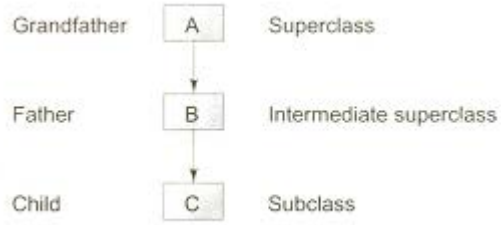
A subclass constructor is used to construct the instance variables of both the subclass and the superclass. The subclass constructor uses the keyword base to invoke the constructor method of the superclass.

```
public BedRoom (int x, int y, int z) : base (x, y)
{
    height = z;
}
```

Here .the derived constructor takes three arguments, the first two to provide values to the base constructor and the third one to provide value to its own class member. Note that base(x,y) behaves like a method call and therefore arguments are specified without types. The type and order of these arguments must match the type and order of base constructor arguments.

When the compiler encounters base(x,y), it passes the values x and y to the base class constructor which takes two int arguments. The base class constructor is called before the derived class constructor is executed. That is, the data members length and breadth are assigned values before the member height is assigned its value.

Multilevel Inheritance



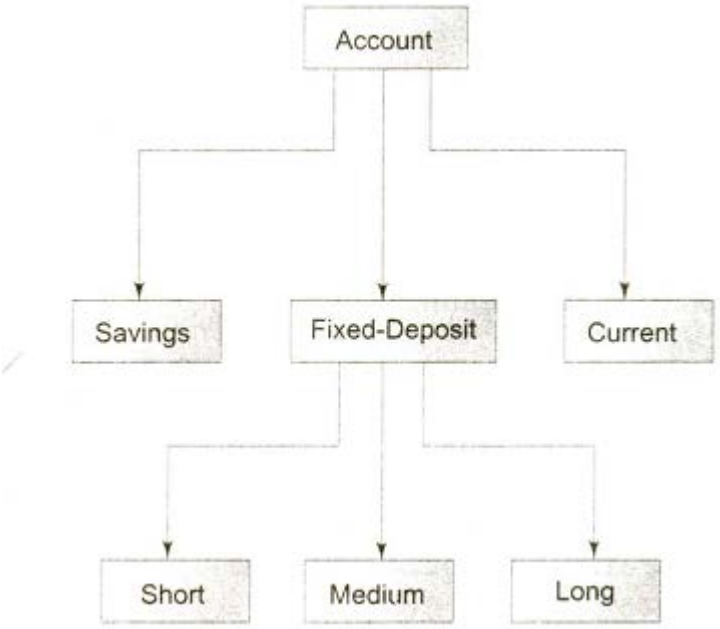
The class A serves as a base class for the derived class B which in turn serves as a base class for the derived class C. The chain ABC is known as inheritance path.

A derived class with multilevel base classes is declared as follows:

```
class A
{
     . . . .
     . . . .
}
class B : A // First level derivation
{
     . . . .
     . . . .
}
class C : B // Second level derivation
{
     . . . .
     . . . .
}
```

Hierarchical Inheritance

Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below the level. Following figure shows a hierarchical c1assification of accounts in a commercial bank. This is possible because all the accounts possess certain common features.



Overriding Methods

A method defined in a super class is inherited by its subclass and is used by the objects created by the subclass. Method inheritance enables us to define and use methods repeatedly in subclasses.

However, there may be occasions when we want an object to respond to the same method but behave differently when that method is called. That means, we should override the method defined in the superclass. This is possible by defining a method in the subclass that has the same name, same arguments and same return type as a method in the superclass. Then, when that method is called, the method defined in the subclass is invoked and executed instead of the one in the superclass, provided that:

• We specify the method in base class as virtual

• Implement the method in subclass using the keyword override.

This is known as overriding.

Illustration of method overriding

```
using System;
class Super                          //base class
{
    protected int x ;
    public Super (int x)
    {
          this.x = x ;


    }
    public virtual void Display () // method defined with virtual

    {
          Console.WriteLine ("Super x = " + x);
    }
}
class Sub : Super                    //derived class
{
    int y;
    public Sub (int x, int y) : base (x)
    {
          this.y = y;
    }
    public override void Display ( )    // method defined again
                                          //with override
    {
          Console.WriteLine("Super x = " + x) ;
          Console.WriteLine("Sub y = " + y);

    }
}
class OverrideTest
{
    public static void Main( )
    {
          Sub s1 = new Sub (100,200) ;
          s1.Display ( ) ;
    }
}
```

Hiding Methods

When we were overriding a base class method, we declared the base class method as virtual and the subclass method with the keyword override. This resulted in 'hiding' the base class method from the subclass.

Now, let us assume that we wish to derive from a class provided by someone else and we also want to redefine some methods contained in it. Here, we cannot declare the base class methods as virtual. Then, how do we override a method without declaring it virtual? This is possible in C#. We can use the modifier new to tell the compiler that the derived class method "hides" the base class method.

Hiding a base class method

```
using System;
class Base
{
     public void Display( )
     {
          Console.WriteLine("Base Method");
     }
}

class Derived : Base
{
     public new void Display( )      // hides base method
     {
          Console.WriteLine("Derive Method");
     }
}

class HideTest
{
     public static void Main( )
     {
          Derived d = new Derived( );
          d.Display ( );
     }
}
```

Abstract Classes

In a number of hierarchical applications, we would have one base class and a number of different derived classes. The top-most base class simply acts as a base for others and is not useful on its own. In such situations, we might not want anyone to create its objects. We can do this by making the base class abstract.

The abstract is a modifier and when used to declare a class indicates that the class cannot be instantiated. Only its derived classes (that are not marked abstract) can be instantiated. Example:

```
abstract class Base
{
     . . . .
}

class Derived : Base
{
     . . . .
}
. . . .
. . . .
. . . .
Base b1;        //Error
Derived d1;     //OK
```

Some characteristics of an abstract class are:

• It cannot be instantiated directly

• It can have abstract members

• We cannot apply a sealed modifier to it

Abstract methods

Similar to abstract classes, we can also create abstract methods. When an instance method declaration includes the modifier abstract, the method is said to be an abstract method.

An abstract method is implicitly a virtual method and does not provide any implementation. Therefore, an abstract method does not have method body. Example:

public abstract void Draw(int x, int y);

Note that the method body simply consists of a semicolon. Some characteristics of an abstract method are:

• It cannot have implementation

• Its implementation must be provided in non-abstract derived classes by overriding the method

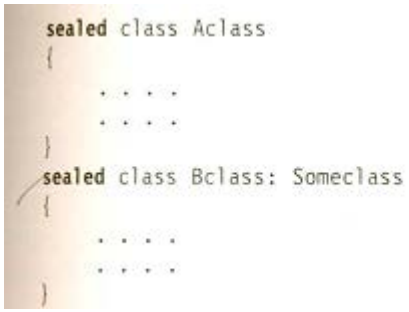
•It can be declared only in abstract classes.

• It cannot take either static or virtual modifiers.

• An abstract declaration is permitted to override a virtual method.

Sealed Classes: Preventing Inheritance

Sometimes, we may like to prevent a class being further subc1assed for security reasons. A class that cannot be subclassed is called a sealed class. This is achieved in C# using the modifier sealed as follows:



A sealed class cannot also be an abstract class.

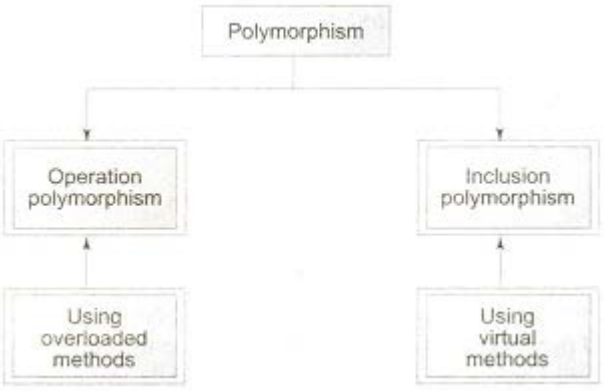
Sealed Methods

When an instance method declaration includes the sealed modifier, the method is said to be a sealed method. It means a derived class cannot override this method.

```
class A
{
     public virtual void Fun( )
     {
     . . . .
     }
}
class B : A
{
     public sealed override void Fun ( )
     {
     . . . .
     }
}
```

Polymorphism

Polymorphism means 'one name, many forms'. Essentially, polymorphism is the capability of one object to behave in multiple ways. Polymorphism can be achieved in two ways as shown in Figure. C# supports both of them.



Operation Polymorphism

Operation polymorphism is implemented using overloaded methods and operators. The overloaded methods are 'selected' for invoking by matching arguments, in terms of number, type and order. This information is known to the compiler at the time of compilation and, therefore, the compiler is able to select and bind the appropriate method to the object for a particular call at compile time itself. This process is called early binding, or static binding, or static linking. It is also known as compile time polymorphism.

Operation polymorphism

```
using System;
class Dog
{
}
class Cat
{
}
class Operation
{

     static void Call (Dog d)
     {


          Console.WriteLine ("Dog is called");
     }
     static void Call (Cat c)
     {
          Console.WriteLine (" Cat is called ");
     }

     public static void Main( )
     {
          Dog dog = new Dog( );
          Cat cat = new Cat ( );
          Call(dog); //invoking Call( )
          Call(cat); //again invoking Call( )
     }
}
```

Inclusion Polymorphism

Inclusion polymorphism is achieved through the use of virtual functions. Assume that the classA implements a virtual method M and classes B and C that are derived from A override the virtual melhod M. When B is cast to A, a call to the method M from A is dispatched to B. Similarly, when C is cast to A a call to M is dispatched to C. The decision on exactly which method to call is delayed until runtime and therefore, it is also known as runtime polymorphism. Since the method is linked with a particular class much later after compilation, this process is termed late binding. It is also known as dynamic binding because the selection of the appropriate method is done dynamically at runtime.

Inclusion polymorphism

```
using System;
class Maruthi
{
     public virtual void Display ( ) //virtual method
     {
          Console.WritLine("Maruthi car");
     }
}
class Esteem : Maruthi
{
     public override void Display( )
     {
          Console.WriteLine("Maruthi  Esteem");
     }
}
class Zen : Maruthi
{
     public override void Display ( )
     {
          Console.WriteLine("Maruthi  Zen");
     }
}
class Inclusion
{
     public static void Main( )
     {
          Maruthi m = new Maruthi ( );
          m = new Esteem ( );  //upcasting
          m.Display ( );
          m = new Zen ( );     //upcasting
          m.Display ( )
     }
}
```

Managing Errors and Exceptions

Errors are mistakes that can make a program go wrong. An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash. It is therefore important to detect and manage properly all the possible errors and error conditions in the program so that they do not terminate or cause the system to crash during execution.

Types of Errors

Errors may be broadly classified into two categories:

- Compile-time errors
- Run-time errors

Compile-Time Errors

All syntax errors will be detected and displayed by the C# compiler and therefore these errors are known as compile-time errors. Whenever the compiler displays an error, it will not create the executable file. It is therefore necessary that we fix all the errors before we can successfully compile and run the program.

- Missing semicolons
- Missing (or mismatch of) brackets in classes and methods
- Misspelling of identifiers and keywords
- Missing double quotes in strings
- Use of undeclared variables
- Incompatible types in assignments / initialization

Run-time Errors

Sometimes, a program may compile successfully creating the .exe file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow. Most common run-time errors are:

- Dividing an integer by zero
- Accessing an element that is out of the bounds of an array.
- Trying to store a value into an array of an incompatible class or type.
- Passing a parameter that is not in a valid range or value for a method.
- Attempting to use a negative size for an array.

• Converting an invalid string to a number

• Accessing a character that is out of bounds of a string

Exceptions

An exception is a condition that is caused by a run-time error in the program. When the C# compiler encounters an error such as dividing an integer by zero, it creates an exception object and throws it to inform us that an error has occurred.

If the exception object is not caught and handled properly, the compiler will display an error message and will terminate the program. If we want the program to continue with the execution of the remaining code then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as exception handling.

Steps in exception handling :

• Find the problem (Hit the exception)

• Inform that an error has occurred (Throw the exception)

• Receive the error information (Catch the exception)

• Take corrective actions (Handle the exception)

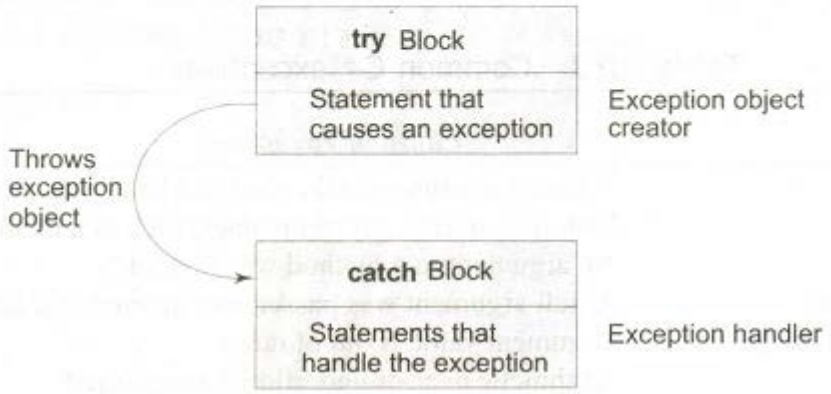
Table 18.1 Common C# exceptions

Exception Class	*Cause of Exception*
SystemException	A failed run-time check; used as a base class for other exceptions
AccessException	Failure to access a type member, such as a method or field
ArgumentException	An argument to a method was invalid.
ArgumentNullException	A null argument was passed to a method that does not accept it
ArgumentOutofRangeException	Argument value is out of range
ArithmeticException	Arithmetic over-or underflow has occurred
ArrayTypeMismatchException	Attempt to store the wrong type of object in an array
BadImageFormatException	Image is in the wrong format
CoreException	Base class for exceptions thrown by the runtime
DivideByZeroException	An attempt was made to divide by zero
FormatException	The format of an argument is wrong
IndexOutofRangeException	An array index is out of bounds
InvalidCastException	An attempt was made to cast to an invalid class
InvalidOperationException	A method was called at an invalid time
MissingMemberException	An invalid version of a DLL was accessed
NotFiniteNumberException	A number is not valid
NotSupportedException	Indicates that a method is not implemented by a class
NullReferenceException	Attempt to use an unassigned reference
OutofMemoryException	Not enough memory to continue execution
StackOverflowException	A stack has overflowed.

Syntax of exception handling

C# uses a keyword try to preface a block of code that is likely to cause an error condition and 'throw' an exception. A catch block defined by the keyword catch 'catches' the exception 'thrown' by the try block and handles it appropriately. The catch block is added immediately after the try block. The following example illustrates the use of simple try and catch statements:

```
.....
.....
try

  {
       statement;      // generates an exception
  }
  catch (Exception e)
  {
       statement;      // processes the exception
  }
  .....
  .....
```



Using try and catch for exception handling

```
using System;
class Error3
{
     public static void Main( )
     {
          int a = 10;
          int b = 5;
          int c = 5;
          int x, y ;

          try
          {
               x = a / (b-c);  / / Exception here
          }

          catch (Exception e)
          {
               Console.WriteLine("Division by zero");
          }

          y = a / (b+c);
          Console.WriteLine("y = " + y);
     }
}
```

MULTIPLE CATCH STATEMENTS

It is possible to have more than one catch statement in the catch block as illustrated below

```
.....
.....
try
{
          statement ;              / / generates an exception
}

catch (Exception-Type-1 e)
{
          statement;               / / processes exception type 1
}

catch (Exception-Type-2 e)
{
          statement;               / / processes exception type 2
}
.
.
.
catch (Exception-type-N e)
{
          statement ;              / / processes exception type N
 }
.....
.....
```

Using multiple catch blocks

```
using System;
class Error4
{
     public static void Main( )
     {
          int [ ] a = {5,10};
          int b = 5;

          try
          {
               int x = a[2] / b - a[1];
          }

          catch(ArithmeticException e)
          {
               Console.WriteLine("Division by zero");
          }

          catch(IndexOutOfRangeException e)
          {
               Console.WriteLine("Array index error");
          }

          catch(ArrayTypeMismatchException e)
          {
               Console.WriteLine("Wrong data type");
          }
          int y = a[1] / a[0];
          Console.WriteLine("y = " + y);
     }
}
```

The exception hierarchy

All C# exceptions are derived from the class Exception. When an exception occurs, the proper catch handler is determined by matching the type of exception to the name of the exception mentioned. If we are going to catch exceptions at different levels in the hierarchy, we need to put them in the right order. The rule is that we must always put the handlers for the most derived exception class first. Consider the following code :

```
. . . .
. . . .
try
{
. . .//throw Divide by Zero Exception
{
catch(Exception e)
{
. . . .
}
catch (DivideByZeroException e)
{
. . . .
}
```

This code will generate a compiler error, because the exception is caught by the first catch (which is a more general one) and the second catch is therefore unreachable. In C#, having unreachable code is always an error. The code must be rewritten as follows:

```
try
{
. . . . //throw Divide By Zero Exception
}
catch(DivideByZeroException e)
{
. . . .
}
catch(Exception e)
{
. . . .
}
```

General catch handler

A catch block which will catch any exception is called a general catch handler. A general catch handler does not specify any parameter and can be written as:

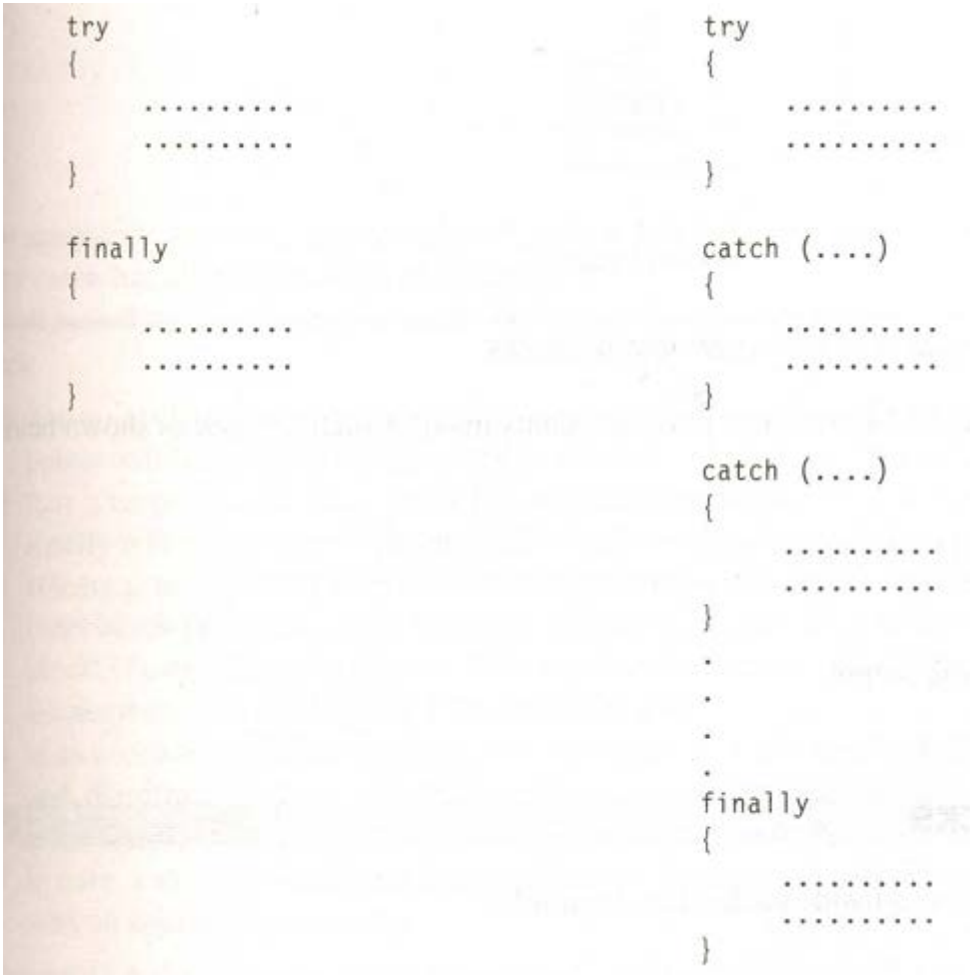
```
try
{
    . . . . //causes an exception

}
catch       //no parameters
{
    . . . . //handles error
    . . . .
```

Note that catch (Exception e) can handle all the exceptions thrown by the C# code and therefore can be used as a general catch handler. However, if the program uses libraries written in other languages, then there may be an exception that is not derived from the

class Exception. Such exceptions can be handled by the parameter-less catch statement. This handler is always placed at the end. Since there is no parameter, it does not catch any information about the exception and therefore we do not know what went wrong.

Using finally statement

C# supports another statement known as a finally statement that can be used to handle an exception that is not caught by any of the previous catch statements. A finally block can be used to handle any exception generated within a try block. It may be added immediately after the try block or after the last catch block shown as follows:



```
try
{
     ..........
     ..........
}

finally
{
     ..........
     ..........
}
```

```
try
{
     ..........
     ..........
}

catch (....)
{
     ..........
     ..........
}

catch (....)
{
     ..........
     ..........
}
.
.
.
.
finally
{
     ..........
     ..........
}
```

Throwing our own exceptions

There may be times when we would like to throw our own exceptions. We can do this by using the keyword throw as follows:

throw new Throwable_subclass;

Throwing our own exception

```
using System;
class MyException:Exception
{
    public MyException (string message) : base (message)
    {
    }

    public MyException ( )
    {
    }
    public MyException (string message, Exception inner)
        : base (message, inner)
    {
    }
}

class TestMyException
{
    public static void Main( )
    {
        int x = 5, y = 1000;

        try
        {
            float z = (float) x / (float) y ;
            if(z < 0.01)
            {
                throw new MyException("Number is too small");
            }
        }

        catch (MyException   e)
        {
            Console.WriteLine("Caught my exception");
            Console.WriteLine(e.Message);
        }

        finally
        {
            Console.WriteLine("I am always here");
        }
    }
}
```

Checked and Unchecked operations

Stack overflows are usual problems during arithmetic operations and conversion of integer types. C# supports two operators, checked and unchecked, which can be used for checking (or unchecking) stack overflows during program execution. If an operation is checked, then an exception will be thrown if overflow occurs. If it is not checked, no exception will be raised but we will lose data. For example, consider the code

```
int a = 200000
int b = 300000
try
{
     int m = checked ( a * b);
}
catch (OverflowException e)
{
     Console.WriteLine (e);
}
```

Since a*b produces a value that will easily exceed the maximum value for an int, an overflow occurs. As the operation is checked with operator checked, an overflow exception will be thrown. In this case we will get output like this:

```
System.OverflowException : An exception
Of type System.OverflowException was
thrown at…….
```

If we want to suppress the overflow checking, we can mark the code as unchecked

int n = unchecked (a * b);

In this case, no exception will be raised, but we will lose data.

Interfaces

C# does not support multiple inheritance. That is, classes in C# cannot have more than one superclass. For instance

```
Class A : B, C
{
        . . . .
        . . . .
}
```

is not permitted in C#. However, the designers of C# could not overlook the importance of multiple inheritance. A large number of real-life applications require the use of multiple inheritance whereby we inherit methods and properties from several distinct classes.

C# provides an alternate approach known as interface to support the concept of multiple inheritance. Although a C# class cannot be a subclass of more than one superclass, it can implement more than one interface, thereby enabling us to create classes that build upon other classes without the problems created by multiple inheritance.

An interface in C# is a reference type. It is basically a kind of class with some differences. Major differences include:

All the members of an interface are implicitly public and abstract.

• An interface cannot contain constant fields, constructors and destructors.

• Its members cannot be declared static.

• Since the methods in an interface are abstract, they do not include implementation code.

• An interface can inherit multiple interfaces

Defining an Interface

An interface can contain one or more methods, properties, indexers and events but none of them are implemented in the interface itself. It is the responsibility of the class that implements the interface to define the code for implementation of these members.

The syntax for defining an interface is very similar to that used for defining a class. The general form of an interface definition is:

```
interface InterfaceName
{
    Member declarations;
}
```

Here, interface is the keyword and InterfaceName is a valid C# identifier. Remember declarations will contain only a list of members without implementation code. Given below is a simple interface that defines a single method:

```
interface Show
{
     void Display ( );     // Note semicolon here
}
```

In addition to methods, interfaces can declare properties, indexers and events. Example:

```
interface Example
{
     int Aproperty
     {
          get ;
     }
     event someEvent Changed;
     void Display ( );
}
```

Extending an interface

Like classes, interfaces can also be extended. That is, an interface can be subinterfaced from other interfaces. The new subinterface will inherit all the members of the superinterface in the manner similar to subclasses. This is achieved as follows:

```
interface name2 : name1
{

     Members of name2
}
```

For example, we can put all members of particular behaviour category in one interface and the members of another category in the other.

Consider the code below:

```
interace Addition
{
     int Add (int x, int y) ;
}

interface Compute : Addition
{
     int Sub (int x, int y);
}
```

We can also combine several interfaces together into a single interface. Following declarations are valid:

```
interface I1
{
     . . . .
}
interface I2
{
     . . . .
}
interface I3 : I1, I2 //multiple inheritance
{
     . . . .
}
```

Implementing Interfaces

Interfaces are used as 'superclasses' whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows:

```
class classname : interfacename
{
    body of classname
}
```

Here the class classname 'implements' the interface interfacename. A more general form of implementation may look like this:

```
class classname : superclass, interface1, interface2. . . .
{
    body of classname
}
```

Implementation of multiple interfaces

```
using System;
interface Addition
{
      int Add ( );
}
interface Multiplication
{
      int Mul ( );
}
class Computation  : Addition, Multiplication
{

      int x, y;

      public Computation (int x, int y )        //Constructor
      {
                this.x = x;
                this.y = y;
      }
      public int Add ( )                        //Implement Add ( )
      {
                return ( x + y );
      }
      public int Mul ( )                        //Implement Mul ( )
      {
                return ( x * y );
      }
}

class InterfaceTest1
{
      public static void Main( )
      {
                Computation com = new Computation (10,20);

                Addition add = (Addition ) com;                  // casting
                Console.WriteLine ("Sum = " + add.Add ( ));

                Multiplication mul = (Multiplication) com;       // casting
                Console.WriteLine("Product = " + mul.Mul ( ) );
      }
}
```

Multiple implementation of an interface

```
using System;
interface Area
{
      double Compute ( double x );
}
class Square   :  Area


{
      public double Compute (double x)
      {
           return (x * x)
      }
}
class Circle : Area
{
      public double Compute (double x)
      {
           return (Math.PI * x * x);
      }
}
class InterfaceTest2
{


      public static void Main( )
      {
                'Square sqr = new Square ( );
                Circle cir = new Circle ( );

                Area area;
                area = sqr as Area;      //casting
                Console.WriteLine ("Area of Square ="
                           + area.Compute(10.0));

                area = cir as Area;      //casting

                Console.WriteLine ("Area of Circle="
                          + area.Compute(10.0));
      }
}
```

Abstract class and interface

Like any other class, an abstract class can use an interface in the base class list. However, the interface methods are implemented as abstract methods. Example:

```
interface A
{
     void Method ( );
}
abstract class B : A
{
     . . . .
     . . . .
     public abstract void Method ( );
}
```

Note that the class B does not implement the interface method; it simply redeclares as a public abstract method. It is the duty of the class that derives from B to override and implement the method.

Note that interfaces are similar to abstract classes. In fact, we can convert an interface into an abstract class. Consider the following interface:

```
interface A
{
     void Print ( );
}
```

This is identical to the following abstract class:

```
abstract class B
{
     abstract public void Print ( );
}
```

Now, a class can inherit from B instead of implementing the interface A. However, the class that inherits B cannot inherit any other class directly. If it is an interface, then the class can not only implement the interface but also use another class as base class thus implementing in effect multiple inheritance.

Delegates and Events

A delegate is a userdefined reference type contains a reference to the methods in a class. This reference can be changed dynamically at run time as desired. Using delegates the method to be called at runtime can be identified. Executing a delegate will in turn execute the method that it references.

Consider an example of a coffee vending machine, which dispenses different flavors of coffee, such as cappuccino and black coffee. On selecting the desired flavor of coffee, the vending machine decides to dispense the ingredients, such as milk powder, coffee powder, hot water, cappuccino coffee powder. All the materials are placed in different containers inside the vending machine. The required material is dispensed when you select a flavor.

Suppose, you select black coffee, the vending machine will call methods to dispense hot water and coffee powder only. The reference to these methods is made dynamically, when you press the required button to dispense black coffee.

To implement delegates in your application you need to declare delegates, instantiate delegates and use delegates.

Declaring Delegates

The methods that can be referenced by a delegate are determined by the delegate declaration. The delegate can refer to the methods, which have the same signature as that of the delegate**.**

The following syntax is used for delegate declaration:

```
delegate <return type><delegate-name><parameter list>
```

Consider the following example of delegate declaration:

```
public delegate void MyDelegate (string s);
```

Instantiating Delegates

Instantiating delegates means making it point (or refer) to some method. We can instantiate delegate by creating delegate object of the delegate type and assign the address of the required method to the delegate object.

```
public void DelegateFunction(string PassValue)
{
    // Method implementation Here
}
    //Delegate Declaration
    public delegate void MyDelegate(string ArgValue);

public void UseMethod()
{
    //Delegate Instantiation
    MyDelegate DelegateObject = new MyDelegate(DelegateFunction);
}
```

Using delegate

Using delegate means invoking a method using delegates. A delegate is called in a manner similar to calling a method.

The following example shows how to use a delegate:

```
// This code is to print data to the output device, which is either a
file or a screen
using System;
using System.IO;

// Program to write the data to the console and file
namespace Chapter12_Ex1
{
public class PrintToDevice
  {
```

```
    //Creating the variables of Stream classes
    static FileStream FStream;
    static StreamWriter SWriter;
    //Defining a Delegate
    public delegate void PrintData(String s);

    //Method to print a string to the console
    public static void WriteConsole (string str)
    {
    Console.WriteLine("{0} Console",str);
    }

    //Method to print a string to a file
    public static void WriteFile (string s)
{
          //Initializing stream objects
          FStream  = new FileStream("c:\\StoreData.txt",
          FileMode.Append, FileAccess.Write);
          SWriter = new StreamWriter(FStream);
          s= s + " File";
          //Writing a string to the file
          SWriter.WriteLine(s);
          //removing the content from the buffer
          SWriter.Flush();
          SWriter.Close();
          FStream.Close();
}
     //Method to send the string data to respective methods
    public static void DisplayData(PrintData PMethod)
{
    PMethod("This should go to the");
}
    public static void Main()
    {
          //Initializing the Delegate object

          PrintData Cn = new PrintData (WriteConsole);
          PrintData Fl = new PrintData (WriteFile);
          //Invoking the DisplayData method with the Delegate
          object as the argument
          //Using Delegate
          DisplayData (Cn);
          DisplayData (Fl);
          Console.ReadLine();
    }
}
}
```

Types of Delegates

There are two types of delegates, Single-cast delegate and Multicast delegate. A Single-cast delegate can call only one method at a time, whereas a Multicast delegate can call multiple methods at the same time.

The following code shows how to use a multicast delegate

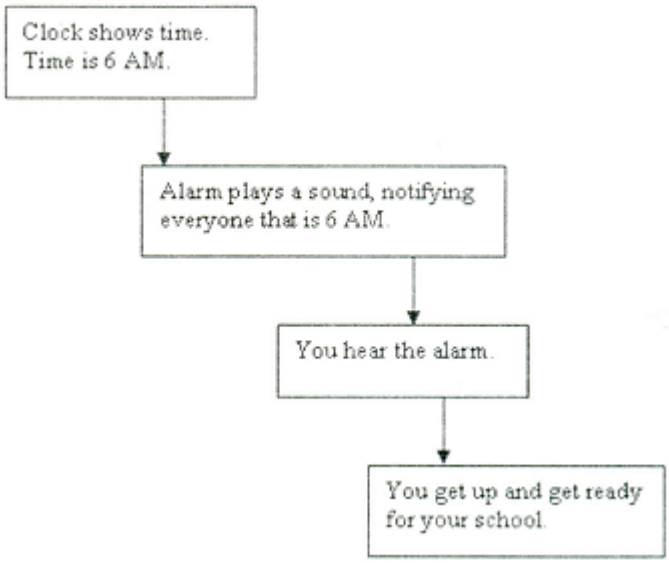
```
using System;
using System.IO;

// Program to write the data to the console and file
namespace Chapter12_Ex2
{
      public class PrintToDevice
      {
            //Creating the variables of Stream classes
            static FileStream FStream;
            static StreamWriter SWriter;
            //Defining a Delegate
            public delegate void PrintData(String s);

            //Method to print a string to the console
            public static void WriteConsole (String str)
            {
                  Console.WriteLine("{0} Console",str);
            }

            //Method to print a string to a file
            public static void WriteFile (String s)
            {
                  //Initializing stream objects
                  FStream = new FileStream("c:\\StoreData.txt",
                  FileMode.Append, FileAccess.Write);
                  SWriter = new StreamWriter(FStream);
                  s= s + " File";
                  //Writing a string to the file
                  SWriter.WriteLine(s);
                  //removing the content from the buffer
                  SWriter.Flush();
                  SWriter.Close();
                  FStream.Close();
            }
            //Method to send the string data to respective methods
            public static void DisplayData(PrintData PMethod)
            {
                  PMethod("This should go to the");
            }
            public static void Main()
            {
                  //Initializing the Delegate object
                  PrintData MlDelegate = new PrintData(WriteConsole);
                  MlDelegate += new PrintData(WriteFile);
                  DisplayData(MlDelegate);
                  MlDelegate -= new PrintData(WriteFile);
                  DisplayData(MlDelegate);
            }

      }
```

Events

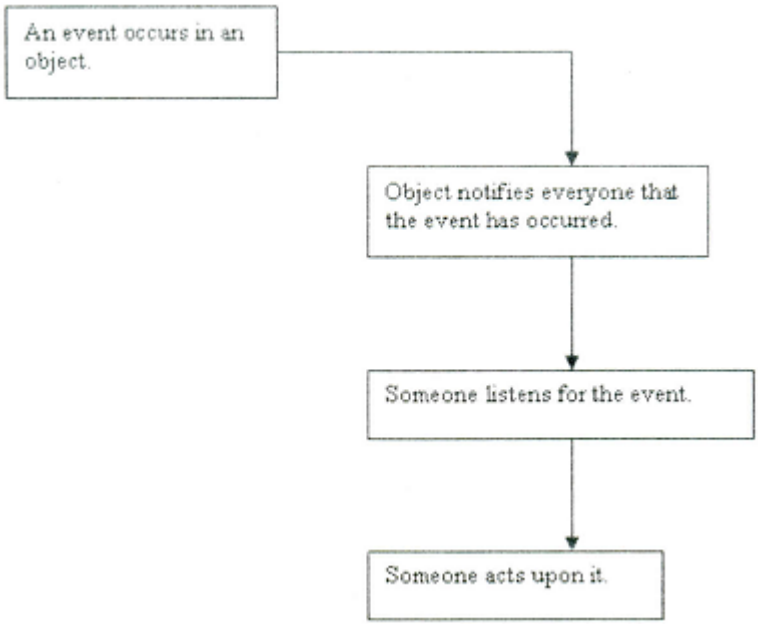
An event is an action or occurrence, such as clicks, key presses, mouse movements, or system generated notifications. Applications can respond to events when they occur. An example of a notification is interrupts. Events are messages sent by the object to indicate the occurrence of the event. Events are an effective mean of inter-process communication.

Consider an example of an event and the response to the event. A clock is an object that shows 6 AM time and generates an event in the form of an alarm. You accept the alarm event and act accordingly.

The following figure shows the alarm event and handling of the event



The following figure is the generalized representation that explains events and event handling.



In C#, delegates are used with events to implement event handling.

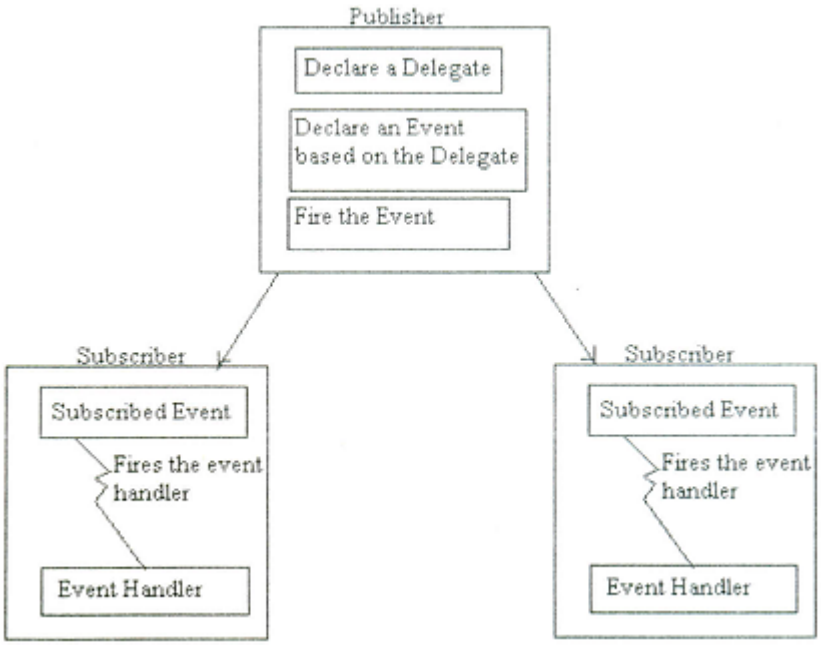
Using Delegates with Events

The events are declared and raised in a class and associated with the event handlers using delegates within the same class or other classes. Events are part of a class and the same class is used to publish its events. The other classes can, however, accept these events or in other words can subscribe to these events. Events use the publisher and subscriber model.

A publisher is an object that contains the definition of the event and the delegate. The association of the event with the delegate is also specified in the publisher class. The object of the publisher class invokes the event, which is notified to the other objects

A subscriber is an object that wants to accept the event and provide a handler to the event. The delegate of the publisher class invokes the method of the subscriber class. This method in the subscriber class is the event handler. The publisher and subscriber model implementation can be defined by the same class.

The following figure shows the mechanism used by the publisher and subscriber objects



Defining an Event

The definition of the event in a publisher class includes the declaration of the delegate as well as the declaration of the event based on the delegate. The following code defines a delegate named TimeToRise and an event named RingAlarm, which invokes the TimeToRise delegate when it is raised:

```
public delegate void TimeToRise();
private event TimeToRise RingAlarm;
```

Subscribing to an Event

The event of the publisher class needs to be associated with its handler. The event handler method is associated with the event using the delegate. When the publisher object raises the event, the subscribing object associates the method, which needs to be called.

Consider a class named Student which contains a method named WakeUp (). The requirement states that the method WakeUp() should be called at 6 AM. The requirement could be implemented using events. The following code shows how the Student class subscribes to the event named TimeToRi se:

```
Student PD= new Student();
RingAlarm = new TimeToRise(PD.WakeUp);
```

Notifying Subscribing objects

To notify all the objects that have subscribed to an event, the event needs to be raised. Raising an event is like calling a method.

if(condition is true)

RaiseEvent();

When even is raised it will invoke all the delegates of the objects that are subscribed to that particular event.

```
using System;

namespace Example_4

{
  /// <summary>
  /// The program demonstrates the use of Events.
  /// </summary>

  // First class
  class ClassA
  {
    public void DispMethod()
    {
      Console.WriteLine("Class A has been notified of NotifyEveryOne Event!");
    }
  }
  // Second class
  class ClassB
  {
    public void DispMethod()
    {
      Console.WriteLine("Class B has been notified of NotifyEveryOne Event!");
    }
  }

  class TestEvents
  {
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
      // Object of delegate
      Delegate objDelegate = new Delegate();
      // Object of ClassA
      ClassA objClassA = new ClassA();
      // Object of ClassB
      ClassB objClassB = new ClassB();

      // Subscribing to the event
      objDelegate.NotifyEveryOne += new Delegate.MeDelegate(objClassA.
DispMethod);
      objDelegate.NotifyEveryOne += new Delegate.MeDelegate(objClassB.
DispMethod);

      // Invoking method that contains code to raise the event
      objDelegate.Notify();
```

```
    }
}

class Delegate
{
    // Defining a delegate
    public delegate void MeDelegate();
    // Defining an event
    public event MeDelegate NotifyEveryOne;

    public void Notify()
    {
        // If the event is not null
        if(NotifyEveryOne != null)
        {
            Console.WriteLine("Raise Event : ");
            // Raising the event
            NotifyEveryOne();
        }
    }
}
}
```