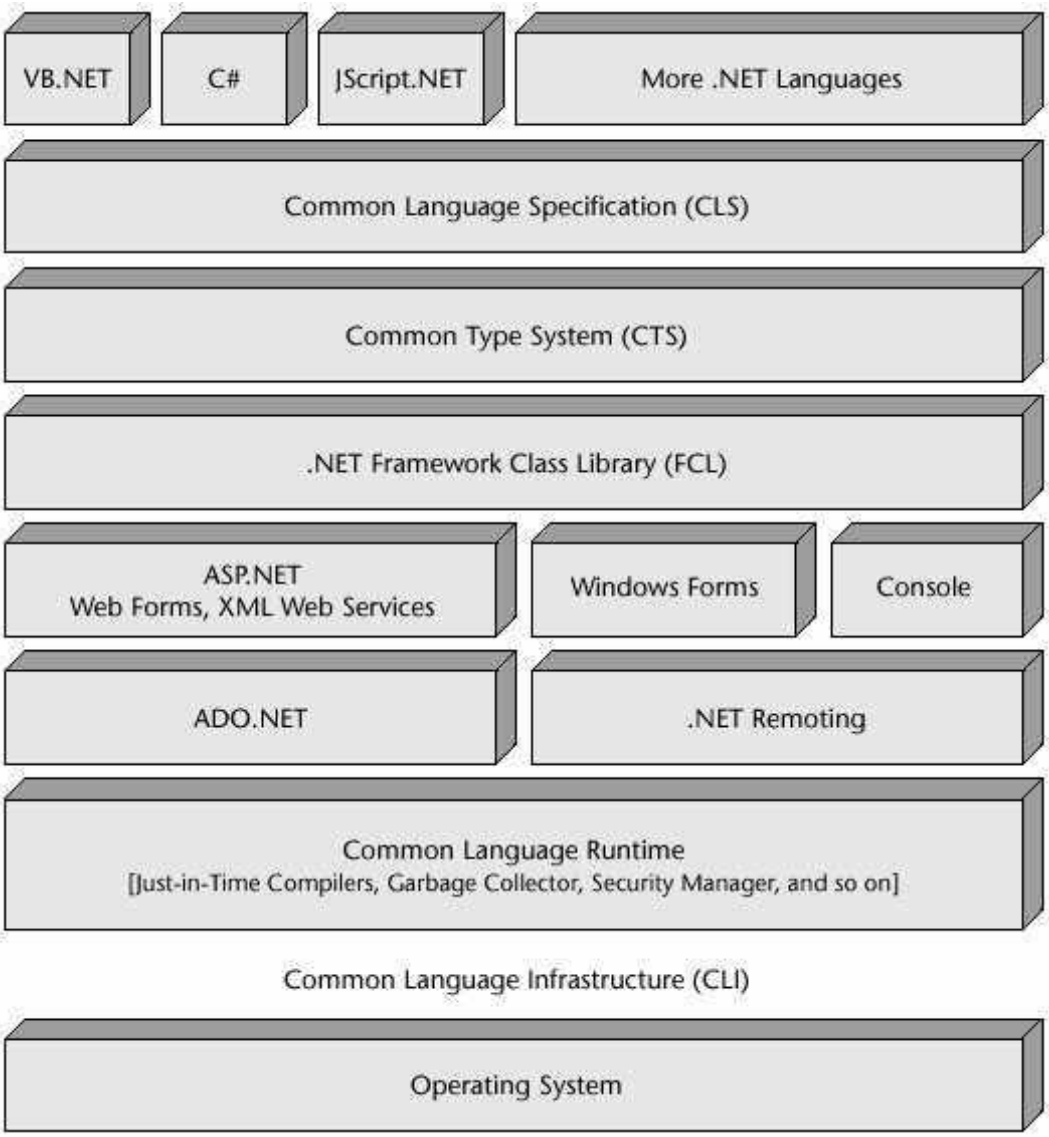
.NET architecture



Common Language Specification (CLS))

One of the obvious themes of .NET is unification and interoperability between various programming languages. In order to achieve this; certain rules must be laid and all the languages must follow these rules. In other words we can not have languages running around creating their own extensions and their own fancy new data types. CLS is the collection of the rules and constraints that every language (that seeks to achieve .NET compatibility) must follow.

The Common Language Specification is a set of basic language features (constructs and constraints) that serves as a guide for library writers and compiler writers. It allows libraries to be fully usable from any language supporting the CLS, and for those languages to integrate with each other. The Common Language Specification is a subset of the common type system. The CLS is actually a set of restrictions on the CTS. The CLS defines not only the types allowed in external calls, but the rules for using them, depending on the goal of the user.

Informally CLS is a simply a contract between programming language designers and class library authors. The CLS is basically just a subset of the entire set of features supported by the CLR. The CLS includes things such as calling virtual methods, overloading methods and does not include things such as unsigned types. CLS is weighted very heavily in favor of the library designers.

The Common Language Specification describes a common level of language functionality. The CLS is a set of rules that a language compiler must adhere to in order to create .NET applications that run in the CLR. Anyone who wants to write a .NET·-compliant compiler needs simply to adhere to these rules and that's it. The relatively high minimum bar of the CLS enables the creation of a club of CLS-compliant languages. Each member of this club enjoys dual benefits: complete access to .NET framework functionality and rich interoperability with other compliant languages. For example a VB class can inherit from a C# class and override its virtual methods.

The common language specification (CLS) is a collection of rules and restrictions that allow interoperation between languages. Even though the CLI does not require compilers to follow CLS, code that follows the CLS rules is compatible with all other languages that follow the CLS rules.

Microsoft has defined three level of CLS compatibility/compliance. The goals and objectives of each compliance level have been set aside. The three compliance levels with their brief description are given below:

Compliant producer

The component developed in this type of language can be used by any other language.

Consumer

The language in this category can use classes produced in any other language. In simple words this means that the language can instantiate classes developed in other language. This is similar to how COM components can be i:=1stantiated by your ASP code.

Extender

Languages in this category can not just use the classes as in CONSUMER category; but can also extend classes using inheritance.

Languages that come with Microsoft Visual Studio namely Visual C++, Visual Basic and

C#; all satisfy the above three categories. Vendors can select any of the above categories as the targeted compliance level(s) for their languages.

Common Type System (CTS)

The language interoperability, and .NET Class Framework, are not possible without all the language sharing the same data types. What this means is that an "int" should mean the same in VB, VC++, C# and all other .NET compliant languages. Same idea follows for all the other data types. This is achieved through introduction of Common Type System (CTS).

Common type system (CTS) is an important part of the runtimes support for cross language integration. The common type system performs the following functions:

• Establishes a framework that enables cross-language integration, type safety, and high performance code execution.

• Provides an object-oriented model that supports the complete implementation of many programming languages.

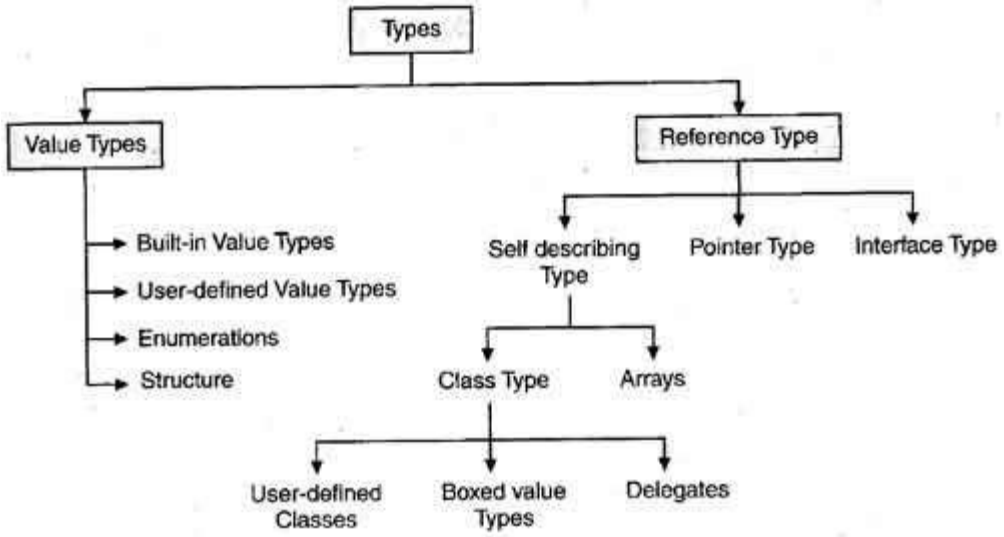
The common type system supports two general categories of types:

1. Value types

Value types directly contain their data, and instances of value types are either allocated on the stack or allocated inline in a structure. Value types can be built-in, user-defined or enumerations types.

2. Reference types

Reference types stores a reference to the value's memory address, and are allocated on the heap. Reference types can be self-describing types, pointers types, or interface types. The type of a reference type can be determined from values of self-describing types. Self-describing types are further split into arrays and class types are user-defined classes, boxed value types, and delegates.



Common type system

CTS, much like Java, define every data type as a Class. Every .NET compliant language must stick to this definition. Since CTS defines every data type as a class; this means that only Object-Oriented (or Object-Based) languages can achieve .NET compliance.

CLR

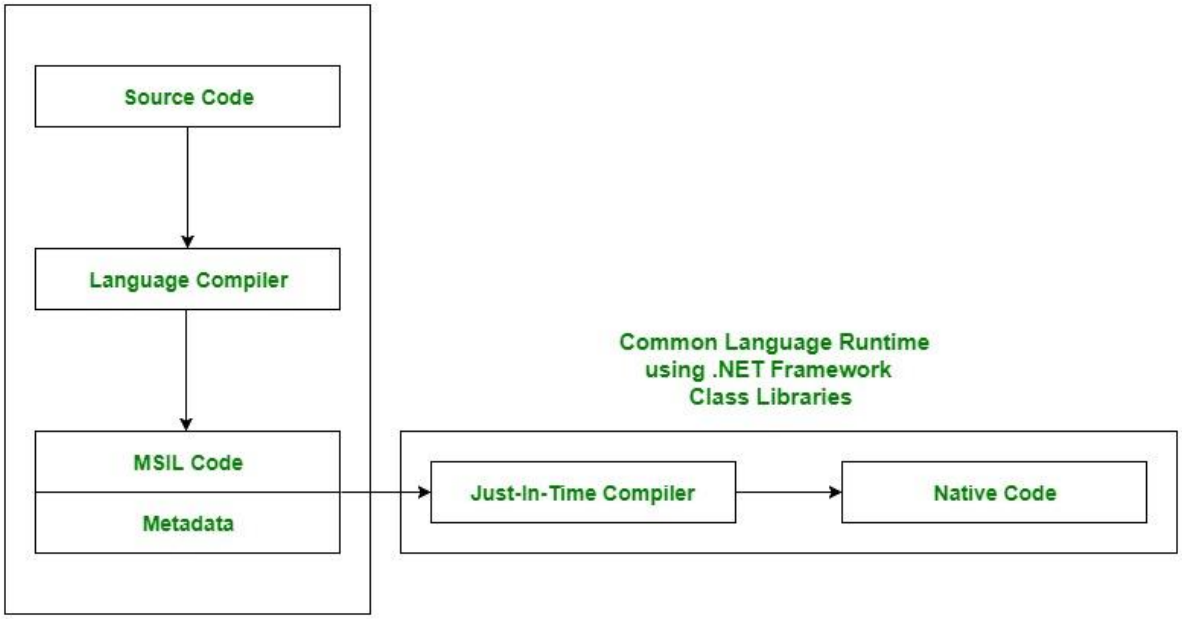
The CLR stands for Common Language Runtime is an Execution Environment . It works as a layer between Operating Systems and the applications written in .Net languages that conforms to the Common Language Specification (CLS). The main function of Common Language Runtime (CLR) is to convert the Managed Code into native code and then execute the Program. The Managed Code compiled only when it needed, that is it converts the appropriate instructions when each function is called . The Common Language Runtime (CLR) 's just in time (JIT) compilation converts Intermediate Language (MSIL) to native code on demand at application run time.

When .Net application is executed at that time control will go to Operating System,then Operating System Create a process to load CLR

The program used by the operating system for loading CLR is called runtime host,which are diffrent depending upon the type of application that is desktop or web based application

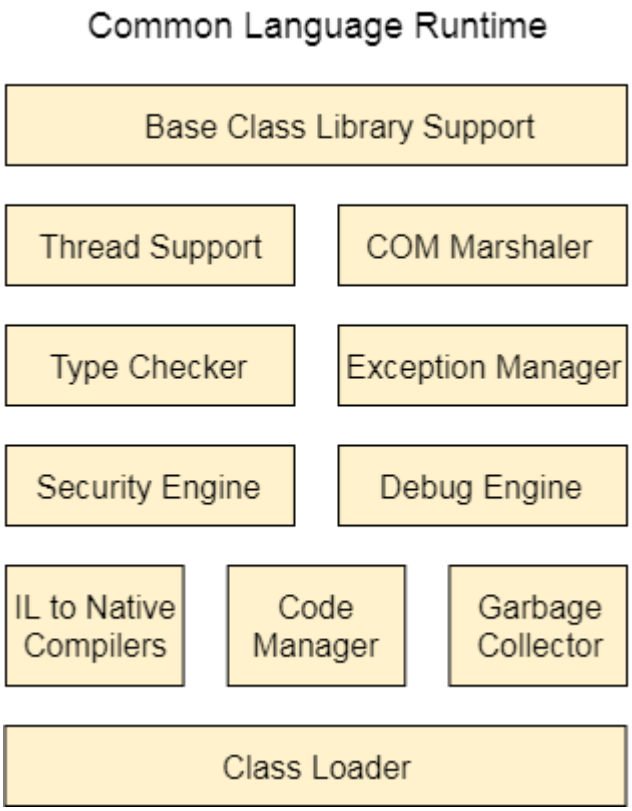
- Suppose you have written a C# program and save it in a file which is known as the Source Code.
- Language specific compiler compiles the source code into the *MSIL(Microsoft Intermediate Language)* which is also know as the *CIL(Common Intermediate Language)* or *IL(Intermediate Language)* along with its metadata. *Metadata* includes the all the types, actual implementation of each function of the program. MSIL is machine independent code.
- Now CLR comes into existence. CLR provides the services and runtime environment to the MSIL code. Internally CLR includes the JIT(Just-In-Time) compiler which converts the MSIL code to machine code which further executed by CPU. CLR also uses the .NET Framework

class libraries. Metadata provides information about the programming language, environment, version, and class libraries to the CLR by which CLR handles the MSIL code. As CLR is common so it allows an instance of a class that written in a different language to call a method of the class which written in another language.



.NET CLR Structure

Following is the component structure of Common Language Runtime.



Type Checker

Type checker will verify types used in the application with CTS or CLS standards supported by CLR, this provides type safety.

COM marshaller
COM marshaller will provide communication with COM component, this supports COM interoperability.

Debug Manager
Debug Manager service will activate debugger utility to support line by line execution , the developer can make changes as per requirement without terminating application execution.

Thread Support
Thread support will manage more than one execution path in applicaiton process. This provides multithreading support.

IL to Native compiler
IL to native compiler is called JIT compiler(just-in-time compiler) , this will convert IL code into operating System native code.

Exception Manager
Exception Manager will handle exceptions thrown by application by executing catch block provided by exception, it there is no catch block , it will terminate application.

Garbage Collector
Garbage Collector will release memory of unused objects, this provides automatic memory management.

Microsoft Intermediate Language (MSIL)

A .NET programming language (C#, VB.NET, J# etc.) does not compile into executable code; instead it compiles into an intermediate code called Microsoft Intermediate Language (MSIL). As a programmer one need not worry about the syntax of MSIL - since our source code in automatically converted to MSIL. The MSIL code is then send to the CLR (Common Language Runtime) that converts the code to machine language which is then run on the host machine.

MSIL is similar to Java Byte code. A Java program is compiled into Java Byte code (the .class file) by a Java compiler, the class file is then sent to JVM which converts it into the host machine language.

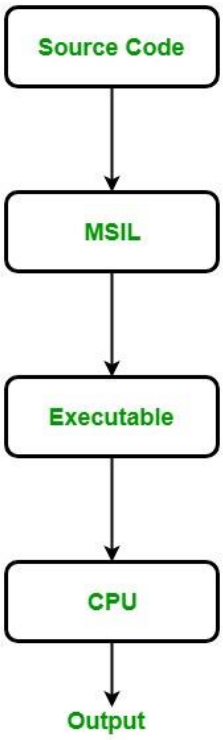
Managed code and Unmanaged code in .NET

What is managed code?

A code which is written to aimed to get the services of the managed runtime environment execution like CLR(Common Language Runtime) in .NET Framework is known as Managed Code. It always implemented by the managed runtime environment instead of directly executed by the operating system. The managed runtime environment provides different types of services like garbage collection, type checking, exception handling, bounds checking, etc. to code automatically without the interference of the programmer. It also provides memory allocation, type safety, etc to the code. The application is written in the languages like Java, C#, VB.Net, etc. are always aimed at runtime environment services to manage the execution and the code written in these types of languages are known as managed code.

In the case of .NET Framework, the compiler always compiles the manages code in the intermediate language(MSIL) and then create an executable. When the programmer runs the executable, then the Just In Time Compiler of CLR compiles the intermediate language in the native code which is specific to the underlying architecture. Here this process is taking place under

a managed runtime execution environment so this environment is responsible for the working of the code. The execution of managed code is as shown in the below image, the source code is written in any language of .NET Framework.



The managed code also provides platform independence because when the managed code compiled into the intermediate language, then the JIT compiler compiles this intermediate language in the architecture specific instruction.

What are the **advantages** of using Managed Code?

- It improves the security of the application like when you use runtime environment, it automatically checks the memory buffers to guard against buffer overflow.
- It implement the garbage collection automatically.
- It also provides runtime type checking/dynamic type checking.
- It also provides reference checking which means it checks whether the reference point to the valid object or not and also check they are not duplicate.

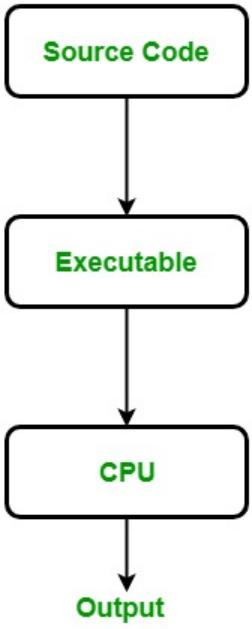
What are the **disadvantages** of Managed Code?

The main disadvantage of managed language is that you are not allowed to allocate memory directly, or you cannot get the low-level access of the CPU architecture.

What is Unmanaged code?

A code which is directly executed by the operating system is known as Unmanaged code. It always aimed for the processor architecture and depends upon computer architecture. When this code is compiled it always tends to get a specific architecture and always run on that platform, in other words, whenever you want to execute the same code for the different architecture you have to recompile that code again according to that architecture. It always compiles to the native code that is specific to the architecture.
In unmanaged code, the memory allocation, type safety, security, etc are managed by the developer. Due to this, there are several problems related to memory occur like buffer overflow, memory leak, pointer override, etc. The executable files of unmanaged code are generally in

binary images, x86 code which is directly loaded into memory. The application written in VB 6.0, C, C++, etc are always in unmanaged code. The execution of unmanaged code is as shown in the below image:



What are the **advantages** of using Unmanaged Code?

- It provides the low-level access to the programmer.
- It also provides direct access to the hardware.
- It allows the programmer to bypass some parameters and restriction that are used by the managed code framework.

What are the **disadvantages** of Unmanaged Code?

- It does not provide security to the application.
- Due to the access to memory allocation the issues related to memory occur like memory buffer overflow, etc.
- Error and exceptions are also handled by the programmer.
- It does not focus on garbage collection.

Difference between Managed and Unmanaged code in .NET

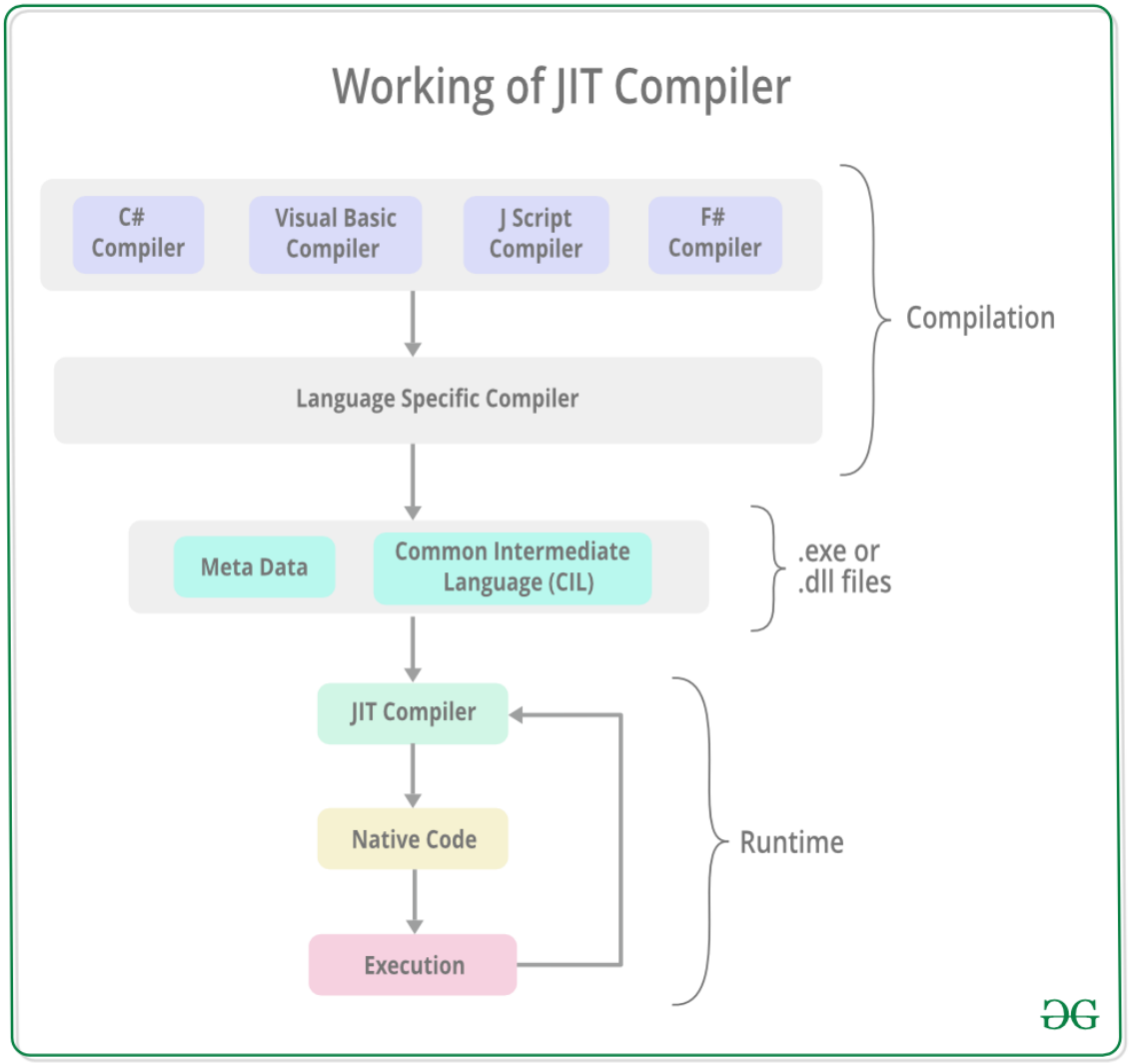
Managed code is the code which is managed by the CLR(Common Language Runtime) in *.NET Framework*. Whereas the Unmanaged code is the code which is directly executed by the operating system. Below are some important differences between the Managed code and Unmanaged code:

MANAGED CODE	UNMANAGED CODE
It is executed by managed runtime environment or managed by the CLR.	It is executed directly by the operating system.
It provides security to the application written in .NET Framework.	It does not provide any security to the application.
Memory buffer overflow does not occur.	Memory buffer overflow may occur.
It provide runtime services like Garbage Collection, exception handling, etc.	It does not provide runtime services like Garbage Collection, exception handling, etc.
The source code is complied in the intermideate language know as *IL or MSIL or CIL*.	The source code direclty compile into native langugae.
It does not provide low-level access to the prgrammer.	It provide low-level access to the prgrammer.

Just-In-Time(JIT) Compiler in .NET

Just-In-Time compiler(JIT) is a part of Common Language Runtime (CLR) in *.NET* which is responsible for managing the execution of *.NET* programs regardless of any *.NET* programming language. A language-specific compiler converts the source code to the intermediate language. This intermediate language is then converted into the machine code by the Just-In-Time (JIT) compiler. This machine code is specific to the computer environment that the JIT compiler runs on.

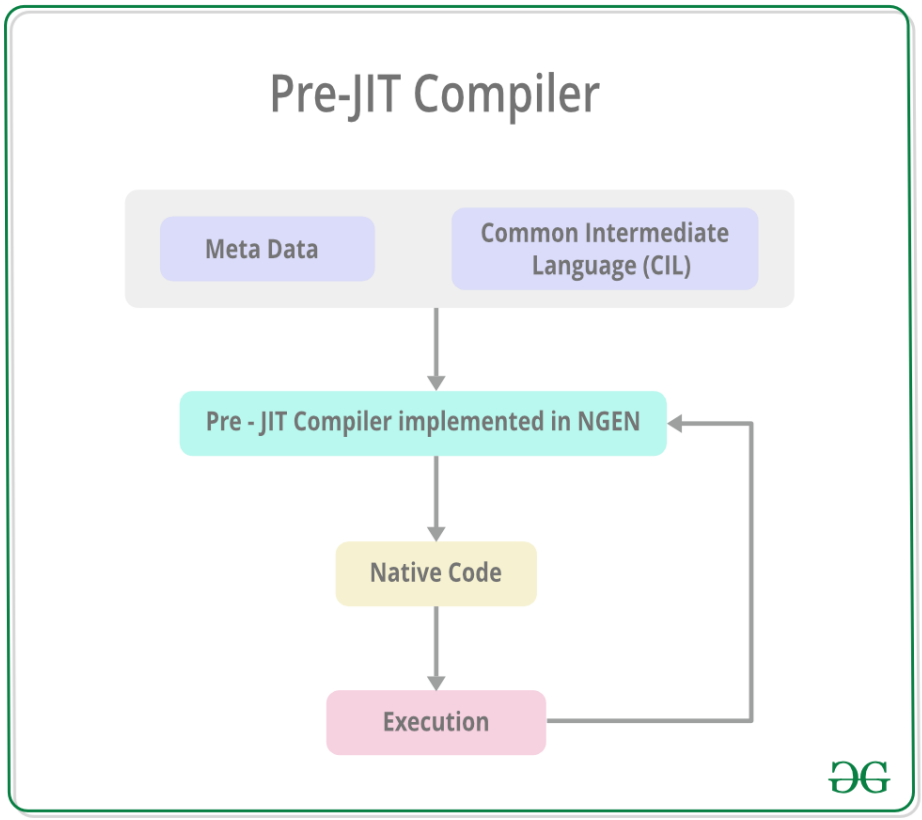
Working of JIT Compiler: The JIT compiler is required to speed up the code execution and provide support for multiple platforms. Its working is given as follows:



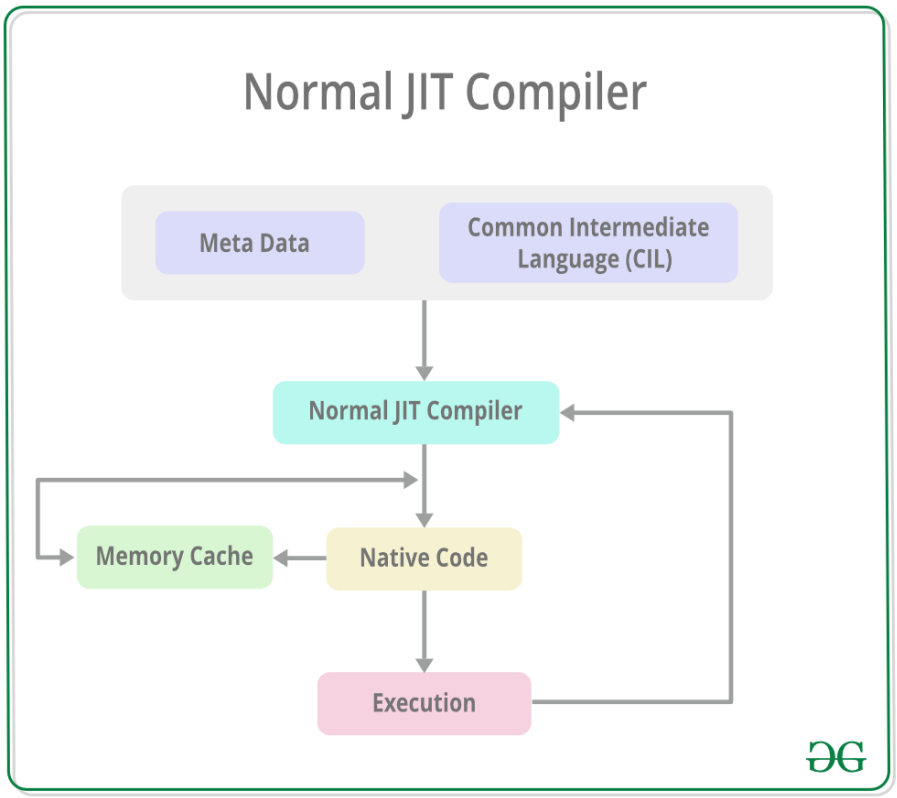
The JIT compiler converts the Microsoft Intermediate Language(MSIL) or Common Intermediate Language(CIL) into the machine code. This is done before the MSIL or CIL can be executed. The MSIL is converted into machine code on a requirement basis i.e. the JIT compiler compiles the MSIL or CIL as required rather than the whole of it. The compiled MSIL or CIL is stored so that it is available for subsequent calls if required.

Types of Just-In-Time Compiler: There are 3 types of JIT compilers which are as follows:

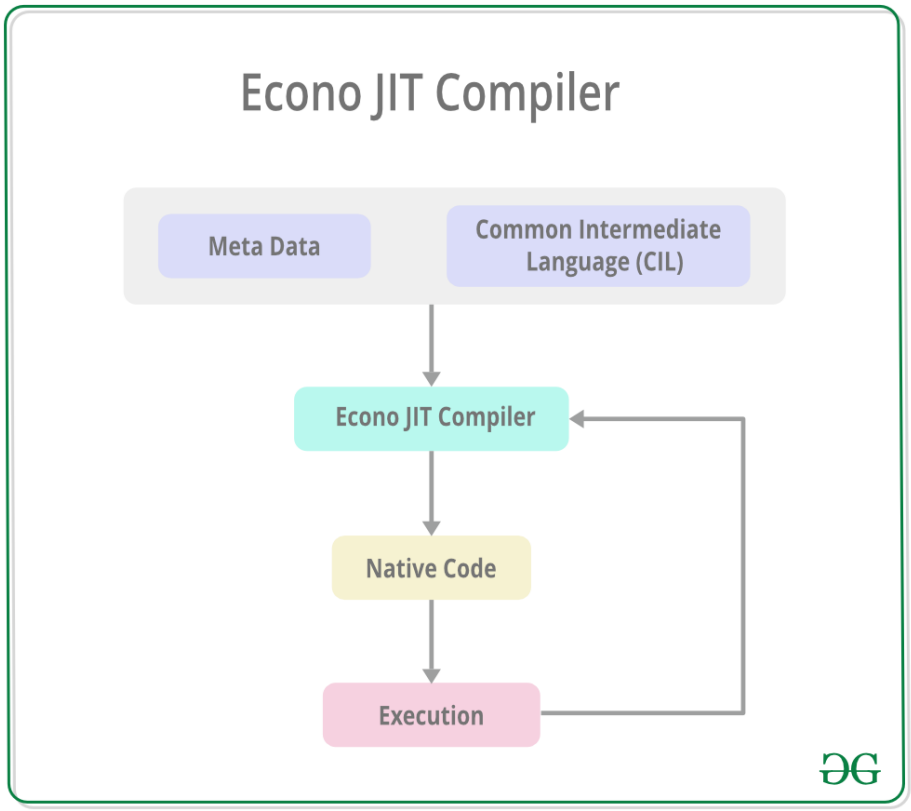
1. **Pre-JIT Compiler**: All the source code is compiled into the machine code at the same time in a single compilation cycle using the Pre-JIT Compiler. This compilation process is performed at application deployment time. And this compiler is always implemented in the *Ngen.exe (Native Image Generator)*.



2. **Normal JIT Compiler:** The source code methods that are required at run-time are compiled into machine code the first time they are called by the Normal JIT Compiler. After that, they are stored in the cache and used whenever they are called again.



3. **Econo JIT Compiler:** The source code methods that are required at run-time are compiled into machine code by the Econo JIT Compiler. After these methods are not required anymore, they are removed.



Advantages of JIT Compiler:

- The JIT compiler requires less memory usage as only the methods that are required at run-time are compiled into machine code by the JIT Compiler.
- Page faults are reduced by using the JIT compiler as the methods required together are most probably in the same memory page.
- Code optimization based on statistical analysis can be performed by the JIT compiler while the code is running.

Disadvantages of JIT compiler:

- The JIT compiler requires more startup time while the application is executed initially.
- The cache memory is heavily used by the JIT compiler to store the source code methods that are required **at run-time.**

Garbage collection

The Garbage collection is very important technique in the .Net framework to free the unused managed code objects in the memory and free the space to the process. I will explain about the basics of the Garbage collection in this article.

Garbage Collection in .Net framework

The garbage collection (GC) is new feature in Microsoft .net framework. When we have a class that represents an object in the runtime that allocates a memory space in the heap memory. All the behavior of that objects can be done in the allotted memory in the heap. Once the activities related to that object is get finished then it will be there as unused space in the memory.

The earlier releases of Microsoft products have used a method like once the process of that object get finished then it will be cleared from the memory. For instance Visual Basic, An object get finishes that work then there we have to define a "nothing" to that object. So, it clears the memory space to the processors.

Microsoft was planning to introduce a method that should automate the cleaning of unused memory space in the heap after the life time of that object. Eventually they have introduced a new technique "Garbage collection". It is very important part in the .Net framework. Now it handles this object clear in the memory implicitly. It overcomes the existing explicit unused memory space clearance.

Garbage Collection

The heap memory is divided into number of generations. Normally it is three generations. The Generation 0 is for short live objects, Generation 1 is for medium live objects which are moved from Generation 0. Generation 3 is mostly stable objects.

When an object is created then it will allocate the memory space which will be higher. It will be in the Generation 0 and the memory allocation will be continuous without any space between the generations of garbage collectors.

How Garbage Collection Works

Implicit Garbage Collection should be handled by the .Net framework. When object is created then it will be placed in the Generation 0. The garbage collection uses an algorithm which checks the objects in the generation, the objects life time get over then it will be removed from the memory. The two kinds of objects. One is Live Objects and Dead Objects. The Garbage collection algorithm collects all unused objects that are dead objects in the generation. If the live objects running for long time then based on that life time it will be moved to next generation.

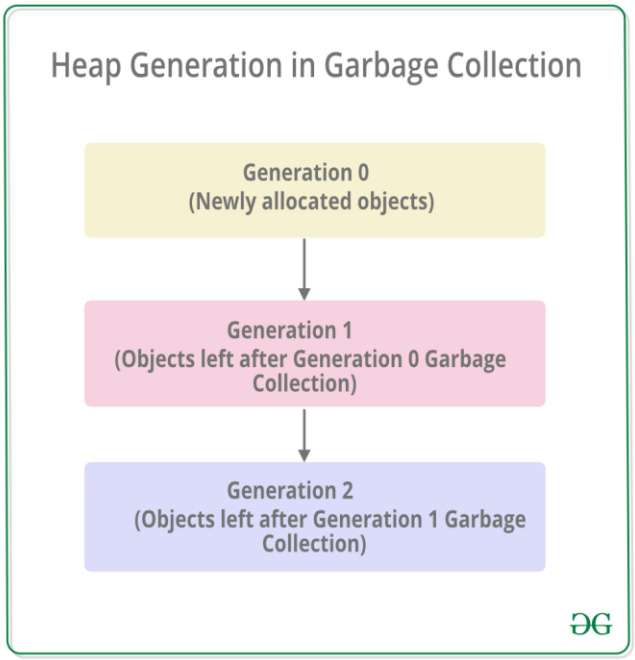
The object cleaning in the generation will not take place exactly after the life time over of the particular objects. It takes own time to implement the sweeping algorithm to free the spaces to the process.

How GC works in C#

C# garbage collection belongs to the tracing variety. It’s often called a generational approach since it employs the concept of generations to figure out which objects are eligible for collection.

Memory is divided into spaces called generations. The collector starts claiming objects in the youngest generation. Then it promotes the survivors to the next generation. The C# garbage collection uses three generations in total:

- Generation 0—This generation holds short-lived objects. Here’s where the collection process happens most often. When you instantiate a new object, it goes in this generation by default. The exceptions are objects whose sizes are equal to or greater than 85,000 bytes. Those objects are large, so they go straight into generation 2.
- Generation 1—This is an intermediate space between the short-lived and long-lived layers.
- Generation 2—Finally, this is the generation that holds objects that live the longest in the application—sometimes as long as the whole duration of the app. GC takes place here less frequently.



Methods in GC Class

The GC class controls the garbage collector of the system. Some of the methods in the GC class are given as follows:

GC.GetGeneration() Method : This method returns the generation number of the target object. It requires a single parameter i.e. the target object for which the generation number is required.

A program that demonstrates the *GC.GetGeneration()* method is given as follows:

```
using System;
public class Demo {

   public static void Main(string[] args)
   {
      Demo obj = new Demo();
      Console.WriteLine("The generation number of object obj is: "
                                 + GC.GetGeneration(obj));
   }
}
```

Output:

The generation number of object obj is: 0

GC.GetTotalMemory() Method : This method returns the number of bytes that are allocated in the system. It requires a single boolean parameter where true means that the method waits for the occurrence of garbage collection before returning and false means the opposite.

```
using System;
public class Demo {

   public static void Main(string[] args)
   {
      Console.WriteLine("Total Memory:" + GC.GetTotalMemory(false));

      Demo obj = new Demo();
```

```
        Console.WriteLine("The generation number of object obj is: "
                                + GC.GetGeneration(obj));

        Console.WriteLine("Total Memory:" + GC.GetTotalMemory(false));
    }
}
```

Output:

Total Memory:4197120

The generation number of object obj is: 0

Total Memory:4204024

Note: The output may vary as it depends on the system.

GC.Collect() Method : Garbage collection can be forced in the system using the *GC.Collect() method*. This method requires a single parameter i.e. number of the oldest generation for which garbage collection occurs.

A program that demonstrates the *GC.Collect() Method* is given as follows:

```
using System;
public class Demo {

    public static void Main(string[] args)
    {
        GC.Collect(0);
        Console.WriteLine("Garbage Collection in Generation 0 is: "
                                + GC.CollectionCount(0));
    }
}
```

Output:

Garbage Collection in Generation 0 is: 1

Benefits of Garbage Collection

- Garbage Collection succeeds in allocating objects efficiently on the heap memory using the generations of garbage collection.
- Manual freeing of memory is not needed as garbage collection automatically releases the memory space after it is no longer required.
- Garbage collection handles memory allocation safely so that no objects use the contents of another object mistakenly.
- The constructors of newly created objects do not have to initialize all the data fields as garbage collection clears the memory of objects that were previously released.

.NET Framework Class Library (FCL)

.NET Framework Class Library is the collection of classes, namespaces, interfaces and value types that are used for .NET applications.

It contains thousands of classes that supports the following functions.

- Base and user-defined data types
- Support for exceptions handling
- input/output and stream operations
- Communications with the underlying system
- Access to data
- Ability to create Windows-based GUI applications
- Ability to create web-client and server applications
- Support for creating web services

.NET Framework Class Library Namespaces

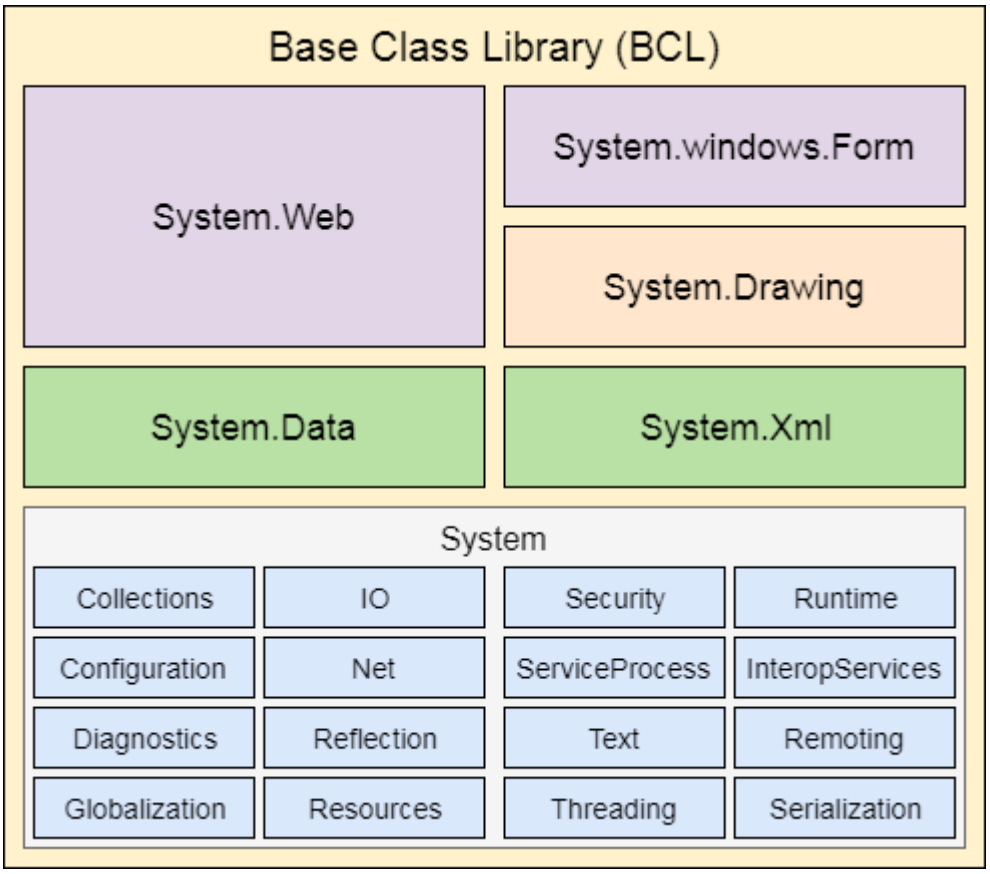
Following are the commonly used namespaces that contains useful classes and interfaces and defined in Framework Class Library.

Namespaces	Description
System	It includes all common datatypes, string values, arrays and methods for data conversion.
System.Data, System.Data.Common, System.Data.OleDb, System.Data.SqlClient, System.Data.SqlTypes	These are used to access a database, perform commands on a database and retrieve database.
System.IO, System.DirectoryServices, System.IO.IsolatedStorage	These are used to access, read and write files.
System.Diagnostics	It is used to debug and trace the execution of an application.
System.Net, System.Net.Sockets	These are used to communicate over the Internet when creating peer-to-peer applications.
System.Windows.Forms, System.Windows.Forms.Design	These namespaces are used to create Windows-based applications using Windows user interface components.

System.Web, System.WebCaching, System.Web.UI, System.Web.UI.Design, System.Web.UI.WebControls, System.Web.UI.HtmlControls, System.Web.Configuration, System.Web.Hosting, System.Web.Mail, System.Web.SessionState	These are used to create ASP. NET Web applications that run over the web.
System.Web.Services, System.Web.Services.Description, System.Web.Services.Configuration, System.Web.Services.Discovery, System.Web.Services.Protocols	These are used to create XML Web services and components that can be published over the web.
System.Security, System.Security.Permissions, System.Security.Policy, System.WebSecurity, System.Security.Cryptography	These are used for authentication, authorization, and encryption purpose.
System.Xml, System.Xml.Schema, System.Xml.Serialization, System.Xml.XPath, System.Xml.Xsl	These namespaces are used to create and access XML files.

.NET Framework Base Class Library

.NET Base Class Library is the sub part of the Framework that provides library support to Common Language Runtime to work properly. It includes the System namespace and core types of the .NET framework.



C# Visual Studio IDE

Visual Studio is the Integrated Development Environment in which developers work when creating programs in one of many languages, including C#, for the .NET Framework . It is used to create console and graphical user interface (GUI) applications along with Windows Forms or WPF (Windows Presentation Foundation) applications, web applications, and web services in both native code together with managed code for all platforms supported by Microsoft Windows, Windows Mobile, Windows CE, .NET Framework, .NET Compact Framework and Microsoft Silverlight.

It offers a set of tools that help you to write and modify the code for your programs, and also detect and correct errors in your programs. Visual Studio supports various programming languages by means of language services, which allow the code editor and debugger to support nearly any programming language, provided a language-specific service exists. Like any other IDE, it includes a code editor that supports syntax highlighting and code completion using IntelliSense for not only variables, functions and methods but also language constructs like loops and queries.

Microsoft Visual Studio is a powerful IDE that ensures quality code throughout the entire application lifecycle, from design to deployment. Some windows are used for writing code, some for designing interfaces, and others for getting a general overview of files or classes in your application. Microsoft Visual Studio includes a host of visual designers to aid in the development of various types of applications. These tools include such as Windows Forms Designer, WPF (Windows Presentation Foundation) Designer, Web development, Class designer, Data designer and Mapping designer.

Microsoft Visual Studio is available in the following editions

Visual Studio Express - Visual Studio Express is a free edition and with your express edition you can build the next great app for Windows 8, Windows Phone, and the web. The languages available as part of the Express editions are:

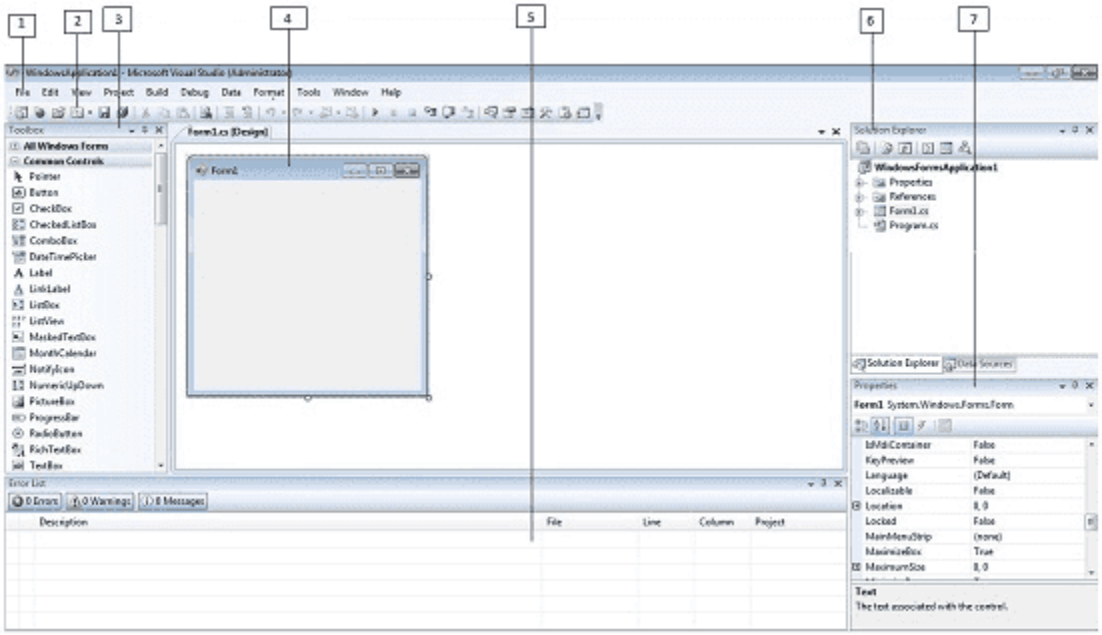
Visual Basic Express
Visual C++ Express
Visual C# Express
Visual Web Developer Express
Express for Windows Phone

Other editions are Visual Studio Professional, Visual Studio Premium, Visual Studio Tools for Office, Visual Studio Ultimate, Visual Studio Team System and Test Professional

Visual Studio Version history

Product name	Codename	Internal version	Supported .NET Framework versions	Release date
Visual Studio	N/A	4.0	N/A	1995-04
Visual Studio 97	*Boston*	5.0	N/A	1997-02
Visual Studio 6.0	*Aspen*	6.0	N/A	1998-06
Visual Studio .NET (2002)	*Rainier*	7.0	1.0	2002-02-13
Visual Studio .NET 2003	*Everett*	7.1	1.1	2003-04-24
Visual Studio 2005	*Whidbey*	8.0	2.0, 3.0	2005-11-07
Visual Studio 2008	*Orcas*	9.0	2.0, 3.0, 3.5	2007-11-19
Visual Studio 2010	*Dev10/Rosario*	10.0	2.0, 3.0, 3.5, 4.0	2010-04-12
Visual Studio 2012	*Dev11*	11.0	2.0, 3.0, 3.5, 4.0, 4.5	2012-09-12
Visual Studio 2013	*Dev12*	12.0	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1	Upcoming

C# is designed for building a variety of applications that run on the .NET Framework. Before you start learning more about C# programming, it is important to understand the development environment and identify some of the frequently using programming tools in the Visual Studio IDE.



- 1. Menu Bar

- 2. Standard Toolbar
- 3. ToolBox
- 4. Forms Designer
- 5. Output Window
- 6. Solution Explorer
- 7. Properties Window

Combined with the .NET Framework, C# enables the creation of Windows applications, Web services, database tools, components, controls, and more. Visual Studio organizes your work in projects and solutions. A solution can contain more than one project, such as a DLL and an executable that references that DLL. From the following C# chapters you will learn how to use these Visual Studio features for your C# programming needs.

1. **Code Editor:** Where the user will write code.
2. **Output Window:** Here the Visual Studio shows the outputs, compiler warnings, error messages and debugging information.
3. **Solution Explorer:** It shows the files on which the user is currently working.
4. **Properties:** It will give additional information and context about the selected parts of the current project.

What is C#

C# is pronounced as "C-Sharp". It is an object-oriented programming language provided by Microsoft that runs on .Net Framework.

By the help of C# programming language, we can develop different types of secured and robust applications:

- Window applications
- Web applications
- Distributed applications
- Web service applications
- Database applications etc.

C# is approved as a standard by ECMA and ISO. C# is designed for CLI (Common Language Infrastructure). CLI is a specification that describes executable code and runtime environment.

C# programming language is influenced by C++, Java, Eiffel, Modula-3, Pascal etc. languages.

Java vs C#

There are many differences and similarities between Java and C#. A list of top differences between Java and C# are given below:

No.	Java	C#
1)	Java is a **high level, robust, secured and object-oriented programming** language developed by Oracle.	C# is an **object-oriented programming** language developed by Microsoft that runs on .Net Framework.

2)	Java programming language is designed to be run on a Java platform, by the help of **Java Runtime Environment (JRE).**	C# programming language is designed to be run on the **Common Language Runtime (CLR).**
3)	Java type safety is safe.	C# type safety is unsafe.
4)	In java, built-in data types that are passed by value are called **primitive types.**	In C#, built-in data types that are passed by value are called **simple types.**
5)	Arrays in Java are direct specialization of **Object.**	Arrays in C# are specialization of **System.**
6)	Java does not support **conditional compilation.**	C# supports conditional compilation using preprocessor directives.
7)	Java doesn't support goto statement.	C# supports goto statement.
8)	Java doesn't support **structures and unions.**	C# supports structures and unions.
9)	Java supports checked exception and unchecked exception.	C# supports unchecked exception.

C# Features

C# is object oriented programming language. It provides a lot of **features** that are given below.

1) Simple

C# is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2) Modern Programming Language

C# programming is based upon the current trend and it is very powerful and simple for building scalable, interoperable and robust applications.

3) Object Oriented

C# is object oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grow.

4) Type Safe

C# type safe code can only access the memory location that it has permission to execute. Therefore it improves a security of the program.

5) Interoperability

Interoperability process enables the C# programs to do almost anything that a native C++ application can do.

6) Scalable and Updateable

C# is automatic scalable and updateable programming language. For updating our application we delete the old files and update them with new ones.

7) Component Oriented

C# is component oriented programming language. It is the predominant software development methodology used to develop more robust and highly scalable applications.

8) Structured Programming Language

C# is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

9) Rich Library

C# provides a lot of inbuilt functions that makes the development fast.

10) Fast Speed

The compilation and execution time of C# language is fast.

C# - Basic Syntax

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

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

Let us look at implementation of a Rectangle class and discuss C# basic syntax −

```
using System;
namespace RectangleApplication {
  class Rectangle {

    // member variables
    double length;
    double width;

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

```
    }
  }
}
```

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

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

The *using* Keyword

The first statement in any C# program is

```
using System;
```

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

The *class* Keyword

The **class** keyword is used for declaring a class.

Comments in C#

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

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

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

```
}//end class Rectangle
```

Member Variables

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

Member Functions

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

Instantiating a Class

In the preceding program, the class *ExecuteRectangle* contains the *Main()* method and instantiates the *Rectangle* class.

Identifiers

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

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

C# Keywords

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

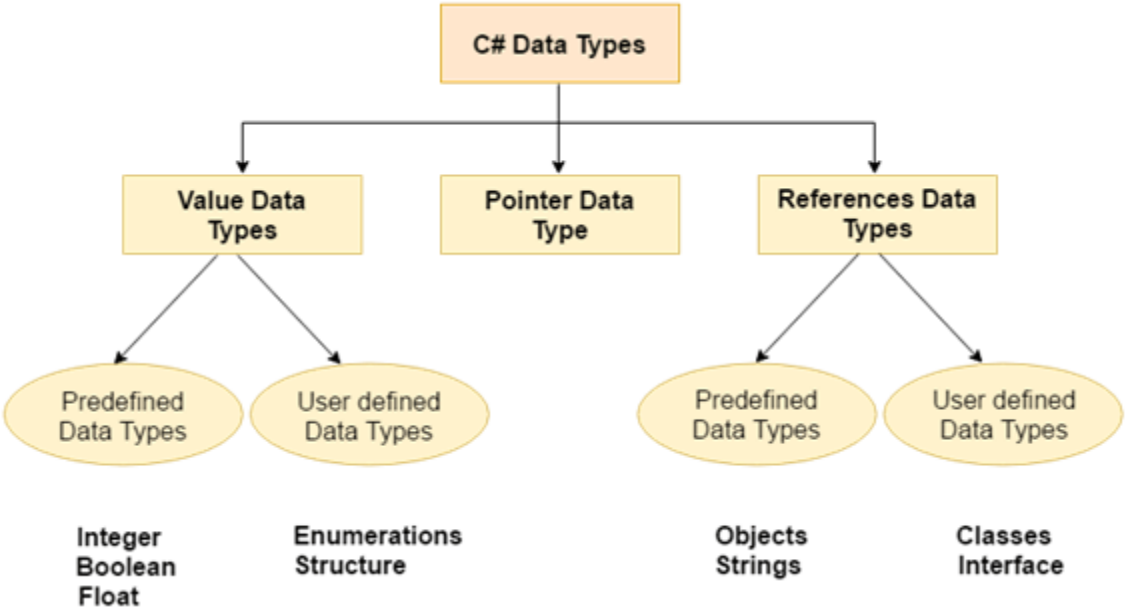
In C#, some identifiers have special meaning in context of code, such as get and set are called contextual keywords.

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

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

C# Data Types

A data type specifies the type of data that a variable can store such as integer, floating, character etc.



The variables in C#, are categorized into the following types −

- Value types
- Reference types
- Pointer types

Value Type

Value type variables can be assigned a value directly. They are derived from the class **System.ValueType**.

The value types directly contain data. Some examples are **int, char, and float**, which stores numbers, alphabets, and floating point numbers, respectively. When you declare an **int** type, the system allocates memory to store the value.

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

Type	Represents	Range	Default Value
bool	Boolean value	True or False	False
byte	8-bit unsigned integer	0 to 255	0
char	16-bit Unicode character	U +0000 to U +ffff	'\0'
decimal	128-bit precise decimal values with 28-29 significant digits	$(-7.9 \times 10^{28}$ to $7.9 \times 10^{28}) / 10^{0}$ to 28	0.0M

double	64-bit double-precision floating point type	(+/-)5.0 x 10^{-324} to (+/-)1.7 x 10^{308}	0.0D
float	32-bit single-precision floating point type	-3.4 x 10^{38} to + 3.4 x 10^{38}	0.0F
int	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0
long	64-bit signed integer type	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
sbyte	8-bit signed integer type	-128 to 127	0
short	16-bit signed integer type	-32,768 to 32,767	0
uint	32-bit unsigned integer type	0 to 4,294,967,295	0
ulong	64-bit unsigned integer type	0 to 18,446,744,073,709,551,615	0
ushort	16-bit unsigned integer type	0 to 65,535	0

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

```
using System;

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

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

```
Size of int: 4
```

Reference Type

The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables.

In other words, they refer to a memory location. Using multiple variables, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value. Example of **built-in** reference types are: **object**, **dynamic,** and **string**.

Object Type

The **Object Type** is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for System.Object class. The object types can be assigned values of any other types, value types,

reference types, predefined or user-defined types. However, before assigning values, it needs type conversion.

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

object obj;

obj = 100; // this is boxing

Dynamic Type

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

Syntax for declaring a dynamic type is −

dynamic <variable_name> = value;

For example,

dynamic d = 20;

Dynamic types are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run time.

String Type

The **String Type** allows you to assign any string values to a variable. The string type is an alias for the System.String class. It is derived from object type. The value for a string type can be assigned using string literals in two forms: quoted and @quoted.

For example,

String str = "Tutorials Point";

A @quoted string literal looks as follows −

@"Tutorials Point";

Pointer Type

Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as the pointers in C or C++.

Syntax for declaring a pointer type is −

type* identifier;

For example,

char* cptr;

int* iptr;

C# - Type Conversion

Type conversion is converting one type of data to another type. It is also known as Type Casting. In C#, type casting has two forms −

- **Implicit type conversion** − These conversions are performed by C# in a type-safe manner. For example, are conversions from smaller to larger integral types and conversions from derived classes to base classes.

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

The following example shows an explicit type conversion −

```
using System;
namespace TypeConversionApplication {
  class ExplicitConversion {
    static void Main(string[] args) {
      double d = 5673.74;
      int i;

      // cast double to int.
      i = (int)d;
      Console.WriteLine(i);
      Console.ReadKey();
    }
  }
}
```

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

```
5673
```

C# Type Conversion Methods

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

Sr.No.	Methods & Description
1	**ToBoolean** Converts a type to a Boolean value, where possible.
2	**ToByte** Converts a type to a byte.
3	**ToChar** Converts a type to a single Unicode character, where possible.
4	**ToDateTime** Converts a type (integer or string type) to date-time structures.
5	**ToDecimal** Converts a floating point or integer type to a decimal type.
6	**ToDouble** Converts a type to a double type.
7	**ToInt16**

	Converts a type to a 16-bit integer.
8	**ToInt32** Converts a type to a 32-bit integer.
9	**ToInt64** Converts a type to a 64-bit integer.
10	**ToSbyte** Converts a type to a signed byte type.
11	**ToSingle** Converts a type to a small floating point number.
12	**ToString** Converts a type to a string.
13	**ToType** Converts a type to a specified type.
14	**ToUInt16** Converts a type to an unsigned int type.
15	**ToUInt32** Converts a type to an unsigned long type.
16	**ToUInt64** Converts a type to an unsigned big integer.

The following example converts various value types to string type −

```
using System;

namespace TypeConversionApplication {
   class StringConversion {
      static void Main(string[] args) {
         int i = 75;
         float f = 53.005f;
         double d = 2345.7652;
         bool b = true;

         Console.WriteLine(i.ToString());
         Console.WriteLine(f.ToString());
         Console.WriteLine(d.ToString());
```

```
        Console.WriteLine(b.ToString());
        Console.ReadKey();

    }
  }
}
```

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

75

53.005

2345.7652

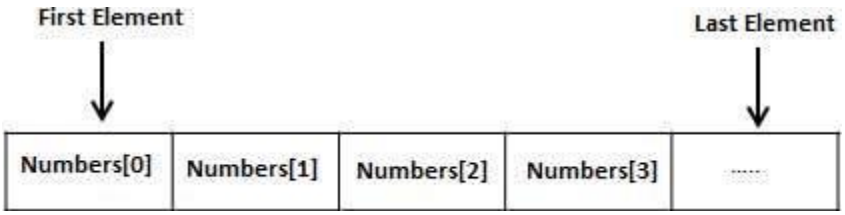
True

C# - Arrays

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

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

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



Declaring Arrays

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

datatype[] arrayName;

where,

- *datatype* is used to specify the type of elements in the array.
- *[ ]* specifies the rank of the array. The rank specifies the size of the array.
- *arrayName* specifies the name of the array.

For example,

double[] balance;

Initializing an Array

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

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

double[] balance = new double[10];

Assigning Values to an Array

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

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

balance[0] = 4500.0;
```

You can assign values to the array at the time of declaration, as shown −

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

You can also create and initialize an array, as shown −

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

You may also omit the size of the array, as shown −

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

You can copy an array variable into another target array variable. In such case, both the target and source point to the same memory location −

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

int[] score = marks;
```

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

Accessing Array Elements

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

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

The following example, demonstrates the above-mentioned concepts declaration, assignment, and accessing arrays −

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

      /* initialize elements of array n */
      for ( i = 0; i < 10; i++ ) {
        n[ i ] = i + 100;
      }

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

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

```
Element[0] = 100
Element[1] = 101
```

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

Using the *foreach* Loop

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

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

      /* initialize elements of array n */
      for ( int i = 0; i < 10; i++ ) {
        n[i] = i + 100;
      }

      /* output each array element's value */
      foreach (int j in n ) {
        int i = j-100;
        Console.WriteLine("Element[{0}] = {1}", i, j);
      }
      Console.ReadKey();
    }
  }
}
```

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

```
Element[0] = 100

Element[1] = 101

Element[2] = 102

Element[3] = 103

Element[4] = 104

Element[5] = 105

Element[6] = 106

Element[7] = 107

Element[8] = 108

Element[9] = 109
```

C# Arrays

There are following few important concepts related to array which should be clear to a C# programmer −

Sr.No.	Concept & Description
1	Multi-dimensional arrays C# supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
2	Jagged arrays C# supports multidimensional arrays, which are arrays of arrays.
3	Passing arrays to functions You can pass to the function a pointer to an array by specifying the array's name without an index.
4	Param arrays This is used for passing unknown number of parameters to a function.
5	The Array Class Defined in System namespace, it is the base class to all arrays, and provides various properties and methods for working with arrays.

C# - Multidimensional Arrays

C# allows multidimensional arrays. Multi-dimensional arrays are also called rectangular array. You can declare a 2-dimensional array of strings as −

string [,] names;

or, a 3-dimensional array of int variables as −

int [, ,] m;

Two-Dimensional Arrays

The simplest form of the multidimensional array is the 2-dimensional array. A 2-dimensional array is a list of one-dimensional arrays.

A 2-dimensional array can be thought of as a table, which has x number of rows and y number of columns. Following is a 2-dimensional array, which contains 3 rows and 4 columns −

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

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

Initializing Two-Dimensional Arrays

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

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

Accessing Two-Dimensional Array Elements

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

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

The above statement takes 4th element from the 3rd row of the array. You can verify it in the above diagram. Let us check the program to handle a two dimensional array −

```
using System;

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

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

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

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

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

C# - Jagged Arrays

A Jagged array is an array of arrays. You can declare a jagged array named *scores* of type **int** as −

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

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

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

for (int i = 0; i < scores.Length; i++) {

   scores[i] = new int[4];

}
```

You can initialize a jagged array as −

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

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

Example

The following example illustrates using a jagged array −

```
using System;

namespace ArrayApplication {
   class MyArray {
      static void Main(string[] args) {

         /* a jagged array of 5 array of integers*/
         int[][] a = new int[][]{new int[]{0,0},new int[]{1,2},
            new int[]{2,4},new int[]{ 3, 6 }, new int[]{ 4, 8 } };
         int i, j;

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

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

```
a[0][0]: 0

a[0][1]: 0

a[1][0]: 1

a[1][1]: 2

a[2][0]: 2

a[2][1]: 4

a[3][0]: 3

a[3][1]: 6
```

a[4][0]: 4

a[4][1]: 8

C# - Classes

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

Defining a Class

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

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

Note −

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

The following example illustrates the concepts discussed so far −

```
using System;
namespace BoxApplication {
   class Box {
      public double length;   // Length of a box
      public double breadth;  // Breadth of a box
      public double height;   // Height of a box
   }
   class Boxtester {
      static void Main(string[] args) {
         Box Box1 = new Box();   // Declare Box1 of type Box
```

```
      Box Box2 = new Box();   // Declare Box2 of type Box
      double volume = 0.0;    // Store the volume of a box here

      // box 1 specification
      Box1.height = 5.0;
      Box1.length = 6.0;
      Box1.breadth = 7.0;

      // box 2 specification
      Box2.height = 10.0;
      Box2.length = 12.0;
      Box2.breadth = 13.0;

      // volume of box 1
      volume = Box1.height * Box1.length * Box1.breadth;
      Console.WriteLine("Volume of Box1 : {0}",  volume);

      // volume of box 2
      volume = Box2.height * Box2.length * Box2.breadth;
      Console.WriteLine("Volume of Box2 : {0}", volume);
      Console.ReadKey();
    }
  }
}
```

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

Volume of Box1 : 210

Volume of Box2 : 1560

Member Functions and Encapsulation

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

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

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

```
using System;

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

    public void setLength( double len ) {
      length = len;
    }
    public void setBreadth( double bre ) {
      breadth = bre;
    }
    public void setHeight( double hei ) {
      height = hei;
    }
    public double getVolume() {
```

```
      return length * breadth * height;
    }
  }
  class Boxtester {
    static void Main(string[] args) {
      Box Box1 = new Box();   // Declare Box1 of type Box
      Box Box2 = new Box();
      double volume;

      // Declare Box2 of type Box
      // box 1 specification
      Box1.setLength(6.0);
      Box1.setBreadth(7.0);
      Box1.setHeight(5.0);

      // box 2 specification
      Box2.setLength(12.0);
      Box2.setBreadth(13.0);
      Box2.setHeight(10.0);

      // volume of box 1
      volume = Box1.getVolume();
      Console.WriteLine("Volume of Box1 : {0}" ,volume);

      // volume of box 2
      volume = Box2.getVolume();
      Console.WriteLine("Volume of Box2 : {0}", volume);

      Console.ReadKey();
    }
  }
}
```

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

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

C# Constructors

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

A constructor has exactly the same name as that of class and it does not have any return type. Following example explains the concept of constructor −

```
using System;

namespace LineApplication {
  class Line {
    private double length;   // Length of a line

    public Line() {
      Console.WriteLine("Object is being created");
    }
    public void setLength( double len ) {
      length = len;
    }
```

```
    public double getLength() {
      return length;
    }

    static void Main(string[] args) {
      Line line = new Line();

      // set line length
      line.setLength(6.0);
      Console.WriteLine("Length of line : {0}", line.getLength());
      Console.ReadKey();
    }
  }
}
```

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

```
Object is being created

Length of line : 6
```

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

```
using System;

namespace LineApplication {
  class Line {
    private double length;   // Length of a line

    public Line(double len) {  //Parameterized constructor
      Console.WriteLine("Object is being created, length = {0}", len);
      length = len;
    }
    public void setLength( double len ) {
      length = len;
    }
    public double getLength() {
      return length;
    }
    static void Main(string[] args) {
      Line line = new Line(10.0);
      Console.WriteLine("Length of line : {0}", line.getLength());

      // set line length
      line.setLength(6.0);
      Console.WriteLine("Length of line : {0}", line.getLength());
      Console.ReadKey();
    }
  }
}
```

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

```
Object is being created, length = 10

Length of line : 10

Length of line : 6
```

C# Destructors

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

Destructor can be very useful for releasing memory resources before exiting the program. Destructors cannot be inherited or overloaded.

Following example explains the concept of destructor −

```
using System;

namespace LineApplication {
  class Line {
    private double length;   // Length of a line

    public Line() {   // constructor
      Console.WriteLine("Object is being created");
    }
    ~Line() {   //destructor
      Console.WriteLine("Object is being deleted");
    }
    public void setLength( double len ) {
      length = len;
    }
    public double getLength() {
      return length;
    }
    static void Main(string[] args) {
      Line line = new Line();

      // set line length
      line.setLength(6.0);
      Console.WriteLine("Length of line : {0}", line.getLength());
    }
  }
}
```

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

```
Object is being created

Length of line : 6

Object is being deleted
```

Static Members of a C# Class

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

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

The following example demonstrates the use of **static variables** −

```
using System;
```

```
namespace StaticVarApplication {
   class StaticVar {
      public static int num;

      public void count() {
         num++;
      }
      public int getNum() {
         return num;
      }
   }
   class StaticTester {
      static void Main(string[] args) {
         StaticVar s1 = new StaticVar();
         StaticVar s2 = new StaticVar();

         s1.count();
         s1.count();
         s1.count();

         s2.count();
         s2.count();
         s2.count();

         Console.WriteLine("Variable num for s1: {0}", s1.getNum());
         Console.WriteLine("Variable num for s2: {0}", s2.getNum());
         Console.ReadKey();
      }
   }
}
```

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

```
Variable num for s1: 6

Variable num for s2: 6
```

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

```
using System;

namespace StaticVarApplication {
   class StaticVar {
      public static int num;

      public void count() {
         num++;
      }
      public static int getNum() {
         return num;
      }
   }
   class StaticTester {
      static void Main(string[] args) {
         StaticVar s = new StaticVar();

         s.count();
```

```
        s.count();
        s.count();

        Console.WriteLine("Variable num: {0}", StaticVar.getNum());
        Console.ReadKey();
      }
   }
}
```

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

```
Variable num: 3
```

C# - Inheritance

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

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

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

Base and Derived Classes

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

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

```
<acess-specifier> class <base_class> {

   ...

}


class <derived_class> : <base_class> {

   ...

}
```

Consider a base class Shape and its derived class Rectangle −

```
using System;

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

```
   // Derived class
   class Rectangle: Shape {
      public int getArea() {
         return (width * height);
      }
   }
   class RectangleTester {
      static void Main(string[] args) {
         Rectangle Rect = new Rectangle();

         Rect.setWidth(5);
         Rect.setHeight(7);

         // Print the area of the object.
         Console.WriteLine("Total area: {0}",  Rect.getArea());
         Console.ReadKey();
      }
   }
}
```

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

```
Total area: 35
```

Initializing Base Class

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

The following program demonstrates this −

```
using System;

namespace RectangleApplication {
   class Rectangle {

      //member variables
      protected double length;
      protected double width;

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

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

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

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

Multiple Inheritance in C#

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

```
using System;

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

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

   // Derived class
   class Rectangle : Shape, PaintCost {
      public int getArea() {
         return (width * height);
      }
      public int getCost(int area) {
         return area * 70;
      }
```

```
  }
  class RectangleTester {
    static void Main(string[] args) {
      Rectangle Rect = new Rectangle();
      int area;

      Rect.setWidth(5);
      Rect.setHeight(7);
      area = Rect.getArea();

      // Print the area of the object.
      Console.WriteLine("Total area: {0}",  Rect.getArea());
      Console.WriteLine("Total paint cost: ${0}" , Rect.getCost(area));
      Console.ReadKey();
    }
  }
}
```

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

Total area: 35

Total paint cost: $2450

C# - Polymorphism

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

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

Static Polymorphism

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

- Function overloading
- Operator overloading**Function Overloading**

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

The following example shows using function **print()** to print different data types −

```
using System;

namespace PolymorphismApplication {
  class Printdata {
    void print(int i) {
      Console.WriteLine("Printing int: {0}", i );
    }
    void print(double f) {
      Console.WriteLine("Printing float: {0}" , f);
    }
    void print(string s) {
      Console.WriteLine("Printing string: {0}", s);
    }
```

```
    static void Main(string[] args) {
      Printdata p = new Printdata();

      // Call print to print integer
      p.print(5);

      // Call print to print float
      p.print(500.263);

      // Call print to print string
      p.print("Hello C++");
      Console.ReadKey();
    }
  }
}
```

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

```
Printing int: 5

Printing float: 500.263

Printing string: Hello C++
```

Dynamic Polymorphism

C# allows you to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it. **Abstract** classes contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.

Here are the rules about abstract classes −

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

The following program demonstrates an abstract class −

```
using System;

namespace PolymorphismApplication {
  abstract class Shape {
    public abstract int area();
  }

  class Rectangle:  Shape {
    private int length;
    private int width;

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

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

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

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

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

Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.

The following program demonstrates this −

```
using System;

namespace PolymorphismApplication {
  class Shape {
    protected int width, height;

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

    }
    public override int area () {
      Console.WriteLine("Rectangle class area :");
      return (width * height);
    }
  }
  class Triangle: Shape {
    public Triangle(int a = 0, int b = 0): base(a, b) {
    }
    public override int area() {
      Console.WriteLine("Triangle class area :");
      return (width * height / 2);
    }
  }
  class Caller {
    public void CallArea(Shape sh) {
```

```
        int a;
        a = sh.area();
        Console.WriteLine("Area: {0}", a);
      }
   }
   class Tester {
      static void Main(string[] args) {
        Caller c = new Caller();
        Rectangle r = new Rectangle(10, 7);
        Triangle t = new Triangle(10, 5);

        c.CallArea(r);
        c.CallArea(t);
        Console.ReadKey();
      }
   }
}
```

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

Rectangle class area:

Area: 70

Triangle class area:

Area: 25

C# - Interfaces

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

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

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

Declaring Interfaces

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

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

Example

The following example demonstrates implementation of the above interface −

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

namespace InterfaceApplication {
```

```
   public interface ITransactions {
      // interface members
      void showTransaction();
      double getAmount();
   }
   public class Transaction : ITransactions {
      private string tCode;
      private string date;
      private double amount;

      public Transaction() {
         tCode = " ";
         date = " ";
         amount = 0.0;
      }
      public Transaction(string c, string d, double a) {
         tCode = c;
         date = d;
         amount = a;
      }
      public double getAmount() {
         return amount;
      }
      public void showTransaction() {
         Console.WriteLine("Transaction: {0}", tCode);
         Console.WriteLine("Date: {0}", date);
         Console.WriteLine("Amount: {0}", getAmount());
      }
   }
   class Tester {

      static void Main(string[] args) {
         Transaction t1 = new Transaction("001", "8/10/2012", 78900.00);
         Transaction t2 = new Transaction("002", "9/10/2012", 451900.00);

         t1.showTransaction();
         t2.showTransaction();
         Console.ReadKey();
      }
   }
}
```

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

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

C# - Delegates

C# delegates are similar to pointers to functions, in C or C++. A **delegate** is a reference type variable that holds the reference to a method. The reference can be changed at runtime.

Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the **System.Delegate** class.

Declaring Delegates

Delegate declaration determines the methods that can be referenced by the delegate. A delegate can refer to a method, which has the same signature as that of the delegate.

For example, consider a delegate −

public delegate int MyDelegate (string s);

The preceding delegate can be used to reference any method that has a single *string* parameter and returns an *int* type variable.

Syntax for delegate declaration is −

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

Instantiating Delegates

Once a delegate type is declared, a delegate object must be created with the **new** keyword and be associated with a particular method. When creating a delegate, the argument passed to the **new** expression is written similar to a method call, but without the arguments to the method. For example −

public delegate void printString(string s);

...

printString ps1 = new printString(WriteToScreen);

printString ps2 = new printString(WriteToFile);

Following example demonstrates declaration, instantiation, and use of a delegate that can be used to reference methods that take an integer parameter and returns an integer value.

```
using System;

delegate int NumberChanger(int n);
namespace DelegateAppl {

   class TestDelegate {
      static int num = 10;

      public static int AddNum(int p) {
         num += p;
         return num;
      }
      public static int MultNum(int q) {
         num *= q;
         return num;
      }
      public static int getNum() {
         return num;
      }
      static void Main(string[] args) {
         //create delegate instances
         NumberChanger nc1 = new NumberChanger(AddNum);
         NumberChanger nc2 = new NumberChanger(MultNum);

         //calling the methods using the delegate objects
         nc1(25);
         Console.WriteLine("Value of Num: {0}", getNum());
         nc2(5);
         Console.WriteLine("Value of Num: {0}", getNum());
```

```
      Console.ReadKey();
    }
  }
}
```

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

```
Value of Num: 35

Value of Num: 175
```

Multicasting of a Delegate

Delegate objects can be composed using the "+" operator. A composed delegate calls the two delegates it was composed from. Only delegates of the same type can be composed. The "-" operator can be used to remove a component delegate from a composed delegate.

Using this property of delegates you can create an invocation list of methods that will be called when a delegate is invoked. This is called **multicasting** of a delegate. The following program demonstrates multicasting of a delegate −

```
using System;

delegate int NumberChanger(int n);
namespace DelegateAppl {
  class TestDelegate {
    static int num = 10;

    public static int AddNum(int p) {
      num += p;
      return num;
    }
    public static int MultNum(int q) {
      num *= q;
      return num;
    }
    public static int getNum() {
      return num;
    }
    static void Main(string[] args) {
      //create delegate instances
      NumberChanger nc;
      NumberChanger nc1 = new NumberChanger(AddNum);
      NumberChanger nc2 = new NumberChanger(MultNum);

      nc = nc1;
      nc += nc2;

      //calling multicast
      nc(5);
      Console.WriteLine("Value of Num: {0}", getNum());
      Console.ReadKey();
    }
  }
}
```

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

```
Value of Num: 75
```

Using Delegates

The following example demonstrates the use of delegate. The delegate *printString* can be used to reference method that takes a string as input and returns nothing.

We use this delegate to call two methods, the first prints the string to the console, and the second one prints it to a file −

```
using System;
using System.IO;
namespace DelegateAppl {

   class PrintString {
      static FileStream fs;
      static StreamWriter sw;

      // delegate declaration
      public delegate void printString(string s);

      // this method prints to the console
      public static void WriteToScreen(string str) {
         Console.WriteLine("The String is: {0}", str);
      }

      //this method prints to a file
      public static void WriteToFile(string s) {
         fs = new FileStream("c:\\message.txt",
         FileMode.Append, FileAccess.Write);
         sw = new StreamWriter(fs);
         sw.WriteLine(s);
         sw.Flush();
         sw.Close();
         fs.Close();
      }

      // this method takes the delegate as parameter and uses it to
      // call the methods as required
      public static void sendString(printString ps) {
         ps("Hello World");
      }

      static void Main(string[] args) {
         printString ps1 = new printString(WriteToScreen);
         printString ps2 = new printString(WriteToFile);
         sendString(ps1);
         sendString(ps2);
         Console.ReadKey();
      }
   }
}
```

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

```
The String is: Hello World
```

C# - Events

Events are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.

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 some other class. The class containing the event is used to publish the event. This is called the **publisher** class. Some other class that accepts this event is called the **subscriber** class. Events use the **publisher-subscriber** model.

A **publisher** is an object that contains the definition of the event and the delegate. The event-delegate association is also defined in this object. A publisher class object invokes the event and it is notified to other objects.

A **subscriber** is an object that accepts the event and provides an event handler. The delegate in the publisher class invokes the method (event handler) of the subscriber class.

Declaring Events

To declare an event inside a class, first a delegate type for the event must be declared. For example,

public delegate string MyDel(string str);

Next, the event itself is declared, using the **event** keyword −

event MyDel MyEvent;

The preceding code defines a delegate named *BoilerLogHandler* and an event named *BoilerEventLog*, which invokes the delegate when it is raised.

Example

```
using System;
namespace SampleApp {
  public delegate string MyDel(string str);

  class EventProgram {
    event MyDel MyEvent;

    public EventProgram() {
      this.MyEvent += new MyDel(this.WelcomeUser);
    }
    public string WelcomeUser(string username) {
      return "Welcome " + username;
    }
    static void Main(string[] args) {
      EventProgram obj1 = new EventProgram();
      string result = obj1.MyEvent("Tutorials Point");
      Console.WriteLine(result);
    }
  }
}
```

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

Welcome Tutorials Point