SATHYABAMA UNIVERSITY

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE MATERIAL

Subject Name : C# AND .NET **UNIT I** **Subject Code : SCSX1008**

1.1 Overview of .NET
1.2 Advantages of .NET over the other languages
1.3 Overview of .NET binaries
1.4 Intermediate Language – Metadata
1.5. NET Namespaces
1.6 Common language runtime
1.7 Common type system
1.8 Common language specification
1.9 C# fundamentals
1.10C# class – object
1.11 string formatting
1.12 Types – scope – Constants
1.13 C# iteration – Control flow
1.14Operators
1.15 Array
1.16String
1.17 Enumerations
1.18 Structures
1.19Custom namespaces
1.20 Object oriented programming concepts -Class – Encapsulation
1.21 Inheritance
1.22 Polymorphic
1.23 Casting.

1.1 Overview of .NET

What is .NET?

- .NET is a platform that provides a standardized set of services.
  - It's just like Windows, except distributed over the Internet.
  - It exports a common interface so that it's programs can be run on any system that supports .NET.

.NET Framework

- A specific software framework
  - Includes a common runtime
  - Programming model for .NET
  - Platform for running .NET managed code in a virtual machine
  - Provides a very good environment to develop networked applications and Web Services
  - Provides programming API and unified language-independent development framework

The Core of .NET Framework: FCL & CLR

- Common Language Runtime
  - Garbage collection
  - Language integration
  - Multiple versioning support
  - Integrated security

- Framework Class Library
  - Provides the core functionality:
    ASP.NET, Web Services, ADO.NET, Windows Forms, IO, XML, etc.

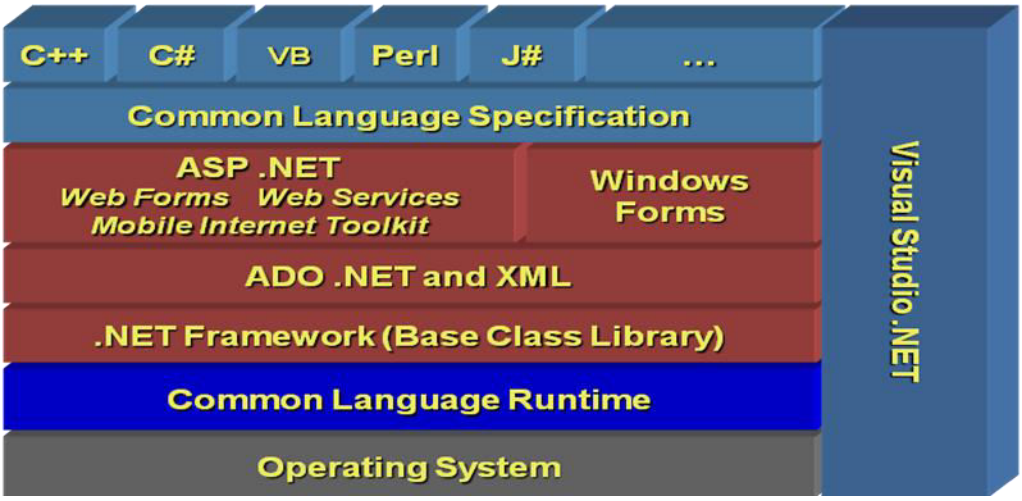
A new software platform for the desktop and the Web.

SATHYABAMA UNIVERSITY

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE MATERIAL

Subject Name : C# AND .NET **UNIT I** **Subject Code : SCSX1008**



The .NET Framework is designed to fulfill the following objectives:

- To provide a consistent object-oriented programming environment where object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
- To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.
- To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

1.2 ADVANTAGES OF .NET OVER THE OTHER LANGUAGES

➔ Language Interoperability. (means .net supports more than 40 languages writing code in one .net language can be used in other .net language)

➔ Language Independent platform.

➔ Through this technology one can develop web application as well window application (desktop application).

➔ Provides cross language inheritance.

➔ Support side by side execution.

➔ Memory Leak and crash Protection.

➔ Powerful database-driven functionality.

ADVANTAGES OVER C AND C++ :

- It is compiled to an intermediate language (CIL) independently of the language it was developed or the target architecture and operating system
- Automatic garbage collection.
- Pointers no longer needed (but optional).
- Reflection capabilities .
- Don't need to worry about header files ".h"
- Definition of classes and functions can be done in any order
- Declaration of functions and classes not needed.
- There are no global functions or variables, everything belongs to a class
- All the variables are initialized to their default values before being used (this is automatic by default but can be done manually using static constructors)
- Can't use non-Boolean variables (integers, floats...) as conditions. This is much more clean and less error prone
- Apps can be executed within a restricted sandbox

ADVANTAGES OVER C++ AND JAVA

- Formalized concept of get-set methods, so the code becomes more legible
- More clean events management (using delegates)

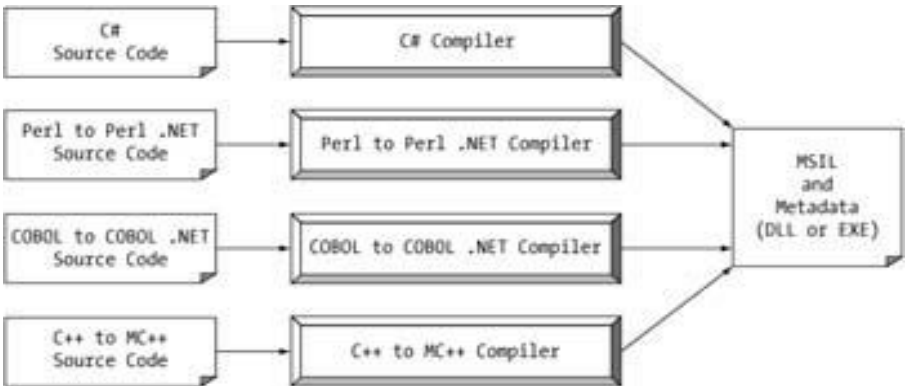
ADVANTAGES OVER JAVA

- Usually it is much more efficient than java and runs faster
- CIL (Common (.NET) Intermediate Language) is a standard language, while java byte codes aren't.
- It has more primitive types (*value types*), including unsigned numeric types.
- Indexers let to access objects as if they were arrays.
- Conditional compilation.
- Simplified multithreading.
- Operator overloading. It can make development a bit trickier but they are optional and sometimes very useful.
- (limited) use of pointers if you really need them, as when calling unmanaged (native) libraries which doesn't run on top of the virtual machine (CLR)

1.3 Overview of .NET binaries

.NET binaries take the same file extension as classic COM binaries (*.dll or *.exe), they have absolutely no internal similarities. For example, *.dll .NET binaries do not export methods to facilitate communications with the classic COM runtime (given that .NET is not COM).

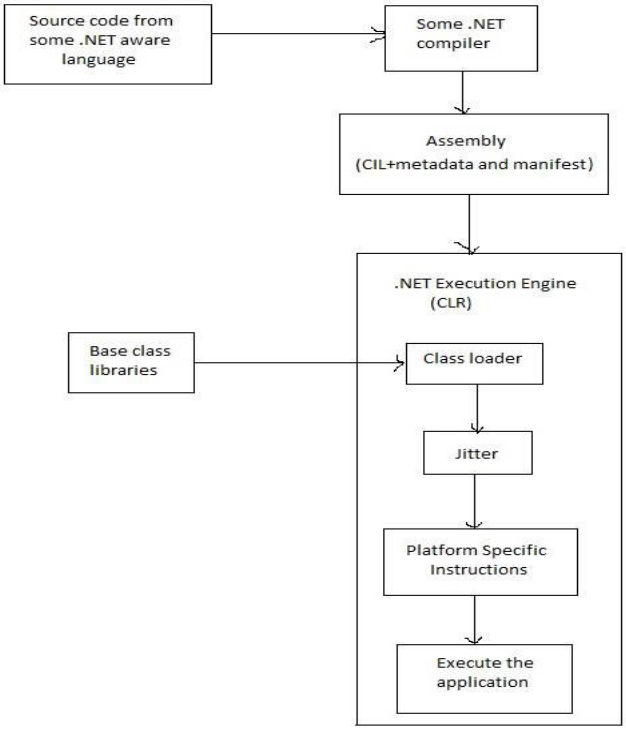
Furthermore, .NET binaries are not described using IDL and are not registered into the system registry. Perhaps most important, unlike classic COM servers, .NET binaries do not contain platform-specific instructions, but rather platform-agnostic "intermediate language" (IL).



When a *.dll or *.exe has been created using a .NET-aware compiler, the resulting module is bundled into an "assembly. As mentioned, an assembly contains CIL code, which is conceptually similar to Java byte code in that it is not compiled to platform-specific instructions until absolutely necessary. Typically "absolutely necessary" is the point at which a block of CIL instructions (such as a method implementation) are referenced for use by the .NET runtime engine.

In addition to CIL instructions, assemblies also contain metadata that describes in vivid detail the characteristics of every "type" living within the binary. For example, if you have a class named Car contained within a given assembly, the type metadata describes details such as Car's base class, which interfaces are implemented by Car (if any), as well as a full description of each member supported by the Car type.

Finally, in addition to CIL and type metadata, assemblies themselves are also described using metadata, which is officially termed a *manifest*. The manifest contains information about the current version of the assembly, culture information (used for localizing string and image resources), and a list of all externally referenced assemblies that are required for proper execution.



1.4 Intermediate Language

This is the language code generated by the C# compiler or any .NET-aware compiler.

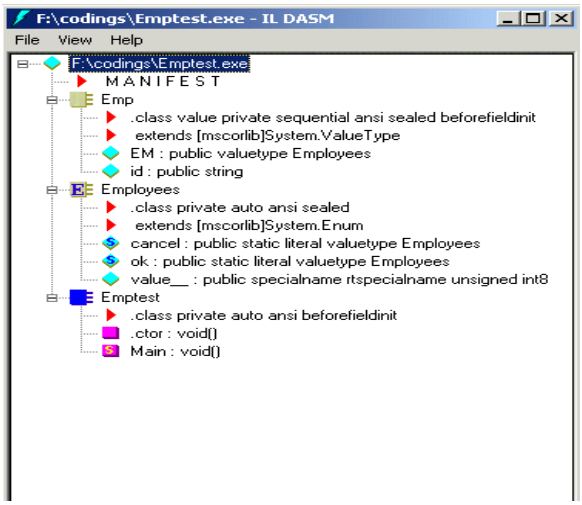
Intermediate Language: This converts native code into byte code, which is CPU –independent.
i.e. machine understandable code.

MSIL is Microsoft Intermediate Language: When we compile .Net applications, its complied to MSIL, which is not machine read language. Hence Common Language Runtime (CLR) with Just In Time Complier (JLT) converts this MSIL to native code (binary code), which is machine language.

All .NET languages generate this code. This is the code that is executed during runtime.

This MSIL code can be viewed with the help of a utility called Intermediate Language Disassembler (ILDASM). This utility displays the application's information in a tree-like fashion. Because the contents of this file are read-only, a programmer or anybody accessing these files cannot make any modifications to the output generated by the source code.

To view the MSIL Code for a sample Hello C# program, open the Hello.exe file from the ILDASM tool; it can be accessed via the Run command on the Start menu. Locate this tool manually. Normally, it will be in the Bin folder of the .NET SDK installation. Figure 1 shows an outline of this tool.



The main advantages of IL are:

1. IL is not dependent on any language and there is a possibility to create applications with modules that were written using different .NET compatible languages.
2. Platform independence - IL can be compiled to different platforms or operating systems.

1.4 Meta data

Metadata is machine-readable information about a resource, or ""data about data." In .NET, metadata includes type definitions, version information, external assembly references, and other standardized information.

Metadata describes contents in an assembly classes, interfaces, enums, structs, etc., and their containing namespaces, the name of each type, its visibility/scope, its base class, the nterfaces it implemented, its methods and their scope, and each method's parameters, type's properties, and so on.

.NET Metadata

Within the Common Language Runtime (CLR), metadata is data that describes the state of the assembly and a detailed description of each type, attribute within the assembly. Metadata includes the following information which is contained in a *manifest.* ILDASM is a .NET dissembler which allows a developer to look at the metadata (manifest) of an assembly

Assembly Metadata: Identity, types, dependent assemblies, and security permissions required for the assembly.

Type Metadata : Name, bases, interfaces, visibility and members.

Attributes : Descriptive elements that annotate assemblies, types, and members.

.NET Framework Class Library

The .NET Framework class library is a collection of reusable types that tightly integrate with the common language runtime. The class library is object oriented, providing types from which your own managed code can derive functionality. This not only makes the .NET Framework types easy to use, but also reduces the time associated with learning new features of the .NET Framework. In addition, third-party components can integrate seamlessly with classes in the .NET Framework.

For example, the .NET Framework collection classes implement a set of interfaces that you can use to develop your own collection classes. The .NET Framework types enable you to accomplish a range of common programming tasks, including tasks such as string management, data collection, database connectivity, and file access. In addition to these common tasks, the class library includes types that support a variety of specialized development scenarios. For example, you can use the .NET Framework to develop the following types of applications and services:

- Console applications.
- Windows GUI applications (Windows Forms).
- ASP.NET applications.
- XML Web services.
- Windows services.

1.5 NAMESPACES

C# does not come with a pre-defined set of languages specific classes, there is no C# class library. C# uses the existing types supplied by the .NET framework.To keep all the types within this binary well organized the .NET platform makes extensive use of namespace concept. The difference between language specific library such as MFC, is that any language targeting the .NET runtime brakes use of same namespace.

System is the root namespace for other .NET namespaces.
Namespaces are a way to group semantically related types.

A declarative region that provides a way to keep one set of names separate from another,i.e the names declared in one namespace will not conflict with with the same names in another namespace.

Within an Namespace we can declare

*classes	*structures
*delegates	*Enumerations
*Interfaces	*and another Namespaces.

Pre defined .NET NAMESPACES:-

.NET namespace	Meaning
System	Contains low level classes dialing with primitive types mathematical and manipulations
System.Collections	This namespace defines a number of stock container objects(ArrayList, Queue) as well as base types and interfaces allow you to build customized collections
System.Data	Used for database manipulations.
System.Diagnostics	Used to debug & trace your code
Sysatem.Drawing	Contains various primitives such as bitmaps, fonts, icons etc
System.IO	Includes file IO, buffering
System.NET	Contains types related to network programming
System.Security	Contains numerous types dealing with permissions, cryptography etc.
System.Threading	Deals with threading issues
System.Web	Deals with web Applications

.NET namespace	Meaning
System.Windows.Forms	Contains the types that facilitate the construction of windows, dialogboxes etc.

syntax:

```
namespace name
{
….
….
….
}
```

Example:

```
usingSystem;
class AA
{
public static void Main()
{
Console.WriteLine("we are inside namespace system");
}
}
```

output:

we are inside namespace system

1.6 COMMON LANGUAGE RUNTIME

Features of the Common Language Runtime

The common language runtime manages memory, thread execution, code execution, code safety verification, compilation, and other system services. These features are intrinsic to the managed code that runs on the common language runtime.

With regards to security, managed components are awarded varying degrees of trust, depending on a number of factors that include their origin (such as the Internet, enterprise network, or local computer). This means that a managed component might or might not be able to perform file-access operations, registry-access operations, or other sensitive functions, even if it is being used in the same active application.

The runtime enforces code access security. For example, users can trust that an executable embedded in a Web page can play an animation on screen or sing a song, but cannot access

their personal data, file system, or network. The security features of the runtime thus enable legitimate Internet-deployed software to be exceptionally feature rich.

The runtime also enforces code robustness by implementing a strict type-and-code-verification infrastructure called the common type system (CTS). The CTS ensures that all managed code is self-describing. The various Microsoft and third-party language compilers generate managed code that conforms to the CTS. This means that managed code can consume other managed types and instances, while strictly enforcing type fidelity and type safety.

In addition, the managed environment of the runtime eliminates many common software issues. For example, the runtime automatically handles object layout and manages references to objects, releasing them when they are no longer being used. This automatic memory management resolves the two most common application errors, memory leaks and invalid memory references.

The runtime also accelerates developer productivity. For example, programmers can write applications in their development language of choice, yet take full advantage of the runtime, the class library, and components written in other languages by other developers. Any compiler vendor who chooses to target the runtime can do so. Language compilers that target the .NET Framework make the features of the .NET Framework available to existing code written in that language, greatly easing the migration process for existing applications.

While the runtime is designed for the software of the future, it also supports software of today and yesterday. Interoperability between managed and unmanaged code enables developers to continue to use necessary COM components and DLLs.

The runtime is designed to enhance performance. Although the common language runtime provides many standard runtime services, managed code is never interpreted. A feature called just-in-time (JIT) compiling enables all managed code to run in the native machine language of the system on which it is executing. Meanwhile, the memory manager removes the possibilities of fragmented memory and increases memory locality-of-reference to further increase performance.

Finally, the runtime can be hosted by high-performance, server-side applications, such as Microsoft® SQL Server™ and Internet Information Services (IIS). This infrastructure enables you to use managed code to write your business logic, while still enjoying the superior performance of the industry's best enterprise servers that support runtime hosting.

Common Language Runtime

- Manages running code – like a virtual machine
  - Threading
  - Memory management
  - No interpreter: JIT-compiler produces native code – during the program installation or at run time
- Fine-grained evidence-based security
  - Code access security
    - Code can be verified to guarantee type safety
    - No unsafe casts, no un-initialized variables and no out-of-bounds array indexing
  - Role-based security

Managed Code

- Code that targets the CLR is referred to as managed code
- All managed code has the features of the CLR
  - Object-oriented
  - Type-safe
  - Cross-language integration
  - Cross language exception handling
  - Multiple version support
- Managed code is represented in special Intermediate Language (IL)

Automatic Memory Management

- The CLR manages memory for managed code
  - All allocations of objects and buffers made from a *Managed Heap*
  - Unused objects and buffers are cleaned up automatically through *Garbage Collection*
- Some of the worst bugs in software development are not possible with managed code
  - Leaked memory or objects
  - References to freed or non-existent objects
  - Reading of uninitialised variables
- Pointerless environment

Multiple Language Support

- IL (MSIL or CIL) – Intermediate Language
  - It is low-level (machine) language, like Assembler, but is Object-oriented
- CTS is a rich type system built into the CLR
  - Implements various types (int, float, string, …)
  - And operations on those types
- CLS is a set of specifications that all languages and libraries need to follow
  - This will ensure interoperability between languages

Intermediate Language

- .NET languages are compiled to an Intermediate Language (IL)
- IL is also known as MSIL or CIL
- CLR compiles IL in just-in-time (JIT) manner – each function is compiled just before execution
- The JIT code stays in memory for subsequent calls
- Recompilations of assemblies are also possible

1.7 COMMON TYPE SYSTEM (CTS)

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

The common type system performs the following functions:

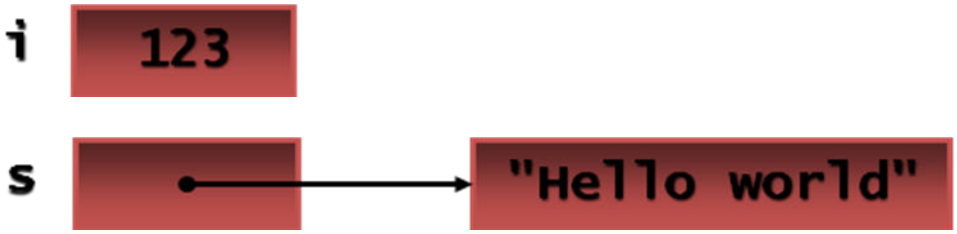
- Establishes a framework that helps enable cross-language integration, type safety, and high performance code execution.
- Provides an object-oriented model that supports the complete implementation of many programming languages.
- Defines rules that languages must follow, which helps ensure that objects written in different languages can interact with each other.
- Encapsulates data structures.

- All .NET languages have the same primitive data types. An *int* in C# is the same as an *int* in VB.NET
- When communicating between modules written in any .NET language, the types are guaranteed to be compatible on the binary level
- Types can be:
  - Value types – passed by value, stored in the stack
  - Reference types – passed by reference, stored in the heap
- Strings are a primitive data type now.

There are two general types of categories in .Net Framework that Common Type System support. They are value types and reference types. Value types contain data and are user-defined or built-in. they are placed in a stack or in order in a structure. Reference types store a reference of the value's memory address. They are allocated in a heap structure. There are many other types that can be defined under Value types and Reference types.

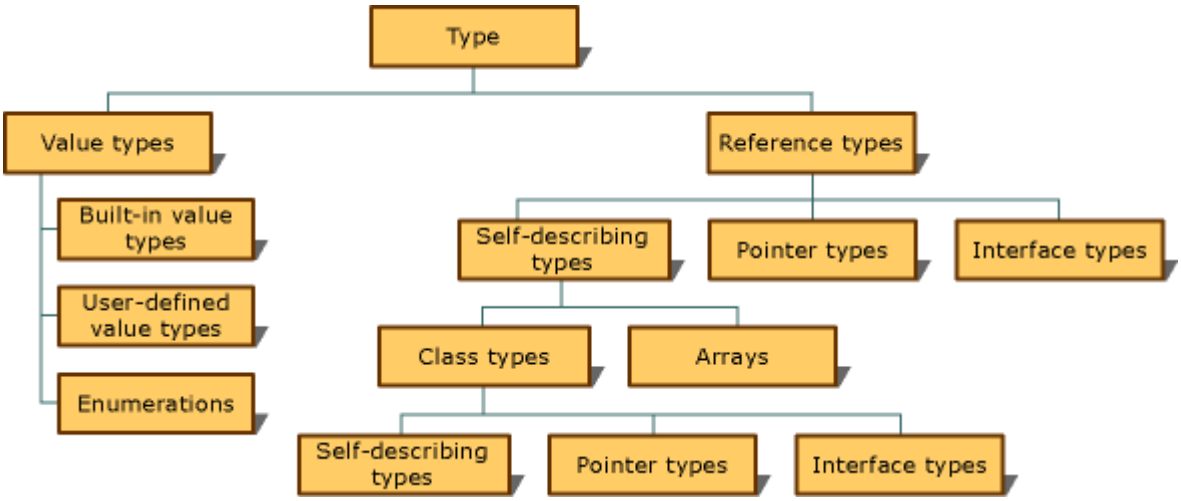
Type System

- Value types
  - Directly contain data
  - Cannot be null
- Reference types
  - Contain references to objects
  - May be null
- int i = 123;
- string s = "Hello world";



- Value types
  - Primitives int i;
  - Enums enum State { Off, On }
  - Structs struct Point { int x, y; }
- Reference types
  - Classes class Foo: Bar, IFoo {...}
  - Interfaces interface IFoo: IBar {...}
  - Arrays string[] a = new string[10];
  - Delegates delegate void Empty();

Type classification



CTS Structure Types

The concept of a structure is also formalized under the CTS.In general, a structure can be thought of as a lightweight class type having value semantics. For example, CTS structures may define any number of parameterized constructors (the no-argument constructor is reserved). In this way, you are able to establish the value of each field during the time of construction.

While structures are best suited for modeling geometric and mathematical types, the following type offers a bit more pizzazz.

```
// Create a C# structure.
struct Baby
{
        // Structures can contain fields.
        public string name;

        // Structures can contain constructors (with arguments).
        public Baby(string name)
                { this.name = name; }

// Structures may take methods.
        public void Cry()
                {     Console.WriteLine("Waaaaaaaaaaah!!!"); }
    public bool IsSleeping() { return false; }
```

```
        public bool IsChanged() { return false; }

    }
```

Here is our structure in action (assume this logic is contained in some Main() method):

```
    // Welcome to the world Max Barnaby!!
        Baby barnaBaby = new Baby("Max");
        Console.WriteLine("Changed?: {0} ", barnaBaby.IsChanged());
        Console.WriteLine("Sleeping?: {0} ", barnaBaby.IsSleeping());

    // Show your true colors Max...
        for(int i = 0; i < 10000; i++)
        barnaBaby.Cry();
```

As you will see, all CTS structures are derived from a common base class: **System.ValueType.** This base class configures a type to behave as a stack-allocated entity rather than a heap-allocated entity. CTS permits structures to implement any number of interfaces; however, structures may not function as the base type to other classes or structures and are therefore explicitly "sealed."

CTS Enumeration Types

Enumerations are a handy programming construct that allows you to group name/value pairs under a specific name.ific name. For example, assume you are creating a video game application that allows the user to select one of three player types (Wizard, Fighter, or Thief). Rather than keeping track of raw numerical values to represent each possibility, you could build a custom enumeration:

```
// A C# enumeration.

public enum PlayerType

{ Wizard = 100, Fighter = 200, Thief = 300 } ;
```

By default, the storage used to hold each item is a System.Int32 (i.e., a 32-bit integer), however it is possible to alter this storage slot if need be (e.g., when programming for a low memory device such as a Pocket PC). Also, the CTS demands that enumerated types derive from a common base class,System.Enum.

1.8 COMMON LANGUAGE SPECIFICATION (CLS)

To fully interact with other objects regardless of the language they were implemented in, objects must expose to callers only those features that are common to all the languages they must interoperate with. For this reason, the Common Language Specification (CLS), which is a set of basic language features needed by many applications, has been defined. The CLS rules define a subset of the Common Type System; that is, all the rules that apply to the common type system apply to the CLS, except where stricter rules are defined in the CLS. The CLS helps enhance and ensure language interoperability by defining a set of features that developer can rely on to be available in a wide variety of languages. The CLS also establishes requirements for CLS compliance; these help you determine whether your managed code conforms to the CLS and to what extent a given tool supports the development of managed code that uses CLS features.

If your component uses only CLS features in the API that it exposes to other code (including derived classes), the component is guaranteed to be accessible from any programming language that supports the CLS. Components that adhere to the CLS rules and use only the features included in the CLS are said to be CLS-compliant components.

Most of the members defined by types in the .NET Framework Class Library are CLS-compliant. However, some types in the class library have one or more members that are not CLS-compliant. These members enable support for language features that are not in the CLS. The types and members that are not CLS-compliant are identified as such in the reference documentation, and in all cases a CLS-compliant alternative is available. For more information about the types in the .NET Framework class library, see the .NET Framework Class Library.

The CLS was designed to be large enough to include the language constructs that are commonly needed by developers, yet small enough that most languages are able to support it. In addition, any language constructs that makes it impossible to rapidly verify the type safety of code was excluded from the CLS so that all CLS-compliant languages can produce verifiable code if they choose to do so. For more information about verification of type safety, see Managed Execution Process.

The Common Language Specification (CLS), which is a set of basic language features needed by many .Net applications to fully interact with other objects regardless of the language in which they were implemented. The CLS represents the guidelines defined by for the .NET Framework. These specifications are normally used by the compiler developers and are available for all languages, which target the .NET Framework.

The CLS helps enhance and ensure language interoperability by defining a set of features that developer can rely on to be available in a wide variety of languages.

- Any language that conforms to the CLS is a .NET language
- A language that conforms to the CLS has the ability to take full advantage of the Framework Class Library (FCL)
- CLS is standardized by ECMA

.NET Languages

- Languages provided by Microsoft
  - C++, C#, J#, VB.NET, JScript
- Third-parties languages
  - Perl, Python, Pascal, APL, COBOL, Eiffel, Haskell, ML, Oberon, Scheme, Smalltalk…
- Advanced multi-language features
  - Cross-language inheritance and exceptions handling
- Object system is built in, not bolted on
  - No additional rules or API to learn

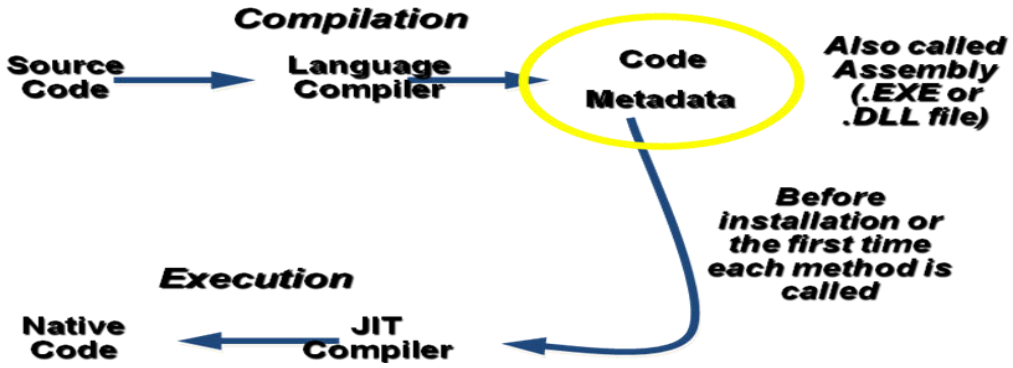
1.9 C# Language Fundamentals

- Mixture between C++, Java and Delphi
- Component-oriented
  - Properties, Methods, Events
  - Attributes, XML documentation
  - All in one place, no header files, IDL, etc.
  - Can be embedded in ASP+ pages
- Everything really is an object
  - Primitive types aren't magic
  - Unified type system == Deep simplicity
  - Improved extensibility and reusability

C# Language – Example

```
using System;
class HelloWorld
{
        public static void main()
        {
                Console.WriteLine("Hello, world!");
        }
}
```

Code Compilation and Execution



ASSEMBLIES

- DLL or EXE file
- Smallest deployable unit in the CLR
- Have unique version number
- No version conflicts (known as DLL hell)
- Contains IL code to be executed
- Security boundary – permissions are granted at the assembly level
- Type boundary – all types include the assembly name they are a part of
- Self-describing manifest – metadata that describes the types in the assembly

1.10 Classes

A *class* is a construct that enables you to create your own custom types by grouping together variables of other types, methods and events. A class is like a blueprint. It defines the data and behavior of a type. If the class is not declared as static, client code can use it by creating *objects* or *instances* which are assigned to a variable. The variable remains in memory until all references to it go out of scope. At that time, the CLR marks it as eligible for garbage collection. If the class is declared as static, then only one copy exists in memory and client code can only access it through the class itself, not an *instance variable*.

Declaring Classes

Classes are declared by using the class keyword, as shown in the following example:

```
Public class Customer
{
   //Fields, properties, methods and events go here...
}
```

The **class** keyword is preceded by the access level. Because public is used in this case, anyone can create objects from this class. The name of the class follows the **class** keyword. The remainder of the definition is the class body, where the behavior and data are defined. Fields, properties, methods, and events on a class are collectively referred to as *class members*.

Creating Objects

Although they are sometimes used interchangeably, a class and an object are different things. A class defines a type of object, but it is not an object itself. An object is a concrete entity based on a class, and is sometimes referred to as an instance of a class.

Objects can be created by using the new keyword followed by the name of the class that the object will be based on, like this:

```
Customer object1 = new Customer();
```

When an instance of a class is created, a reference to the object is passed back to the programmer. In the previous example, object1 is a reference to an object that is based on Customer. This reference refers to the new object but does not contain the object data itself. In fact, you can create an object reference without creating an object at all:

```
Customer object2;
```

This syntax is not recommended for creating object references because trying to access an object through such a reference will fail at run time. However, such a reference can be made to refer to an object, either by creating a new object, or by assigning it to an existing object, such as this:

```
Customer object3 = new Customer();
Customer object4 = object3;
```

This code creates two object references that both refer to the same object. Therefore, any changes to the object made through object3 will be reflected in subsequent uses ofobject4. Because objects that are based on classes are referred to by reference, classes are known as reference types.

1.11 String Formatting

String.Format Method

Converts the value of objects to strings based on the formats specified and inserts them into another string.

Formatting Console Output

.NET introduces a new style of string formatting, slightly reminiscentof the C printf() function, but without the cryptic %d, %s, or %c flags. A simple example follows

```
static void Main(string[] args)
{
...
int theInt = 90;
double theDouble = 9.99;
bool theBool = true;
// The '\n' token in a string literal inserts a newline.
Console.WriteLine("Int is: {0}\nDouble is: {1}\nBool is: {2}",
theInt, theDouble, theBool);
}
```

The first parameter to WriteLine() represents a string literal that contains optional placeholders designated by {0}, {1}, {2}, and so forth (curly bracket numbering always begins with zero). The remaining parameters to WriteLine() are simply the values to be inserted into the respective placeholders (in this case, an int, a double, and a bool).

:

```
// Fill placeholders using an array of objects.
   object[] stuff = {"Hello", 20.9, 1, "There", "83", 99.99933} ;
  Console.WriteLine("The Stuff: {0} , {1} , {2} , {3} , {4} , {5} ", stuff);
```

It is also permissible for a given placeholder to repeat within a given string. For example, if you are a Beatles fan and want to build the string "9, Number 9, Number 9" you would write

```
Console.WriteLine("{0}, Number {0}, Number {0}", 9);
```

1.12 SCOPE

Definition

The scope of a variable, sometimes referred to as accessibility of a variable, refers to where the variable can be read from and/or written to, and the variable's lifetime, or how long it stays in memory. The scope of a procedure or method refers to where a procedure can be called from or under what context you are allowed to call a method.

Scoping terms

Term	Used With…	Visibility
Public	Variables/Properties/Methods/Types	Anywhere in or outside of a project
Private	Variables/Properties/Methods/Types	Only in the block where defined
Protected	Variables/Properties/Methods	Can be used in the class where defined. Can be used within any inherited class.
Friend	Variables/Properties/Methods	Can only be accessed by code in the same project/assembly.
ProtectedFriend	Variables/Properties/Methods	Combination of Protected and Friend

Constants

- The variables whose values do not change during the execution of a program are known as constants.
- Const keyword is used to declare constants.

Eg: Const int n = 10;

Note: Names of constants take the same form as variables names

- After declaration of constants they should not be assigned any other value
- Constants can not be decalred inside a method. They should be declared onlyt at class level

1.13 Control Structures

ITERATION STATEMENTS

Iterative statements repeat a particular statement block until a condition has been satisfied.

Following types of iterative statements are there in C#.

1. While
2. Do While
3. For
4. For each

While Statement

The statement block of while statement is executed while the boolean expression is true. It may execute zero or more times. If the boolean expression is false initially, the statement block is not executed.

```
int i = 9;
int j = 7;
int sum = 0;
while (j < i)
{
sum += j;
j++;
}
```

The value in the *sum* variable will be 15 as this loop will continue 2 times only. As soon as the value of j will reach to 9 the condition j < i will return false and the loop will break.

The while statement is used to iterate while the condition evaluates to true.

Do While Statement

Do While is almost similar to While statement except it validate its Boolean expression after the statement block. It guarantees that the statement block shall run at least once for sure. Further iterations of the statement block continues while the Boolean expression is true.

```
int i = 9;
int j = 10;
int sum = 0;
do
{
sum += j;
j++;
} while (j < i);
```

The value in the *sum* variable will be 10 as j is already greater than i so after first time entering into the loop, the boolean validation (j < i)will return false and loop will break. The above code block may not be the good example of the do while loop; do while can be used to ask for the desired input from the user. If we are not getting the desired input, we can continue asking the question in the while statement block.

For Statement

The For statement iterate a code block until a specified condition is reached similar to while statement. The only difference of for statement has over while statement is that for statement has its own built-in syntax for intiliazing, testing, and incrementing/decrementing (3 clauses) the value of a counter.

The first clause is the initialize clause in which the loop iterators are declared.

The second clause is the boolean expression that must evaluate to a boolean type and the statement block is repeated until this expression is false.

The third clause is to increment/decrement the value that is executed after each iteration.

All three clauses must be separated by a semicolon (;) and are optional. For statement block is repeated zero or more times based on the boolean expression validation (second clause).

```
int j = 10;
int sum = 0;
for (int i = 0; i < j; i++)
{
sum += i;
}
```

The value of *sum* variable will be 45 (0+1+2+3+4+5+6+7+8+9). It will add all the number from 0 to 9 because once it will reach 10 (third clause increase the value of i after each iteration), the second clause (boolean expression) will not satisfy and loop will break.

For Each Statement

For Each statement is designed to loop through a similar type of items in a collection. As each element is enumerated, the identifier is assigned the new element, and the statement block is repeated. The scope of the identifier is within the for each statement block. The identifier must be of the same type extracted from the collection and is read-only.

```
string[] days = { "sunday", "monday", "tuesday" };
string output = string.Empty;
foreach (string s in days)
{
output = string.Concat(output, s + " > ");
}
```

The identifier of the above *foreach* loop is *string s* (because *days* array contains string so identifier must be of the same type). The value of the *output* variable will be "sunday > monday > tuesday > ". In the *days* array, we have three strings and in the for each loop we are concatenating all the items of the loop.

Use of break and continue

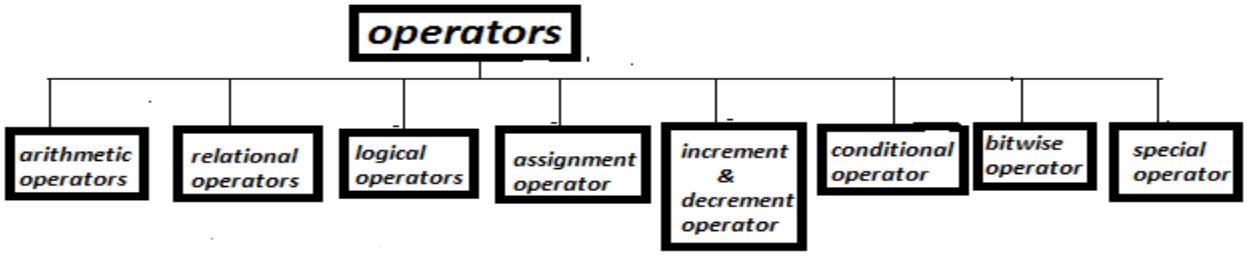
If we want to break the loop for some reason (may be after looping and validating the item, we got the desired result); we can use break (break;) statement to prematurely exist from the loop.

If we want to skip a particular iterations and continue next iteration; we can use continue (continue;) statement. Continue statement transfers the control to the end of the statement block where the next execution continues.

1.14 OPERATORS

An operator is a symbol that tells the computer to perform mathematical or logical manipulations.

Operators are used in programs to manipulate data and programs.



Arithmetic operators:

Operators	meaning
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	modulo division

- The Operators +,-,* and / all work the sdame way as they do in other languages.
- These can operate any built-in numeric data type.we cannot use these operators in Boolean type.

Arithmetic Operators are used as:

a-b a+b
a*b a/b
a%b -a*b

Here a and b may be variables or constants and are known as Operands.

Integer Arithmetic

- When both the operands in a single arithmetic expression such as a+b are integers,the expression is called an integer expression,and the operation is called integer arithmetic.
- Integer arithmetic always yields an integer value.

Example:
a=14 and b=4
a-b=10
a+b=18
a*b=56
a/b=3

Real Arithmetic

- An arithmetic Operation involving only real operands is called real arithmetic.
- A real operand may assume values either in decimal or exponential notation.
- Modulus operand returns the floating-point equivalent of an intiger division.

Example:
a=20.5
b=6.4
a+b=26.9
a-b=14.1
a*b=131.2
a/b=3.203125

Mixed-mode Arithmetic

- When one of the operands is real and other is integer, the expression is called a mixed-mode arithmetic expression.

Example:
15/10.0 produces the result 1.5
15/10 produces the result 1

Relational Operators

- We often compare two quantities and depending on their relation, take certain decisions,For example,we may compare the age of two persons or the price of two items and so on.These Comparisions can be done with the help of relational operators.

Example:

a<b or x<20

Expression containing a relational operator is termed as a relational expression. The value of a relational expression is either true or false. For example, Relational expression are

used in decision statements such as if and while to decide the course of action of a running program.

Operators	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

- The logical operators && and ll are used when we want to form compound condition by combining two or more relations. An example is:
  a>b && x==10

- An expression of this kind which combines two or more relational expression is termed as a logical expression or a compound expression. Like the simple relational expressions, a logical expression also yields a value of true or false.

operators	meaning
&&	logical AND
| |	logical OR
!	logical NOT
&	bitwase logical AND
|	bitwise logical OR
^	bitwise logical exclusive OR

Assignment Operators

- Assignment operators are used to assign the value of an expression to a variable.
- Assignment operator, '='.

Example:
v OP=exp
Where v is a variable ,exp is an expression and op is a C# binary operator.

- The operator OP= is known as the shorthand assignment operator.
- The shorthand operator y+=x means 'add y+1 to x' or 'increment x by y+1'. X +=3;
- The use of shorthand assignment operators has three advantages

*What appears on the left-hand side need not be repeated and therefore it becomes easier to write

*The statement is more concise and easier to read

*The use of shorthand operators results in a more efficient code.

Statement with simple assignment operators	statements with shorthand operator
a=a+1	a+=1
a=a-1	a-=1
a=a*(n+1)	a*=n+1
a=a/(n+1)	a/=n+1
a=a%b	a%=b

Increment and Decrement Operators

- There are the increment and decrement operators:

++ and --

- The operator ++ adds 1to the operand while—subtracts 1.
- Both are unary operators and are used in the following form:

++m; or m++;
--m; or m--;
++m; is equivalent to m=m+1; (or m + = 1;)
--m; is equivalent to m=m-1; (or m - = 1;)

- We use the increment and decrement operators extensively in for and while loops,
- While ++m and m++ mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement, Consider the following:

```
m=5;
y= ++m;
```

Example:

```
m=10
n=20
++m=11
n++=20
m=11
n=21
```

Conditional Operator

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

  exp1?exp2:exp3

  where exp1,exp2,exp3 are the expressions.

- The operator?: works as follows: exp1 is evaluated first. If it is true, then the expression exp2 is evaluated and becomes the value of the conditional expression. if exp 1 is false,exp3 is evaluated and its value becomes the value of the conditional expression. Note that only one of the expressions (either exp2 or exp3) is evaluated .For example, consider the following statements:

```
a=10;
b=15;
x=(a>b) ? a:b; equivalent to,
    if (a > b)
    x=a;
    else
    x=b;
```

Bitwise Operators

C# supports the operators that may be used for manipulation of the 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 to 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)
Checked (overflow checking)
Unchecked (prevention of overflow checking)

1.15 Array

Array is the group of related data items. Length property is used to find and the length of the array.

One dimensional Array:

Syntax: datatype[] arrayname = new datatype[size];

eg: int[] a = new int[10];

Initialization of One Dimensional Array:

```
int[] a={10, 20, 30};
```

Accessing array elements:

a[1] => 1st element of array a

Eg:Program to find sum of Array elements

```
using System;
```

```
class DemoArray
        {
        public static void Main()
                {
                int[] a={10,20,30};
                int s=0;
                for(int i=0; i<3; i++)
                        s=s+a[i];
                Console.WriteLine("Sum is "+s);
                }
        }
```

Output : Sum is 60

Multi Dimensional Array:

Contains many rows.

Syntax:

```
datatype[,] arrayname=new datatype [row size][column size];
```

eg:

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

Accessing array elements:

a[2,3]=> 2nd row, 3rd element

Eg:Program to print elements of two dimensional array

```
class TDArray
{
public static void Main()
        {
        int[ , ] a={{10,20},{30,40}};
        for (int i=0; i<a.GetLength(0); i++)
                {
                for (int j=0; j<a.GetLength(1); j++)
                        {
                        Console.Write(a[i,j]);
                        }
                Console.WriteLine("\n");
                }
        }
}
```

Output :

10 20
30 40

Jagged Array:

Jagged Array contains some numbers of inner arrays, each of which may have a Unique Upper Limit.

Syntax:

```
int[][] JA=new int[2][];
```

Ex:

```
JA[0]=new int[2];
JA[1]=new int[3];
```

Now the jagged array contains 2 rows.

1st row contains 2 elements,
2nd row contains 3 elements.

Length Property:

For one and multi dimensional arrays, the length property returns no. of elements in that array.

Length can be used for Jagged Arrays. Here it is possible to obtain the length of each individual array.

Eg:

JA.Length = 2[no. of rows]
JA[1].Length = 2[no. of elements in 1st row

Note:

By default the values of array elements are zero.

Methods of class System.Array:

C# array is derived from System.Array (.Net Base Type). Hence C# arrays can use the members of System.Array.

Binary Search()
Reverse()
Clear()
Sort()

eg: int a=(40, 20, 30, 10}

int i=Array.BinarySearch(a,20)
Now i=1

Array.Sort(a)
Now a= {10, 20, 30, 40}
Array.Reverse(a)
Now a={40, 30, 20, 10}
Array.Clear(a, 0, 2)
Now a={0, 0, 20, 10}

1.16 String

One of the data type is string. Used to store strings.Strings are objects.Hence it is a reference type.

Declaration of strings:

string s1= ”Hello”;

String is derived from the base class System.String (.Net base type). By means of index, you can access individual character but cant modify it.
[Eg: s1[0]='H']

Methods of class System.String:

(1) public static string Copy(string s1)
Used to copy the string.
Eg: string s1=”Hello”
string s2=String.Copy(s1);
s2=”Hello”.

(2)public static int Compare(string s1, string s2, bool ignorecase)
Used to compare strings.
ignorecase --> true =>ignore case
ignorecase --> false => wont ignore case
eg: string s1=”abc”
string s2=”ABC”
int i=String.Compare(s1,s2,false)
i=65-97= -32 [s1<s2]

(3) public static string concat (string s1, string s2)
Used to concatenate two strings.
Eg: string s1 = “Good”
string s2 = “Morning”
string s3 = String.concat(s1,s2);
Now string3 = Good Morning

(4) public int IndexOf(string str or char c)

=>Searches the invoking string for the substring or character and returns the index of the first match.

Eg: string s1=”Good”;
int i = s1.Index Of(‘o');
Now i=1

(5) public int LastIndexOf(string str) or (char c)

=> Searches the invoking string for the substring or character and returns the index of the last match.

Eg: string s1 = “Good”
int i = s1.LastIndexOf ('o');
Now i = 2

(6)public bool StartsWith(String str)

Used to check whether the string starts with given substring.It returns true or false.

Eg: string s1=”Good”
bool b = s1.StartsWith(“Good”);
b=true

(7) public bool EndsWith(string str)

Used to check whether string ends with the given substring.

Eg: string s1=”Good Morning”;
bool b=s1.EndsWith(“ING”);
b=false

(8) public string Tolower() & Toupper()

Used to convert characters from uppercase to lowercase

eg: String S1 = “Good”;
String S2 = S1.ToLower();
S2 = good
String S3 = S2.ToUpper();
S3 = GOOD

(9) Public string Replace(character to be replaces, character for replacement)

Used to replace one character with other.

Eg: String S1 = “Geed”;
String S2 = S1.replace('e', 'o');
S2 = Good

(10) public string insert(string str, int location)

Used to insert the string at the specified location

Eg:

String S1 = This Good

12345

String S2 = S1.insert("is", 5);

S2 = This is good.

(11) public string remove(int start, int count)

Used to remove the count no of character from the given string starting from the location specified by start.

Eg: String S1 = "This is good";
String S2 = S1.remove(5,2);
S2 = This good

1.17 ENUMERATIONS

An ENUMERATION provides an efficient way to define a set of named integral constants that may be assigned to a variable.

- The Enumeration list is a comma separated list of identifiers.
- By default,the value of the first enumeration symbol is 0.

syntax:

enum name [enumeration list];

Exmaple

```
using System;
class Program
{
enum Importance
  {
       None,
       Trivial,
       Regular,
       Important,
       Critical
  };
```

```
static void Main()
{
Importance value = Importance.Critical;
if (value == Importance.Trivial)
{
Console.WriteLine("Not true");
}
else if (value == Importance.Critical)
{
Console.WriteLine("True");
}
}
}
```

Output : True

1.18 STRUCTURE

Definition

Structure is a value data type . It is used to store mixed data types. It provides a unique way of packing different data types together. It is convenient tool for handling a group of logically related data items.

Syntax :

```
struct struct-name
        {
                data type member1;
                data type member2;
        }

        eg:
                struct Student
                {
                        public string name;
                        public int rollnumber;
                }
```

Program for displaying student detail using structure

```
using System;
public struct Student
{
        public string name;
        public int rollnumber;
}

class Csstruct
{
        public static void Main ()
        {
                Student S1;
                S1.name = "lakshmi";
                S1.rollnumber = 101;
                Console.WriteLine(S1.name+"\t"+S1.rollnumber);
        }
}
```

Output : lakshmi 101

Features of C# structure

Structures support defined constructors, but not destructors. The default constructor is automatically defined and cannot be changed.
Structures can have methods.
Structure Variable can be passed as a parameter to any member function.

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

Program for displaying student details by defining constructors

eg:

```
struct Student
{
        public string name;
        public int rollno;
}
class CsStruct
{
        public static void Main ()
        {
                Student S1= new student("lakshmi", 101);
                Console.WriteLine(S1.name+"\t"+S1.rollno);
        }
}
```

Output : lakshmi 101

Program for displaying student details by defining methods inside the structure.

eg:

```
struct student              //structure
{
        public string name;
        public void  Display(string n, int r )     // method
        {
                name =n;
                rollno =r;
        }
}
class csstruct
{
        public static void Main()
        {
                student s1 = new student();       //structure object
                s1. Display ("lakshmi",101)
                Console.WriteLine(s1.name);
                Console.WriteLine(s1.rollno);

        }
}
```

Output : lakshmi 101

Program for displaying student details by passing structure as parameters to any member function.

```
Eg:
struct student
{
        string name;
        int rollno;
}
class csstruct
{
        public static Void display(student s1)
        //sturcture cannot be inherited they can be passed as parameter
        {
                Console.WriteLine(s1.name+"\t"+s1.rollno);
        }
        public static Void Main()
        {
                student s1;
                s1.name = "lakshmi";
                s1.rollno = 101;
                display(s1);
        }
}
```

Output : lakshmi 101

Syntax for Copying one structure to another:

```
eg: student s1,s2;
        s2=s1;//now s1 is copied to s2.
```

Syntax for using Struct variable as a member for another structure

```
eg:
        struct M
        {
                public int x;
        }
        struct N
        {
                public M m;
                public int Y;
        }
```

1.19 DEFINING CUSTOM NAMESPACES

We can also define our own namespaces using the keyword **namespace**.

Syntax :

```
namespace namespace_name
{
        public class calss1{}
                .....
        public class class2{}
}
```

Example program for defining custom namespace **named Myshapes**

```
using System;
namespace Myshapes //Myshapes is the user defined namespace
{
    public class circle
    {
        public void display()
        {
         Console.WriteLine("You have called display method of circle ");
        }
}
public class square
{
          public void display()
         {
         Console.WriteLine("You have called display method of square ");
         }

    }
}
```

Program for accessing custom namespace Myshapes

eg:

```
using System;
using Myshapes; //import the desired namespace
class ex
{
    public static void Main()
    {
```

```
                circle c = new circle(); //circle object of the desired class
                square s = new square();
                c.display();
                s.display();
        }
}
```

Output : You have called display method of circle
You have called display method of square

Resolving Name classes across Namespaces

Fully qualified name resolves the name clash across multiple namespaces

eg:

One more Namespace My3DShapes that also contain the same class cirlce.

```
using System;
Namespace My3Dshapes
{
        public class circle
        {}
        public class Square
        {}
}
```

Now we can differentiate circle class of MyShapes and My3DShapes by means of fully qualifed names.

Eg:

```
using MyShapes;
using My3DShapes;
class ex
{
        PSVM()
        {
                MyShapes.Circle c1 = new MyShapes.circle();
                My3DShapes.Circle c2 = new My3DShapes.Circle();
        }
}
```

Nested namespace

Namespaces can also be nested within other namespaces

Syntax

```
using System;
namespace A
{
        namespace B
        {
        define classes //here B within A known as nested Namespaces
        }
}
```

1.20 Object Oriented Programming with C#

There are three major object oriented concepts

Encapsulation
Inheritance
Polymorphism

Encapsulation

This is the languages ability to hide unnecessary implementation details from the end user. Binding of data and functions into the single unit is also known as Encapsulation.

public data members of the class can be accessed by means of object.
Eg:

```
class F
{
        public int a = 20;
}
class F1
{
        public static void Main()
        {
                F  b = new  F();
                Console.WriteLine("Value of a is  " , b.a);
        }
}
}
```

Output : Value of a is 20

There are two ways to access private data numbers

By means of defining two public methods
By means of defining properties

Accessing private data by means of defining public methods

Eg:

```
using System;
class A
{
        Private int a;
        public void set1(int x)
        {
                a = x;
        }
        public int get1()
        {
                return(a);
        }
}
class B
{
        public static void Main()

        {
                A a1 = new A();
                a1.set1(10);//access private data by means of public method.
                Console.WriteLine("Value of private data a is " + (a1.get1());
        }
}
```

Output : Value of private data a is 10

Accessing private data by means of defining properties

This is also used to access private data. Instead of using two public methods for getting and setting values for private data, we can use single property.

Syntax:

```
type name
{
        get {}
        set {}
}
```

Type --> type of the property
Name --> name of the property

Once the property has been defined, any use of name results in a call to its appropriate accessor. The set accessor automatically receives a parameter called calue that contains the value being assigned to the property.

To manipulate internal data using single variable

eg:

```
using System;
class A
{
        Private int a;
        no modifier int prop //syntax for property
        {
                get
                {
                        return(a);
                }
                set
                {
                        a = value;
                }
        }       //property
}       //class
class B
{
        public static void Main()
        {
        A a1 = new A();
        a1.prop(10);//call set accessor of property prop and set value of private data
        Console.WriteLine("Value is "+a1.Prop)//call get accessor of property prop
        }
}
```

Three Types of properties

Read Only Property
Write Only Property
Static Property

If the property contains only get block then that property is known as read only property.

Eg:

```
int Prop
{
        get{}
}
```

If the property contains only set block then that property is known as write only property

```
Eg:
int Prop
{
        Set
        {
                Private data = value;
        }
}
```

Static property

If the property is defined as static then that can be accessed by means of class name.

```
Eg:
class A
{
        Private static int a;
        public static int prop
        {
                get
                {
                        return(a);
                }
                set
                {
                        a = value;
                }
        }
}
class B
{
public static void Main()
{
A.prop =10;// set accessor of static property prop is called with class name .
Console.WriteLine("Value is "+A.prop); // call get accessor of static property prop
}
}
```

1.21 Inheritance

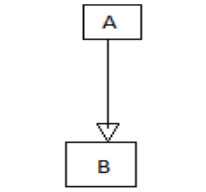
This concept is mainly used for code reuse . Inheritance comes in two flavours

is-a relationship (classical inheritance)
has-a relationship(containment)

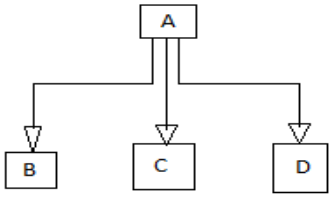
Classical inheritance

Defnition:- the basic idea behind is that new classes may leverage the functionality of other classes.

Eg:



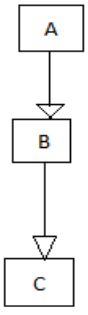
Single Inheritance



Hierarchial Inheritance

class A --> baseclass, parent class, super class
class B --> derived class, child class, sub class

Now the derived class that incorporate all the data and methods of its base class, have its own data member. Now B is a type of A .Classical inheritance can be implemented in different combinations.



Multilevel Inheritance

Containment inheritance:

Containership between class A & B

Ex:

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

the relationship between A and B is has a relationship.

General form of defining sub-class:-

Syntax:

```
class derived class name:base class name
{
        variable declaration;
        method declaration;
}
```

ex:

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

Example program for single inheritance

```
class A
{
        public void display1()
        {
                Console.WriteLine("Base class");
        }
}
class B:A
{
        public void display2()
        {
                Console.WriteLine("Derived class");
        }
}
```

```
class C
    {
        public static void Main()
        {
            B b1 = new B();
            b1.display1();
            b1.display2();
        }
    }
```

OUTPUT :

Base Class
Derived class

Base Keyword

Used to call base class constructor from derived class
Used to access base class data member from derived class

Example program to call base class constructor from derived class

```
    class A
    {
        public int i;
        public A(int x)
        {
            i = x;
        }
    }
    class B:A
    {
        public int i1;
        public B(int x, int y):base(x)//call the base class constructor
        {
            i1 =y;
        }
    }
    class C
    {
        public static void Main()
        {
            B b1 = new B(10,20);
            Console.WriteLine("Value of i is " +b1.i);
            Console.WriteLine("Value of i1 is "+b1.i1);
        }
    }
```

OUTPUT :

Value of i is 10
Value of i1 is 20

Note: if the base class and derived class has the same member, then the base class member can be accessed from the derived class using the keyword base.
Example program to access base class data member from derived class

```
class A
{
        public int i = 10;
}
class B:A
{
        public int i = 20;
        public void display();
        {
                Console.WriteLine("Value of base class i "+base.i);
                Console.WriteLine("Value of derived class i"+i);
        }
}
class c
{
        public static void Main()
        {
                B b1 = new B();
                b1.display();
        }
}
```

OUTPUT :

Value of base class i 10
Value of derived class i 20

Prevent inheritance using keyword sealed:-

```
sealed class A
{}
class B:A//error because A cannot be inherited
{}
```

1.22 Polymorphism

When a message can be processed in different ways is called polymorphism. Polymorphism means many forms.

Polymorphism is one of the fundamental concepts of OOP.

Polymorphism provides following features:

- It allows you to invoke methods of derived class through base class reference during runtime.
- It has the ability for classes to provide different implementations of methods that are called through the same name.

Polymorphism is of two types:

1. Compile time polymorphism/Overloading
2. Runtime polymorphism/Overriding

Compile Time Polymorphism

Compile time polymorphism is method and operators overloading. It is also called early binding.

In method overloading method performs the different task at the different input parameters.

Runtime Time Polymorphism

Runtime time polymorphism is done using inheritance and virtual functions. Method overriding is called runtime polymorphism. It is also called late binding.

When **overriding** a method, you change the behavior of the method for the derived class. **Overloading** a method simply involves having another method with the same prototype.

Following are examples of methods having different prototypes:

```
void area(int side);
void area(int l, int b);
void area(float radius);
```

Program for Method Overloading (Compile Time Polymorphism)

```
using System;

   class Program
   {

         public void display(string name)
         {
            Console.WriteLine("Your name is : " + name);
         }

         public void display(int age, float marks)
         {
            Console.WriteLine("Your age is : " + age);
            Console.WriteLine("Your marks are :" + marks);
         }
      }

      static void Main(string[] args)
      {

         Program obj = new Program();
         obj.display("George");
         obj.display(34, 76.50f);
         Console.ReadLine();
      }
   }
}
```

Method Overriding(Run time Polymorphism)

When a derived class inherits from a base class, it gains all the methods, fields, properties and events of the base class. To change the data and behavior of a base class, you have two choices: you can replace the base member with a new derived member, or you can override a virtual base member.

Replacing a member of a base class with a new derived member requires the new keyword. If a base class defines a method, field, or property, the **new** keyword is used to create a new

definition of that method, field, or property on a derived class. The **new** keyword is placed before the return type of a class member that is being replaced. For example:

```
public class BaseClass
{
   public void DoWork() { }
   public int WorkField;
   public int WorkProperty
   {
      get { return 0; }
   }
}

public class DerivedClass : BaseClass
{
   public new void DoWork() { }
   public new int WorkField;
   public new int WorkProperty
   {
      get { return 0; }
   }
}
```

When the **new** keyword is used, the new class members are called instead of the base class members that have been replaced. Those base class members are called hidden members. Hidden class members can still be called if an instance of the derived class is cast to an instance of the base class. For example:

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Calls the old method.
```

In order for an instance of a derived class to completely take over a class member from a base class, the base class has to declare that member as virtual. This is accomplished by adding the virtual keyword before the return type of the member. A derived class then has the option of using the override keyword, instead of **new**, to replace the base class implementation with its own. For example:

```
public class BaseClass
{
   public virtual void DoWork() { }
   public virtual int WorkProperty
   {
      get { return 0; }
   }
}
public class DerivedClass : BaseClass
{
   public override void DoWork() { }
   public override int WorkProperty
   {
      get { return 0; }
   }
}
```

Fields cannot be virtual; only methods, properties, events and indexers can be virtual. When a derived class overrides a virtual member, that member is called even when an instance of that class is being accessed as an instance of the base class. For example:

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Also calls the new method.
```

Virtual methods and properties allow you to plan ahead for future expansion. Because a virtual member is called regardless of which type the caller is using, it gives derived classes the option to completely change the apparent behavior of the base class.

Virtual members remain virtual indefinitely, no matter how many classes have been declared between the class that originally declared the virtual member. If class A declares a virtual member, and class B derives from A, and class C derives from B, class C inherits the virtual member, and has the option to override it, regardless of whether class B declared an override for that member. For example:

```
public class A
{
   public virtual void DoWork() { }
}
public class B : A
{
   public override void DoWork() { }
}


public class C : B
{
   public override void DoWork() { }
}
```

A derived class can stop virtual inheritance by declaring an override as sealed. This requires putting the sealed keyword before the **override** keyword in the class member declaration. For example:

```
public class C : B
{
   public sealed override void DoWork() { }
}
```

In the previous example, the method DoWork is no longer virtual to any class derived from C. It is still virtual for instances of C, even if they are cast to type B or type A. Sealed methods can be replaced by derived classes using the **new** keyword, as the following example shows:

```
public class D : C
{
   public new void DoWork() { }
}
```

In this case, if DoWork is called on D using a variable of type D, the new DoWork is called. If a variable of type C, B, or A is used to access an instance of D, a call to DoWork will follow the rules of virtual inheritance, routing those calls to the implementation of DoWork on class C.

A derived class that has replaced or overridden a method or property can still access the method or property on the base class using the base keyword. For example:

```
public class A
{
   public virtual void DoWork() { }
}
public class B : A
{
   public override void DoWork() { }
}


public class C : B
{
   public override void DoWork()
   {
      // Call DoWork on B to get B's behavior:
      base.DoWork();

      // DoWork behavior specific to C goes here:
      // ...
   }
}

}
```

1.23 Type conversion and casting

When ever smaller datatype is assigned to larger datatypes then conversion is performed automatically. Known as **widening conversion**.

Eg:
```
int l;
byte b;
l = b //automatic conversion
```

When larger datattype is assigned to smaller type then casting should be performed known as **narrowing conversion**.

Eg:
```
int i;
byte b;
b = (byte)i; //casting
```

Boxing & Unboxing

Boxing:

Any type value a reference can be assigned to an object without an explicit conversion is known as Boxing.

```
Eg:   int i = 10;
      object  o = i;   //Boxing
```

UnBoxing:

Unboxing is the process of converting the object type back to value type.

```
Eg:   int i1 = (int)o;
```