School of Computer Applications & Information Technology



Department of Master of Computer Applications C# & .NET Technologies – 18MCA304

Unit 1 – Introduction to C# & .NET

## Kathiresan V Assistant Professor

School of Computer Science & IT JGI Knowledge Campus Jayanagar 9th block

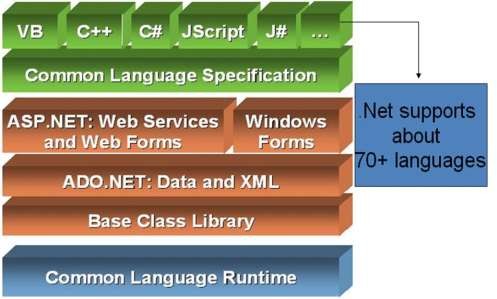
Bengaluru - 69

### Mail Id:

[**v.kathiresan@jainuniversity.ac.in**](mailto:v.kathiresan@jainuniversity.ac.in)

**Components of .Net Framework**

**Components of .Net Framework**



Net Framework is a platform that provides tools and technologies to develop Windows, Web and Enterprise applications. It mainly contains two components,

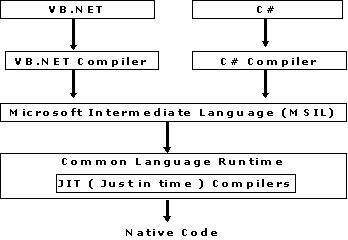
1. Common Language Runtime (CLR)
2. .Net Framework Class Library.

### Common Language Runtime (CLR)

**.Net Framework** provides runtime environment called **Common Language Runtime** (CLR). It provides an environment to run all the .Net Programs. The code which runs under the CLR is called as **Managed Code**. Programmers need not to worry on managing the memory if the programs are running under the CLR as it provides memory management and thread management.

Programmatically, when our program needs memory, CLR allocates the memory for scope and de-allocates the memory if the scope is completed.

Language Compilers (e.g. C#, VB.Net, J#) will convert the Code/Program to **Microsoft Intermediate Language** (MSIL) intern this will be converted to **Native Code** by CLR. See the below Fig.



There are currently over 15 language compilers being built by Microsoft and other companies also producing the code that will execute under CLR.

### .Net Framework Class Library (FCL)

This is also called as Base Class Library and it is common for all types of applications i.e. the way you access the Library Classes and Methods in VB.NET will be the same in C#, and it is common for all other languages in .NET.

The following are different types of applications that can make use of .net class library.

* 1. Windows Application.
  2. Console Application
  3. Web Application.
  4. XML Web Services.
  5. Windows Services.

In short, developers just need to import the BCL in their language code and use its predefined methods and properties to implement common and complex functions like reading and writing to file, graphic rendering, database interaction, and XML document manipulation.

### .NET aware Languages

Understand that C# is not the only language that can be used to build .NET applications. When the .NET platform was first revealed to the general public during the 2000 Microsoft Professional Developers Conference (PDC), several vendors announced they were busy building .NET-aware versions of their respective compilers.

DotNet Languages list

1. [APL](http://msdn.microsoft.com/vstudio/partners/language/dyadicsystems.asp)
2. ASNA Visual RPG[RPG.NET](http://www.asna.com/pages/products_NET_AVR.aspx)
3. Boo Codehaus [Boo – python inspired syntax](http://boo.codehaus.org/)
4. [Fujitsu COBOL](http://www.adtools.com/info/whitepaper/net.html)
5. [Micro Focus Cobol NetExpress](http://www.microfocus.com/products/netexpress/)
6. [Microsoft C#](http://msdn.microsoft.com/vstudio/nextgen/technology/csharpintro.asp)
7. [F#](http://research.microsoft.com/projects/ilx/fsharp.aspx) (a mixed functional/imperative anguage based on [Caml](http://caml.inria.fr/) from Microsoft Research)
8. [Eiffel](http://www.dotnet.eiffel.com/)
9. [Delta Forth](http://www.codeproject.com/dotnet/dforthnet.asp)
10. [Lahey/Fujitsu Fortran for .NET](http://www.lahey.com/dotnet.htm)
11. [Salford Fortran](http://www.salford.co.uk/compilers/ftn95/dotnet.shtml)
12. [Hugs98](http://galois.com/~sof/hugs98.net/) Haskell
13. [Glasgow Haskell](http://haskell.cs.yale.edu/ghc)
14. [Microsoft J#](http://msdn.microsoft.com/visualj)
15. [Microsoft Jscript](http://msdn.microsoft.com/workshop/languages/clinic/scripting07142000.asp)
16. Microsof[t clisp](http://weblogs.asp.net/jtobler/archive/2003/10/01/30000.aspx) (bundled with Framework SDK)
17. [Oberon](http://www.oberon.ethz.ch/oberon.net/)
18. [ActiveState Perl.NET](http://aspn.activestate.com/ASPN/Downloads/PerlNET/More)
19. [ActiveState Python.NET](http://aspn.activestate.com/ASPN/NET/)
20. [Mark Hammond Python.NET](http://starship.python.net/crew/mhammond/dotnet/)
21. [Mercury.NET](http://www.cs.mu.oz.au/research/mercury/dotnet.html)
22. [Mondrian](http://www.mondrian-script.org/)
23. [Component Pascal](http://www2.fit.qut.edu.au/CompSci/PLAS/ComponentPascal)
24. [TMT Pascal](http://www.tmt.com/net.htm)
25. [IronPython](http://www.ironpython.com/)
26. [Microsoft IronPython](http://www.gotdotnet.com/workspaces/workspace.aspx?id=ad7acff7-ab1e-4bcb-99c0-57ac5a3a9742)
27. [Tachy](http://radio.weblogs.com/0101156/stories/2002/03/19/tachy.html) – subset of Scheme
28. [HotDog Scheme](http://hotdog.sourceforge.net/)
29. [Smalltalk SmallScript](http://www.smallscript.net/)
30. [SML.NET](http://www.cl.cam.ac.uk/Research/TSG/SMLNET/)

### Advantages of .NET:

* **Object Oriented**

Everything that you see in the .NET framework is an object. It is the same for what you write within the framework. This means that you get a powerful tool to not just access but also control your apps. This also makes it simpler for you to respond to recurring events.

### Caching

The caching system that .NET includes is extremely robust and easy-to-use.

### Easy Maintenance

Pages, with .NET, are extremely simple to write and maintain. This is because the source code and HTML are both together. In addition to that, the source code executes on the server. What does this mean? This makes your web pages more powerful and flexible.

### Time-Saving

Time is money, and .NET helps you save a lot of that. The way it is developed, .NET removes a large part of the coding requirement. This means that the developers save time, and the app’s time-to-market can be shortened considerably.

### Simplicity

Performing common tasks with .NET is extremely simple and straight forward. Submission of forms is a breeze and so is site configuration, deployment, and client authentication.

### Feature-Rich

There are a range of features that can be explored by the developers in order to create powerful apps. Consider the case of its rich toolbox as also the designer in the visual studio. They let you access such features as automatic deployment, WYSIWYG editing, and drag-and-drop controls.

### Consistency

The management and monitoring of all the processes is performed by the framework. If one of the processes is dead, a new process can be created just as easily. This lets your app be consistently available for handling requests.

### Monitoring

Finally, .NET also stands for its automatic monitoring. It will promptly notice any problems like infinite loops, memory leaks, etc. Not just this, it will also destroy these activities automatically and restart itself.

### C# LANGUAGE OFFERS THE FOLLOWING FEATURES

* No pointer required. C# programs typically have no need for direct pointer manipulation.
* Automatic memory management through garbage-collection. Given this, C# does not support a delete keyword.
* Formal syntactic-constructs for enumerations, structures and class properties.
* C++ like ability to overload operators for a custom type without the complexity.
* Full support for interface-based programming techniques.
* Full support for aspect-based programming techniques via attributes. This brand of development allows you to assign characteristics to types and their members to further qualify the behavior.

### THE BUILDING BLOCKS OF THE .NET PLATFORM

.NET can be understood as

* + a new runtime environment &
  + a common base class library (Figure:1-1) Three building blocks are:
  + CLR (Common Language Runtime)
  + CTS (Common Type System)
  + CLS (Common Language Specification) Runtime-layer is referred to as CLR.

Primary role of CLR: to locate, load & manage .NET types.

In addition, CLR also takes care of

* + automatic memory-management
  + language-integration &
  + ensuring type-safety

CTS

o describes all possible data-types & programming-constructs supported by runtime

o specifies how these entities interact with each other &

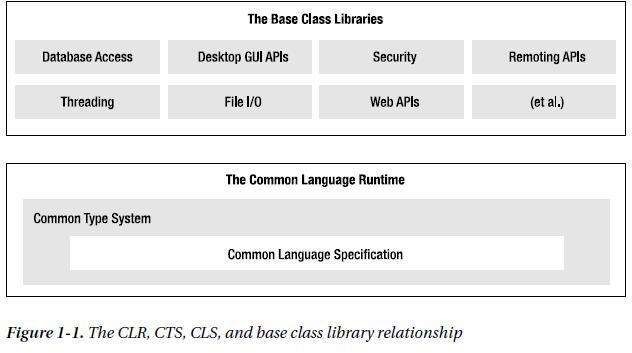
o specifies how they are represented in metadata format

CLS define a subset of common data-types & programming-constructs that all .NET aware- languages can agree on. Thus, the .NET types that expose CLS-compliant features can be used by all .NET-aware languages. But, a data type or programming construct, which is outside the bounds of the CLS, may not be used by every .NET programming language.

### The Role of Base Class Library (BCL)

BCL is a library of functionality available to all .NET-aware languages. This encapsulates various primitives for

* + file reading & writing
  + threads
  + file IO
  + graphical rendering
  + database interaction
  + XML document manipulation
  + programmatic security
  + construction of web enabled front end



### Intermediate Language (IL)

Intermediate language (IL) is an object-oriented programming language designed to be used by compilers for the .NET Framework before static or dynamic compilation to machine code. The IL is used by the .NET Framework to generate machine-independent code as the output of compilation of the source code written in any .NET programming language.

IL is a stack-based assembly language that gets converted to bytecode during execution of a virtual machine. It is defined by the common language infrastructure (CLI) specification. As IL is used for automatic generation of compiled code, there is no need to learn its syntax.

This term is also known as Microsoft intermediate language (MSIL) or common intermediate language (CIL). This is the language code generated by the C# compiler or any .NET-aware compiler. All .NET languages generate this code. This is the code that is executed during runtime.

You can view this MSIL code 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.

### AN OVERVIEW OF .NET BINARIES

* Regardless of which .NET-aware language you choose (like C#, VB.NET, VC++.NET etc), →.NET binaries take same file-extension (.exe or .dll)
  + .NET binaries have absolutely no internal similarities (Figure:1-2)
* .NET binaries do not contain platform-specific instruction but rather platform-agnostic "Common Intermediate Language (CIL)”.
* When .NET binaries have been created using a .NET-aware compiler, the resulting module is bundled into an *assembly.*
* An assembly is a logical grouping of one or more related modules (i.e. CIL, metadata, manifest) that are intended to be deployed as a single unit.
* An assembly contains CIL-code which is not compiled to platform-specific instructions until absolutely-necessary.
* Typically "absolutely-necessary" is the point at which "a block of CIL instructions" are referenced for use by the runtime-engine.
* The assembly also contains *metadata* that describes the characteristics of every "type" living within the binary.
* Assemblies themselves are also described using metadata, which is termed as *manifest.*

The manifest contains information such as

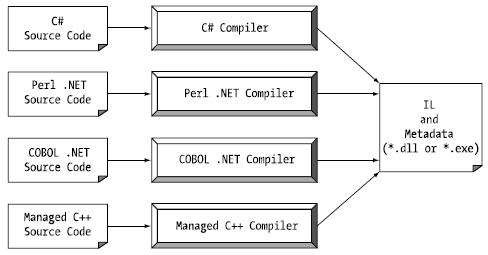
* + name of assembly/module
  + current version of assembly
  + list of files in assembly
  + copyright information
  + list of all externally referenced assemblies that are required for proper execution

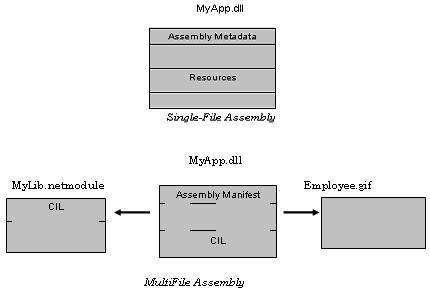
Figure 1-2: All .NET-aware compilers emit IL instructions and metadata

### Managed code & unmanaged code

C# produces the code that can execute within the .NET runtime. The code targeting the .NET runtime is called as managed-code.

Conversely, code that cannot be directly hosted by the .NET runtime is termed unmanaged- code.

### SINGLE-FILE AND MULTI-FILE ASSEMBLIES

* Single-file assemblies contain all the necessary CIL, metadata and manifest in a single well- defined package.
* On the other hand, multi-file assemblies are composed of numerous .NET binaries, each of which is termed a *module.*
* When building a multifile assembly, one of these modules (termed the primary module) must contain the assembly manifest (and possibly CIL instructions and metadata for various types).
* The other related modules contain a module level manifest, CIL, and type metadata.
* The primary module maintains the set of required secondary modules within the assembly manifest.
* When building a multiple assembly, one of the module (termed as primary module) must contain the assembly-manifest. The other related modules contain a module-level manifest, CIL and type metadata.
* Multifile assemblies are used when different modules of the application are written in different languages.

### ROLE OF CIL

CIL is a language that sits above any particular platform-specific instruction set.

Regardless of which .NET-aware language you choose (like C#, VB.NET, VC++.NET etc), the associated compiler produces CIL instructions.

Once the C# complier (csc.exe) compiles the source code file, you end up with a single file

\*.exe assembly that contains a manifest, CIL instructions and metadata describing each aspect of the program.

***Benefits of CIL***

*Language Integration*: Each .NET-aware language produces the same underlying-CIL. Therefore, all .NET-aware languages are able to interact within a well-defined binary arena.

Since CIL is platform-agnostic, the .NET runtime is poised to become a *platform-independent architecture*. Thus, .NET has the potential to allow you to develop an application in any language and have it run on any platform.

### .NET Namespaces

A Namespace in Microsoft .Net is like containers of objects. They may contain unions, classes, structures, interfaces, enumerators and delegates. Main goal of using namespace in .Net is for creating a hierarchical organization of program. In this case a developer does not need to worry about the naming conflicts of classes, functions, variables etc., inside a project. In Microsoft

.Net, every program is created with a default namespace.

This default namespace is called as global namespace. But the program itself can declare any number of namespaces, each of them with a unique name. The advantage is that every namespace can contain any number of classes, functions, variables and also namespaces etc., whose names are unique only inside the namespace. The members with the same name can be created in some other namespace without any compiler complaints from Microsoft .Net.

To declare namespace C# .Net has a reserved keyword namespace. If a new project is created in Visual Studio .NET it automatically adds some global namespaces. These namespaces can be different in different projects. But each of them should be placed under the base namespace

System. The names space must be added and used through the using operator, if used in a different project. Please now have a look at the example of declaring some namespace:

Example:

UsingSystem;

namespace OutNamespace

{

Namespace WorkNamespace

{

/// can be placed some classes, structures etc.

}

}

### METADATA

Metadata describes each and every type (class, structure, enumeration) defined in the binary, as well as the members of each type (properties, methods, events)

It describes each externally referenced assembly that is required by the executing assembly to operate correctly.

It is used

o by numerous aspects of the .NET runtime environment

o by various development tools

o by various object browsing utilities, debugging tools and even the C# compiler itself

It is the backbone of numerous .NET technologies such as .NET Remoting, reflection services and object serialization.

Consider the following example The C# Calculator public class Calc

{

public int Add(int x,int y)

{ return x+y;}

}

Within the resulting "MetaInfo" window, you will find a description of the Add() method looking something like the following:

Method #2

MethodName: Add (06000002)

RVA: 000002064

ImplFlags: [IL] [Managed] (00000000) has This

ReturnType: I4 2 Arguments Argument #1: I4 Argument #2:

I4 2 Parameters

ParamToken: (08000001) Name: x flags: [none] (00000000) ParamToken: (08000002) Name: y flags: [none] (00000000)

In above metadata, you can see that Add() method, return type and method arguments have been fully described by the C# compiler.

### THE ROLE OF MANIFEST

Assemblies themselves are also described using metadata, which is termed as *manifest.*

The manifest contains information such as

→name of assembly/ module

→current version of the assembly

→list of files in assembly

→copyright information

→list of all externally referenced assemblies that are required for proper execution

### COMPILING CIL TO PLATFORM-SPECIFIC INSTRUCTIONS

Assemblies contain CIL instructions and metadata, rather than platform-specific instructions. CIL must be compiled on-the-fly before use. Jitter(Just-in-time compiler) is used to compile the CIL into meaningful CPU instructions.

The .NET runtime environment forces a JIT compiler for each CPU targeting the CLR, each of which is optimized for the platform it is targeting.

Developers can write a single body of code that can be efficiently JIT-compiled and executed on machines with different architectures. For example,

If you are building a .NET application that is to be deployed to a handheld device (such as a Pocket PC), the corresponding Jitter is well equipped to run within a low-memory environment.

On the other hand, if you are deploying your assembly to a back-end server (where memory is seldom an issue), the Jitter will be optimized to function in a high-memory environment

Jitter will cache the results in memory in a manner suited to the target OS. For example, if a call is made to a method named PrintDocument(), the CIL instructions are compiled into platform- specific instructions on the first invocation and retained in memory for later use. Therefore, the next time PrintDocument() is called, there is no need to recompile the CIL.

### UNDERSTANDING THE CTS

A given assembly may contain any number of distinct “types”.

In the world of .NET, “type” is simply a generic term used to refer to a member from the set

{class, structure, interface, enumeration, delegate}.

CTS fully describes all possible data-types & programming constructs supported by the runtime specifies how these entities can interact with each other & details of how they are represented in the metadata format

When you wish to build assemblies that can be used by all possible .NET-aware languages, you need to conform your exposed types to the rules of the CLS.

### CTS Class Types

* A class may be composed of any number of members (methods, constructor) and data points (fields).
* CTS allow a given class to support virtual and abstract members that define a polymorphic interface for derived class.
* Classes may only derive from a single base class (multiple inheritances are not allowed for class).

In C#, classes are declared using the class keyword. For example,

A C# class type public class Calc

{

public int Add(int x, int y)

{

return x + y;

}

}

|  |  |
| --- | --- |
| ***Class Characteristic*** | ***Meaning*** |
| Is the class “sealed” or not? | Sealed classes cannot function as a base class to other classes. |
| Does the class implement any *interfaces?* | An interface is a collection of abstract members that provide a  contract between the object and object-user. The CTS allows a class to implement any number of interfaces. |
| Is the class abstract or concrete? | *Abstract* classes cannot be directly created, but are intended to  define common behaviors for derived types. *Concrete* classes can be created directly. |
| What is the “visibility” of this class? | Each class must be configured with a visibility attribute. Basically, this feature defines if the class may be used by external assemblies, or only from within the defining assembly (e.g., aprivate helper class). |

### CTS Structure Types

A structure can be thought of as a lightweight type having value-semantics.

Structure may define any number of parameterized constructors.

All 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.

The CTS permits structures to implement any number of interfaces; but, structures may not become a base type to any other classes or structures. Therefore structures are explicitly sealed.

In C#, structure is declared using the struct keyword. For example,

A C# structure type struct Point

{

*Structures can contain fields.*

public int xPos, yPos;

*Structures can contain parameterized constructors.*

public Point(int x, int y)

{

xPos = x; yPos = y;

}

*Structures may define methods.*

public void Display()

{

Console.WriteLine("({0}, {1})", xPos, yPos);

}

}

### CTS Interface Type

Interface is a named collection of abstract member definitions, which may be supported by a given class or structure.

Interfaces do not derive from a common base type (not even System.Object)

When a class/structure implements a given interface, you are able to request access to the supplied functionality using an interface-reference in a polymorphic manner.

When you create custom interface using a .NET-aware language, the CTS allows a given interface to derive from multiple base interfaces.

In C#, interface types are defined using the interface keyword, for example:

A C# interface type.

public interface IDraw

{

void Draw();

}

### CTS Enumeration Types

Enumeration is used to group name/value pair under a specific name.

By default, the storage used to hold each item is a System.Int32 (32-bit integer) Enumerated types are derived from a common base class, *System.Enum*

This base class defines a number of interesting members that allow you to extract, manipulate, and transform the underlying name/value pairs programmatically.

Consider an example of creating a video-game application that allows the player to select one of three character categories (Wizard, Fighter, or Thief). Rather than keeping track of raw numerical values to represent each possibility, you could build a custom enumeration as:

*A C# enumeration*

public enum playertype

{

wizard=10, fighter=20, thief=30

};

### CTS Delegate Types

* Delegates are the .NET equivalent of a type-safe C-style function pointer.
* The key difference is that a .NET delegate is a class that derives from System.MulticastDelegate, rather than a simple pointer to a raw memory address.
* Delegates are useful when you wish to provide a way for one entity to forward a call to another entity.
* Delegates provide intrinsic support for multicasting, i.e. forwarding a request to multiple recipients.
* They also provide asynchronous method invocations.
* They provide the foundation for the .NET event architecture.
* In C#, delegates are declared using the delegate keyword as shown in the following example:

This C# delegate type can 'point to' any method returning an integer and taking two integers as input

public delegate int BinaryOp(int x, int y);

### INTRINSIC CTS DATA TYPES

|  |  |  |  |
| --- | --- | --- | --- |
| **.NET Base Type (CTS Data Type)** | **VB.NET Keyword** | **C# Keyword** | **Managed Extensions**  **for C++ Keyword** |
| System.Byte | Byte | byte | unsigned char |
| System.SByte | SByte | sbyte | signed char |
| System.Int16 | Short | short | Short |
| System.Int32 | Integer | int | int or long |
| System.Int64 | Long | long | int64 |
| System.UInt16 | UShort | ushort | unsigned short |
| System.Single | Single | float | Float |
| System.Double | Double | double | Double |
| System.Object | Object | object | Object^ |
| System.Char | Char | char | wchar\_t |
| System.String | String | string | String^ |
| System.Decimal | Decimal | decimal | Decimal |
| System.Boolean | Boolean | bool | Bool |

**UNDERSTANDING THE CLS**

* The Common Language Specification (CLS) is a set of rules that describe the small and complete set of features.
* These features are supported by a .NET-aware compiler to produce a code that can be hosted by CLR.
* Also, this code can be accessed by all languages in the .NET platform.
* The CLS can be viewed as physical subset of the full functionality defined by the CTS.
* The CLS is a set of rules that compiler builders must conform to, if they intend their products to function seamlessly within the .NET-universe.
* Each rule describes how this rule affects those who build the compilers as well as those who interact with them. For example, the CLS Rule 1 says:

*Rule 1: CLS rules apply only to those parts of a type that are exposed outside the defining assembly.*

Given this rule, we can understand that the remaining rules of the CLS do not apply to the logic used to build the inner workings of a .NET type. The only aspects of a type that must match to the CLS are the member definitions themselves (i.e., naming conventions, parameters, and return types). The implementation logic for a member may use any number of non-CLS techniques, as the outside world won‟t know the difference.

To illustrate, the following Add() method is not CLS-compliant, as the parameters and return values make use of unsigned data (which is not a requirement of the CLS):

public class Calc

{

*Exposed unsigned data is not CLS compliant!* public ulong Add(ulong x, ulong y)

{

return x + y;

}

}

We can make use of unsigned data internally as follows:

public class Calc

{

public int Add(int x, int y)

{

*As this ulong variable is only used internally, we are still CLS compliant.*

ulong temp; temp= x+y; return temp;

} }

Now, we have a match to the rules of the CLS, and can assured that all .NET languages are able to invoke the Add() method.

### Ensuring CLS compliance

C# does define a number of programming constructs that are not CLS-compliant. But, we can instruct the C# compiler to check the code for CLS compliance using a single .NET attribute:

Tell the C# compiler to check for CLS compliance.

*[assembly: System.CLSCompliant(true)]*

This statement must be placed outside the scope of any namespace. The [CLSCompliant] attribute will instruct the C# compiler to check each and every line of code against the rules of the CLS. If any CLS violations are discovered, we will receive a compiler error and a description of the offending code.

### UNDERSTANDING THE CLR

* The runtime can be understood as a collection of external services that are required to execute a given compiled unit-of-code. (Figure: 1-3)
* •.NET runtime provides a single well-defined runtime layer that is shared by all .NET aware languages.
* The heart of CLR is physically represented by an assembly named *mscoree.dll* (Common Object Runtime Execution Engine)
* When an assembly is referenced for use, mscoree.dll is loaded automatically, which in turn loads the required assembly into memory.
* Runtime-engine lays out the type in memory compiles the associated CIL into platform- specific instruction performs any security checks and then executes the code
* In addition, runtime-engine is in charge of resolving location of an assembly and finding requested type within binary by reading contained-metadata
* In addition, runtime-engine will also interact with the types contained within the base class libraries.
* *mscorlib.dll* assembly contains a large number of core types that encapsulate a wide variety of common programming tasks as well as the core data types used by all .NET languages.

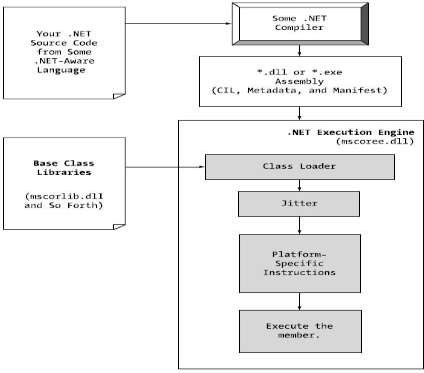


Figure 1-3: mscoree.dll in action

# Iteration statements:

### While

* It is a simple loop. It executes till the code block while the condition is true.
* If the condition is going to be false, the program will exit from the loop. Example:

public class Program { public static void Main() { int num = 1;

while (num <= 10) {

Console.WriteLine("The number is {0}", num); num++;

}

Console.ReadLine();

}

}

### Do-While Statement

* The do while loop is same as while loop except that it will be running at least one time if condition is matched or not.
* It is because it does not check the condition first time. So, it is guaranteed to execute the code of program at least one iteration.

Example:

public class Program { public static void Main() { int num = 1;

do {

Console.WriteLine("The number is {0}", num); num++;

} while (num <= 0); Console.ReadLine();

} }

### For Statement

* The For loop is used if you know the start point and end point.
* You can run a statement or a block of statements repeatedly until a specified expression evaluates to false.
* It is useful where you know in advance how many times program should iterate. public class Program {

public static void Main() { for (int i = 1; i <= 5; i++) {

Console.WriteLine("The iteration number is " + i);

}

Console.ReadLine();

} }

### For each Statement

* "The foreach statement repeats a group of embedded statements for each element in an array or an object collection.
* It means if you are working on the array or collection of object than you should use foreach statement to iterate.
* So, foreach statement iterates the every element of collection and forwards it to next statement.“
* Example: public class Program {

public static void Main() {

string[] daysOfWeek = new string[] {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday”};

foreach(string Day in daysOfWeek) {

Console.WriteLine("The Day is : {0}", Day); } Console.ReadLine();

} }

# Control Statements:

### If-then-else

The if statement has three forms: single selection, if-then-else selection, and multicase selection.

Example 1: //single selection if (i > 0)

Console.WriteLine("The number {0} is positive", i);

Example 2: //if-then-else selection if (i > 0)

Console.WriteLine("The number {0} is positive", i); else

Console.WriteLine("The number {0} is not positive", i);

Example 3: //multicase selection if (i == 0)

Console.WriteLine("The number is zero"); else if (i > 0)

Console.WriteLine("The number {0} is positive", i); else

Console.WriteLine("The number {0} is negative", i);

**Switch**

Case by case iteration will take place. Example:

string day = "Monday"; Console.WriteLine("enter the day :"); day = Console.ReadLine();

switch (day)

{

case "Mon":

break;

case "Monday":

Console.WriteLine("day is Monday: go to work"); break;

default: Console.WriteLine("default"); break;

}

case "reason1":

goto case "reason2"; // this is a jump to mimic fall-through case "reason2":

intOption = 2; break;

case "reason 3":

intOption = 3; break;

case "reason 4":

intOption = 4; break;

case "reason 5":

intOption = 5; break; default: intOption = 9; break;

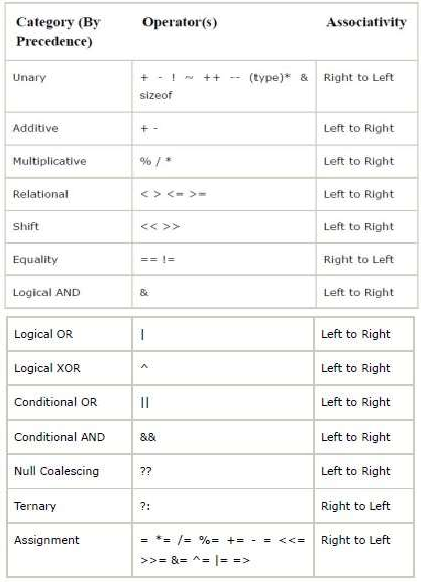
}

# Operators:

* An operator is simply a symbol that is used to perform operations.
* There can be many types of operations like arithmetic, logical, bitwise etc.
* There are following types of operators to perform different types of operations in C# language.
* Arithmetic Operators +, -, \*, /, %
* Relational Operators <, <=, >, >=, ==, !=
* Logical Operators &&, ||, !
* Bitwise Operators &, |, <<, >>, ~, ^
* Assignment Operators =, +=, -=, \*=, /=, %=
* Unary Operators ++, --
* Ternary Operators ?

## Precedence of Operators in C#

The precedence of operator specifies that which operator will be evaluated first and next. The associativity specifies the operators direction to be evaluated, it may be left to right or right to left.



### Data Types

In C#, data types are categorized based on how they store their value in the memory. C# includes following categories of data types:

1. Value type
2. Reference type
3. Pointer type

Here, we will learn about value types and reference types.

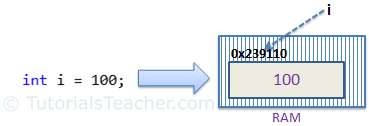
Value Type:

A data type is a value type if it holds a data value within its own memory space. It means variables of these data types directly contain their values.

For example, consider integer variable

int i = 100;

The system stores 100 in the memory space allocated for the variable 'i'. The following image illustrates how 100 is stored at some hypothetical location in the memory (0x239110) for 'i':



Memory allocation for Value Type

The following data types are all of value type:

* Bool, byte, char, decimal, double, enum, float, int, long, sbyte, short, struct, uint, ulong, ushort

Passing by Value:

When you pass a value type variable from one method to another method, the system creates a separate copy of a variable in another method, so that if value got changed in the one method won't affect on the variable in another method.

Example: Value Type

static void ChangeValue(int x)

{

x = 200;

Console.WriteLine(x);

}

static void Main(string[] args)

{

int i = 100; Console.WriteLine(i); ChangeValue(i); Console.WriteLine(i);

}

Output:

100

200

100

In the above example, variable i in Main() method remains unchanged even after we pass it to the ChangeValue() method and change it's value there.

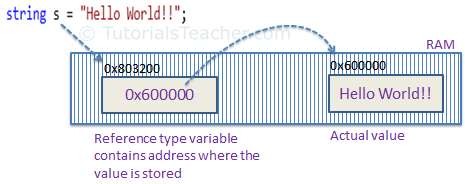
Reference Type

Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the data.

For example, consider following string variable:

string s = "Hello World!!";

The following image shows how the system allocates the memory for the above string variable.



Memory allocation for Reference type

As you can see in the above image, the system selects a random location in

0x600000

s

(0x803200)

memory

for the variable 's'. The value of a variable is

which is the

memory address of the actual data value. Thus, reference type stores the address of the location where the actual value is stored instead of value itself.

The following data types are of reference type:

* + String
  + All arrays, even if their elements are value types
  + Class
  + Delegates

Pass by Reference

When you pass a reference type variable from one method to another, it doesn't create a new copy; instead, it passes the address of the variable. If we now change the value of the variable in a method, it will also be reflected in the calling method.

Example: Reference Type Variable

static void ChangeReferenceType(Student std2)

{

std2.StudentName = "Steve";

}

static void Main(string[] args)

{

Student std1 = new Student(); std1.StudentName = "Bill"; ChangeReferenceType(std1);

Console.WriteLine(std1.StudentName);

}

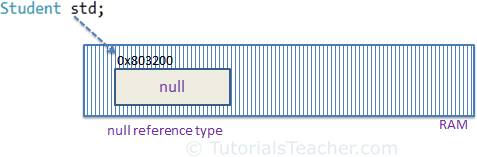
Output:

Steve

In the above example, since Student is an object, when we send the Student object std1 to the ChangeReferenceType() method, what is actually sent is the memory address of std1. Thus, when the ChangeReferenceType() method changes StudentName, it is actually changing StudentName of std1, because std1 and std2 are both pointing to the same address in memory. Therefore, the output is Steve.

Null

Reference types have null value by default, when they are not initialized. For example, a string variable (or any other variable of reference type datatype) without a value assigned to it. In this case, it has a null value, meaning it doesn't point to any other memory location, because it has no value yet.



Null Reference type

A value type variable cannot be null because it holds a value not a memory address. However, value type variables must be assigned some value before use. The compiler will give an error if you try to use a local value type variable without assigning a value to it.

Example: Compile Time Error

void someFunction()

{

int i;

Console.WriteLine(i);

}

However, value type field in a class can be declared without initialization (field not a local

variable in the function) . It will have a default value if not assigned any value, e.g., int will have 0, boolean will have false and so on.

Example: Value Type Field class myClass

{

public int i;

}

myClass mcls = new myClass(); Console.WriteLine(mcls.i);

Output:

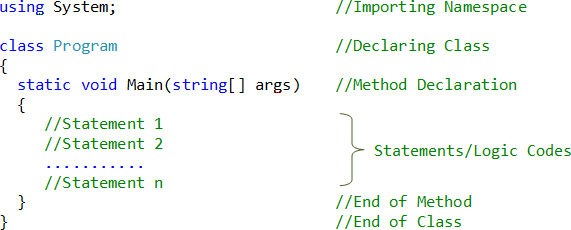
0

Points to Remember :

* + Value type stores the value in its memory space, whereas reference type stores the address of the value where it is stored.
  + Primitive data types and struct are of the 'Value' type. Class objects, string, array, delegates are reference types.
  + Value type passes byval by default. Reference type passes byref by default.
  + Value types and reference types stored in Stack and Heap in the memory depends on the scope of the variable.

### The Anatomy of a Simple C# Program

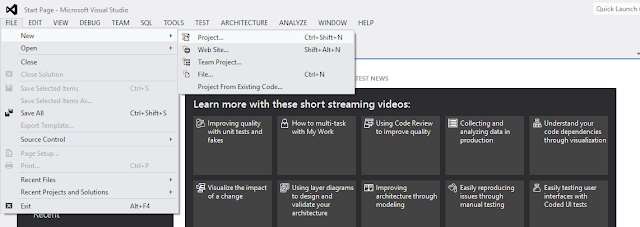
Today we will discuss about the basic structure of a c# program. What are the rules & regulations we have to follow while writing a program in c#. To get an scratch idea see the below figure



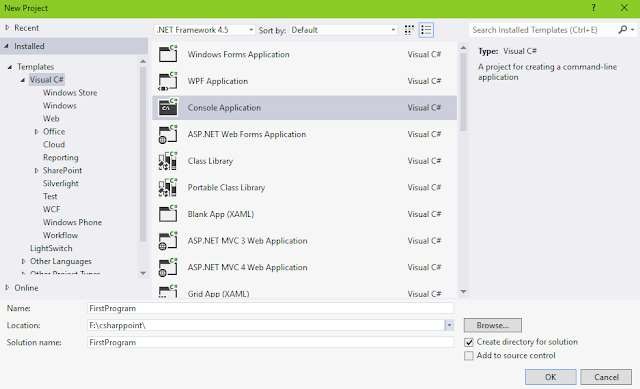
By seeing this figure we can say we have to follow these following 6 steps to write a program in c#

1. Importing Namespace
2. Declaring Class
3. Declaring Method
4. Writing Logic Codes
5. End of Method
6. End of Class

Before we start discussion on those six points, first we will write a program. To write a program you need Visual Studio. Open Visual Studio, Click on file under that click on new project like



After clicking new project another window will come in that window select Visual C# from the left side installed templates. Then Select Console Application. Give your project name and select the location where you want to store your project and click ok as follows



Then just update the program with the following code.

### using System;

**class FirstProgram**

{

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

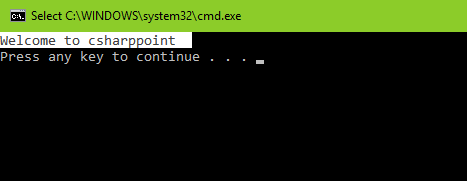
{

Console.WriteLine("Welcome to csharppoint");

}

}

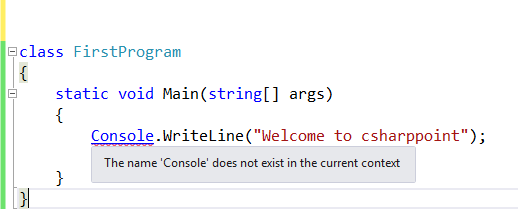
After updating with this code press ctrl + F5 to run the program. After that you will get the output like



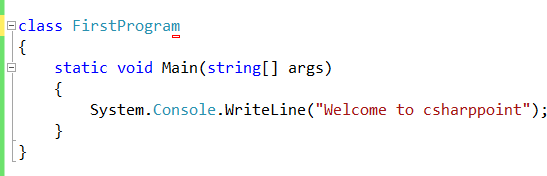
Now the program is running fine. We got the correct output. Now we will discuss what a namespace is and what is the need of that namespace in our program?

*Namespace is collection of classes, Interface, Delegates, Structs & Enums. It is used to organize our codes.*

Now you can ask me how a namespace can organize our code. Yes it is used to organize our codes. To understand how see the below figure



In the above figure we removed that "using System" namespace. After removing that we got a red mark underneath of Console saying that *"The name Console does not exist in the current instance"*. Now if we run the program then we definitely get an error because Console is a class which comes under "*System"* namespace and we are not using that in our current code. So what else can we do? We can use fully qualified path like

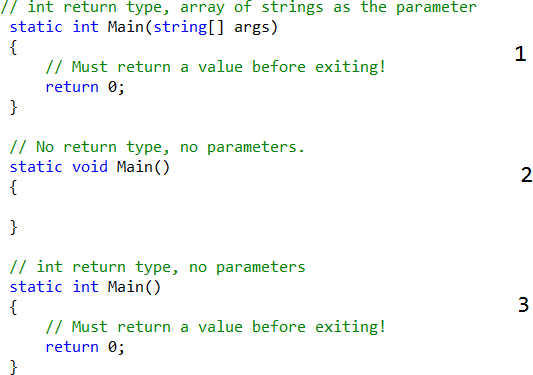


You can notice that, we are using "*System"* before *"Console"* . Now if you run the program you will end up with the correct output means the program will work fine. But think when you have 500 or 1000 of lines code in a program and you have to use fully qualified path for every line!! It will look nasty means the readability of program will be less. So instead of using fully qualified path it is recommended that you use *"using System"* at the top of your code only once. It will improve readability of your program.

Now we can move on to the second point i.e. *Declaring Class.* We will discuss about *"Class"* in a great detail in upcoming days. For the time being you just remember that it is a syntax, which is used to write a program. Now within class there is a Main() which is the entry point of our program. From there only execution of a program starts. In our case we will show an output to the console window, so our logic code is placed within Main().

Note that the signature of Main() is with the ***static*** keyword, which will be examined in detail in upcomig days. For the time being, simply understand that static members are scoped to the class level (rather than the object level) and can thus be invoked without a new class instance. In addition to the static keyword, this Main() method has a single parameter, which happens to be an array of strings (string[] args). Although you are not currently bothering to process this array, this parameter may contain any number of incoming command-line arguments. We will discuss this after, we completing arrays. Finally, this Main() method has been set up with a ***void*** return value, meaning we do not explicitly define a return value using the return keyword before exiting the method scope.

By default, Visual Studio will generate a Main() method that has a void return value and an array of string types as the single input parameter. This is not the only possible form of Main(), we can construct our application’s entry point using any of the following signatures



The most important thing is to remember that, *C# is a case-sensitive programming language*. Therefore, Main is not the same as main, and Readline is not the same as ReadLine. Be aware that all C# keywords are lowercase (e.g., public, lock, class, dynamic), while namespaces, types, and member names begin with an initial capital letter and have capitalized the first letter of any embedded words (e.g., Console.WriteLine, System.Windows.MessageBox, System.Data.SqlClient).