## Common Language Runtime Components

The Common Language Runtime (CLR) is the core part of the Microsoft.NET Framework. It is the execution environment for.NET programs. Imagine a virtual machine between your compiled code and the operating system, offering a layer of abstraction and providing essential services.

These are the elements that are actively involved while your .NET application is running:

1. **JIT Compiler:** As mentioned above, it's crucial for converting IL code to native code at runtime.
2. **Garbage Collector:** Continuously works in the background to manage memory.
3. **Class Loader:** Ensures that the required classes are available when needed.
4. **Exception Handler:** Handles any errors or unexpected events that might arise.

### 1. JIT Compiler

The **JIT compiler** is a key part of the **CLR**. It converts Intermediate Language (IL) code into machine code while your program runs. This IL comes from languages like C# or VB.NET. You can think of the JIT as a bridge that helps various systems communicate.

So, why do we call it Just-In-Time?

It’s all about timing. The JIT compiles code as the program runs. Regular compilers do this before the program starts.

Here’s how it works:

First, it only compiles what’s needed. The JIT doesn’t process all the IL at once. It compiles each method the first time it’s called.

Next, it saves time by caching. Once a method is compiled into native code, the JIT keeps that version. If you call the method again, it uses the saved version. This makes everything quicker.

**JIT (Just In Time)** Compiler makes necessary changes at the time of compilation. These are:
**Inlining**: Swapping a function call with its code to speed it up.
**Register Allocati**on: Storing variables in CPU registers for faster access.
**Code Reordering**: Changing the order of commands for better speed.
**Profiling**: Some JITs watch how the program runs to improve methods that are used a lot.

What types of JIT are there?

There’s Pre-JIT Compilation (NGen). This option compiles assemblies into native code before running, helping your app start faster. However, it can make the install size bigger and add some complexity.

Then there’s Eco-JIT from .NET 6. The Objective is to cut down the use of memory and removing unnecessary code.

What are the perks of JIT?

It lets your code run on any platform with a CLR. The JIT makes sure to create the right native code for each one.
JIT can optimize while running, making it faster than just interpreted code. Compiling and saving the code helps speed things up.
Finally, the JIT can adapt to how the program runs in real time.

## 2. Garbage Collector

It's this behind-the-scenes process that takes care of memory allocation and deallocation automatically. What it mainly does is free up memory that's taken up by objects that the application doesn't need anymore. This is a huge relief for developers because they don't have to deal with the boring and often tricky job of managing memory themselves. Plus, it helps to avoid memory leaks and dangling pointers, which is always a good thing for keeping apps stable and reliable.

**How Does the Garbage Collector Work?**

1. **Memory Allocation**: Whenever you create a new object in .NET, the Common Language Runtime (CLR) allocates some memory for that object from what's called the managed heap. This heap is specifically overseen by the Garbage Collector.

2. **Root Objects**: The Garbage Collector kicks things off by figuring out which objects are "root objects." These are objects that the application can directly access, like static variables, objects that are currently on the stack, and those held by the CPU registers. We consider these roots to be "live".

3. **Marking**: After that, the GC goes through the object graph, starting from those root objects. It marks all the objects that can be reached from the roots as "live." Any object that isn't reachable from the roots? Yeah, that's considered "garbage."

4. **Collection**: Once marking is done, the GC collects those garbage objects, which means it reclaims the memory that was being used by them.

5. **Compaction (Optional)**: After collecting the garbage, the GC might decide to compact the managed heap. Compaction is about moving the live objects closer together, reducing fragmentation, and making it simpler to allocate memory for new objects later on. Not every GC does compaction, and even the ones that do might not compact with every collection.

6. **Generations**: The GC uses a generational approach to make its work more efficient. The managed heap is split into generations—typically three: 0, 1, and 2. New objects go into generation 0. If those objects survive a garbage collection cycle, they get moved to the next generation. The GC tends to collect generation 0 more frequently because newer objects are likely to become garbage faster. This approach is based on the idea that younger objects don't stick around as long.

**Key Concepts Related to Garbage Collection**:

- **Finalization**: Before an object is collected, its Finalize method gets called. This gives the object a chance to clean up, like releasing unmanaged resources. But, here's the catch: Finalization is non-deterministic, meaning you can't predict exactly when it will happen, which can add some overhead, so it's best to use it sparingly.

- **Dispose Pattern**: The Disposable interface and Dispose method give objects a way to explicitly release resources before the GC gets to them. This is super important for getting rid of unmanaged resources like file handles, network connections, and database connections. In C#, using the `using` statement makes sure that Dispose is called automatically.

**Managed Resources**

**Definition**: Managed resources are those directly managed by the CLR and its garbage collector. Typically, these are objects created within the .NET environment.

**Examples**: Think about objects created using the new keyword in C#, like instances of classes, strings, and arrays—they're all managed resources.

**Management**:
The garbage collector keeps track of the lifetime of managed resources, reclaiming their memory when they're no longer needed. So generally, you don't have to worry about releasing them explicitly.

**Unmanaged Resources**
Definition: Unmanaged resources are not directly managed by the CLR. These are usually external resources that interact with the operating system or other external components.
**Examples:**
**File handles**: When you open a file, the operating system gives you a handle, which is an unmanaged resource.
**Network connections**: Setting up a network connection involves interacting with the operating system's networking components; that connection itself is unmanaged. - **Database connections**: Similar to network connections, these are unmanaged resources too.
**Handles to operating system objects**: Windows uses handles to represent various objects like windows, processes, and threads—these are unmanaged resources.
**Unmanaged memory**: You might allocate memory directly from the operating system using techniques like P/Invoke to call native APIs; this memory is considered unmanaged.

- **Weak References**: Weak references allow you to hold onto an object without preventing it from being garbage collected. This can be handy for caching or other situations where you want to access an object if it's still around, but you don't want to keep it alive unnecessarily.

**Benefits of Garbage Collection**:
- **Increased Developer Productivity**: Developers don't have to spend time manually managing memory, allowing them to focus on other aspects of application development.
- **Improved Application Stability:** Garbage collection prevents memory leaks and dangling pointers, which can lead to crashes and other errors.
- **Reduced Risk of Memory-Related Bugs:** Automating memory management reduces the risk of memory-related bugs, making applications more reliable.

**Limitation:**
- **Performance Overhead:** Garbage collection does introduce some performance overhead, as the GC needs to pause the application periodically to perform its work. However, modern GCs are highly optimized, and the overhead is usually minimal.
- **Non-Determinism:** The exact timing of garbage collection is non-deterministic. You cannot predict precisely when the GC will run. This can sometimes make it challenging to reason about the performance of applications in certain situations.

**3. Class Loader**
The Class Loader is an integral part of the CLR. It is what loads assemblies (.NET assemblies which, in turn, contain your compiled code) into memory and makes available for runtime all the types (classes, interfaces, etc.) residing within those assemblies. Although you will not often have to call into it as part of your day-to-day coding, understanding what its purpose is important if you want to really understand how.NET applications run.

Here's a breakdown of the Class Loader's key functions and concepts:
1**. Locate Assemblies**: Class Loader locates the assemblies required by your application. It does this by scanning everywhere, like in the application directory, in the GAC, or other places defined by you.

2. **Assembly Loading** : After the assembly has been located, the Class Loader loads the assembly into memory. It does not download the library at one go but loads it on demand, only when that part of the library has been required.
3. **Dependency Resolution**: Assemblies depend often on other assemblies. The Class Loader has to resolve these dependencies. It has to, therefore, load all assemblies before the application can progress.
4. **Loading Types**: The Class Loader does not load all the types in an assembly at once. This is done on demand as it comes to be used by the application. This enhances startup performance.
5. **Type Verification**: Just before a type is loaded, the Class Loader verification checks are done to ensure that the type is valid and safe to use. This prevents security vulnerabilities and ensures type safety.
6. **Creating Type Object**: Once a type is loaded and verified, the Class Loader creates a type object that represents the type in the CLR. This type object can be used to access information about the type, such as its methods, properties, and fields.

**Key Concepts Related to Class Loading:**
**Assemblies**: An assembly is an unit of deployment for .NET code. An assembly includes compiled code in the form of IL as well as type metadata and resources which are related to other types appearing in the code.
**Types**: Types are the basic building blocks of .NET applications. The types include classes, interfaces, delegates, enums, and structs.
**Namespaces**: All types are logically grouped into names spaces.
**Global Assembly Cache (GAC)**: The GAC is a shared repository of assemblies. The assemblies present in the GAC can be shared by many applications.
**Application Domain**: An application domain is a separately isolated region inside a process within which a .NET application runs. Every application domain has its own Class Loader that ensures loaded assemblies in one application domain are not interfering with another application domain that is currently running.
**Contexts**: The class loader also operates on contexts. Contexts are used to offer finer-grained control over assembly loading.

**How it Works (Simplified)**:
1. Your application begins execution.
2. When the code requires using a type (class, interface, etc.) that hasn't been loaded yet, the Class Loader is called.
3. The Class Loader determines which assembly contains the required type.
4. If the assembly hasn't been loaded, it locates and loads the assembly.
5. The Class Loader loads the actual type.
6. The type is ready now to be used by your application.

**Importance of the Class Loader:**
• **Dynamic Loading**: The on-demand loading of types by the Class Loader boosts performance of the startup of an application
• **Isolation**: Application domains and the Class Loader achieve isolation between applications, and thus prevent them from interfering with each other
• **Version Control**: The Class Loader helps in controlling multiple versions of assemblies.
• **Security** : The type verification that the Class Loader provides to us certainly strengthens the security of.NET applications.

**4. Exception Handler**

The Exception Handler is one of the primary mechanisms allowed by CLR in dealing with errors or exceptional events which arise during application execution. Such a facility lets structured exception handling prevent subsequent crashes of the applications and the chance of graceful recovery or informative reporting of errors.

**What's an Exception?**

An exception is an event that disrupts the normal flow of program execution. It's an indication that something unexpected or erroneous has occurred. Exceptions can be caused by a variety of factors, including the following:

**Invalid input**: Attempting to parse a string that is not a valid number.
**File I/O errors**: Trying to read a file that does not exist or that the application does not have permission to access.
**Network issues**: Network connections go down unexpectedly.
**Null references**: You attempt to access a member of an object that is null.
**Arithmetic exceptions**: Division by zero.
**Out Of Memory Exception**: When the application runs out of memory.
How Does the Exception Handler Work?
The .NET exception handling mechanism relies on the try-catch-finally block.
try Block: You surround the code that might throw an exception by a try block. This indicates the code that you want to watch out for exceptions.

**Catch Blocks**: There is one or more catch blocks placed after the try block. Each catch block will explicitly define what type of exception it can catch. At the point of occurrence of any exception inside the try block, CLR searches for a catch block that can handle this particular type of exception or the catch block of its base class.

**finally Block (Optional)**: The finally block may be used immediately after the try and catch blocks. Code within the finally block is guaranteed to execute whether an exception was thrown and caught or not. It is mainly used for clean up, e.g., closing files or releasing resources.

**Example:**

**C#**

```
try
{
        // Code which may throw an exception (e.g., file I/O)
        string fileContent = File.ReadAllText("myfile.txt");
        int number = int.Parse(fileContent); // May throw FormatException
        //. use the file content .
}
catch (FileNotFoundException ex)
{
        // Handle the FileNotFoundException
        Console.WriteLine("File not found: " + ex.Message);
}
catch (FormatException ex)
{
        // Handle the FormatException
        Console.WriteLine("Invalid number format: " + ex.Message);
}
catch (Exception ex) // Catching a general exception
{
        // Handle any other exception
        Console.WriteLine("An error occurred: " + ex.Message);
}

finally
{
        // Code which always runs (e.g., closing resources)
        Console.WriteLine("Finally block executed.");
}
```

**Key Concepts**:

**Exception Objects**: When an exception is thrown, an exception object is created. The exception object will hold information about the error-for example, it can hold the type of exception that has occurred, a message describing the error, and the call stack, which indicates the method calls in the sequence leading to the exception.
**Exception Propagation**: If an exception is not caught by the current method, it propagates along the call stack to the calling method. It does so until it finds a catch block that can handle the exception or until the exception reaches the top of the call stack, in which case the application may crash.
Custom Exceptions: you could derive exception classes if you wanted special types of exception in your application.

**Benefits of Exception Handling are**
Application's robustness Improves with Error not causing applications to crash at times gracefully while other errors just being printed informative message for users or the logging the error with information that needs debugging.
**Code Readability**: The use of try-catch blocks makes the code more readable and easier to understand by separating the normal code path from the error handling logic.
**Resource Management**: The finally block ensures that resources are released properly, even if exceptions occur.

**Catch specific exceptions**: Avoid catching the generic Exception class unless absolutely necessary. Catch specific exception types whenever possible to handle errors more appropriately.
**Informative Error Messages**: You should display the error messages to your user in ways that are useful and easy to understand but without exposing the sensitive information.
**Log exceptions**: Log exception for debugging and analysis. This can help you track errors within your application
**Use finally for cleanup**: Make sure your resources get released whether an exception has occurred or not by using finally block.
So, An Exception Handler within the CLR is useful, powerful management tool for error in NET. It leads to a structured path toward exception handling. This mechanism protects resources in applications to provide robust environment.

## Components of the CLR

The main components of the CLR include:

1. **Class Loader** – Loads classes and assemblies into memory for execution.
2. **Just-In-Time (JIT) Compiler** – Converts Microsoft Intermediate Language (MSIL) into native machine code for execution.
3. **Garbage Collector (GC)** – Manages memory allocation and automatic deallocation to prevent memory leaks.
4. **Code Manager** – Handles code execution and enforces code access security.
5. **Security Engine** – Implements security policies, including code access security and role-based security.
6. **Thread Support** – Manages threading and multi-threading execution.
7. **Exception Handling** – Provides a structured way to handle runtime errors and exceptions.
8. **Interoperability Services** – Facilitates communication between managed and unmanaged code (e.g., COM and P/Invoke).
9. **Metadata and Reflection** – Stores metadata about classes, methods, and assemblies, allowing runtime inspection and dynamic invocation.
10. **Debugging and Profiling Support** – Provides tools for debugging and performance monitoring of .NET applications.

# Microsoft Intermediate Language (MSIL)

Microsoft Intermediate Language (MSIL), also called Intermediate Language (IL) or Common Intermediate Language (CIL), is a low-level, CPU-independent instruction set used in the .NET framework. It is generated when source code written in C#, VB.NET, F#, or other .NET languages is compiled.
MSIL is not executed directly by the CPU. Instead, the Common Language Runtime (CLR) translates it into native machine code using the Just-In-Time (JIT) Compiler at runtime.

**How MSIL Works in.NET Execution?**
Step-by-Step Execution Process:
**1. Source Code Compilation:**
The compiler, Roslyn for C# and VB.NET, converts high-level code to MSIL + Metadata.
The MSIL is then stored inside a portable executable (PE) file (.exe or.dll) along with metadata.
**2. Assembly Loading:**
When a program is executed, the CLR loads the assembly (EXE or DLL).
3. **JIT Compilation:**
The Just-In-Time (JIT) Compiler translates MSIL into native machine code for execution on the specific OS and hardware.
**4. Execution:**
The native code is executed by the CPU.

**2. Features of MSIL**
• Platform-Independent:
Can run on Windows, Linux, macOS, or any OS with .NET installed.
• Language-Agnostic:
Supports multiple .NET languages (C#, VB.NET, F#, etc.).
• Type-Safe and Secure:
Enforces strict type checking to prevent memory corruption.
• Object-Oriented:
Supports OOP features like classes, inheritance, polymorphism, and interfaces.
• Metadata Support:
Maintains type information, method signatures, security permissions, and attributes in the assembly.
• Exception Handling:
Gives a consistent error-handling model across all .NET languages.
• Managed Execution:
GC manages memory allocation and deallocation.

**3. Example of MSIL Code**
Let's take a simple C# program:

```
// csharp
using System;
class Program
{
        static void Main()
        {
                Console.WriteLine(\"Hello, World!\");
        }
}
```

After compilation, the MSIL code look like

```
.method private hidebysig static void Main() cil managed
// assembly
{
        .entrypoint
        ldstr "Hello, World!"   // Load string onto the evaluation stack
        call void [System.Console]::WriteLine(string)
        ret
}
```

MSIL Instructions Breakdown:
entrypoint → This marks the beginning of the program.
ldstr "Hello, World!" → This loads the string onto the evaluation stack.
call → This is the Console.WriteLine method call.
ret → This returns from the method.

**4. MSIL Instructions and OpCodes**
MSIL contains a set of low-level instructions similar to assembly language. These are called OpCodes (operation codes).

| MSIL Instruction | Description |
|---|---|
| ldstr "text" | Loads a string onto the stack |
| call | Calls a method |
| ret | Returns from a method |
| ldloc | Loads a local variable onto the stack |
| stloc | Stores a value into a local variable |
| ldarg | Loads an argument passed to a function |
| add | Adds two numbers on the stack |
| mul | Multiplies two numbers on the stack |
| div | Divides two numbers on the stack |
| brtrue | Branches if a condition is true |

Example: Adding two numbers in MSIL

```
//assembly
ldc.i4 5      // Load integer 5 onto the stack
ldc.i4 10     // Load integer 10 onto the stack
add          // Add top two values in the stack
stloc.0      // Store result in local variable 0
```

**5. Types of JIT Compilers for MSIL Execution**
The CLR uses different JIT compilation techniques to convert MSIL into native code:

| JIT Type | Description |
|---|---|
| Normal JIT | Converts MSIL to native code on-demand (method-by-method). |
| Pre-JIT (AOT) | Converts entire MSIL code to native code before execution (used in .NET Native, ReadyToRun). |
| Econo JIT | Optimizes memory by discarding native code after execution. |

**6. MSIL and Language Interoperability**
Since all .NET languages compile to MSIL, a C# assembly can be used in VB.NET, and vice versa.
Example:

A C# DLL can be referenced and used in a VB.NET project without modification.

**7. MSIL and Reverse Engineering**

As MSIL is higher-level than machine code, it can be decompiled using tools like:

- ILSpy (Free)
- dotPeek (JetBrains)
- dnSpy (Advanced debugging)

To avoid reverse engineering, developers use obfuscation tools like Dotfuscator to scramble MSIL.

**8. Advanced MSIL Concepts**

**Reflection**: Allows reading MSIL metadata at runtime (e.g., getting method names dynamically).
**Dynamic Code Generation**: Tools like System.Reflection.Emit enable creating MSIL dynamically at runtime.
**Unsafe Code & P/Invoke**: Can call unmanaged C++ DLLs from MSIL using DllImport.

**9. MSIL vs. Java Bytecode**

| Feature | MSIL (CIL) | Java Bytecode |
|---|---|---|
| Platform Dependency | Platform-independent | Platform-independent |
| Execution Engine | Common Language Runtime (CLR) | Java Virtual Machine (JVM) |
| Compilation | JIT or AOT | JIT or AOT |
| Interoperability | Multi-language support (C#, VB.NET, F#) | Primarily for Java |

*****

# Unifying .Net

Microsoft's **Unifying .NET** initiative has been a important task, one which would unify the different .NET implementations under one united platform. Here's what that process means:

**Objectives of Unification**:

**Simplification**:
In the past, .NET used to have distinct implementations such as .NET Framework, .NET Core, and Xamarin/Mono, each having its own set of advantages and disadvantages. This meant fragmentation and complexity for programmers.
The consolidation initiative targets a common base class library and runtime for all.NET apps.

**Cross-Platform Features**:
One of the major motivators was to make it possible for developers to develop apps that are able to execute on different platforms (Windows, macOS, Linux, etc.) with one codebase.

**Better Developer Experience**:
Through consolidating the platform, Microsoft targets offering a better and more standardized developer experience.

**Important Development of .Net**:

**.NET 5**:
This was a significant step in the unification process to unite.NET Framework,.NET Core, and Xamarin/Mono.
It was a huge step towards a unified.NET platform.

**.NET 6 and .NET 7 (and later)**:
These later releases built on the unification effort, further polishing the platform, improving performance, and introducing new features.
.Net 7 particularly has been reported as a major step towards the original intentions of the unification process.

**.NET MAUI (.NET Multi-platform App UI)**:
A cross-platform framework for developing native mobile and desktop applications with C# and XAML, further advancing the unification aim.

**Most Important Features of the Unified.NET**:

**Single Base Class Library (BCL)**:
A single set of libraries that can be shared by all.NET applications.

**Cross-Platform Runtime**:
One runtime that can execute on multiple operating systems.

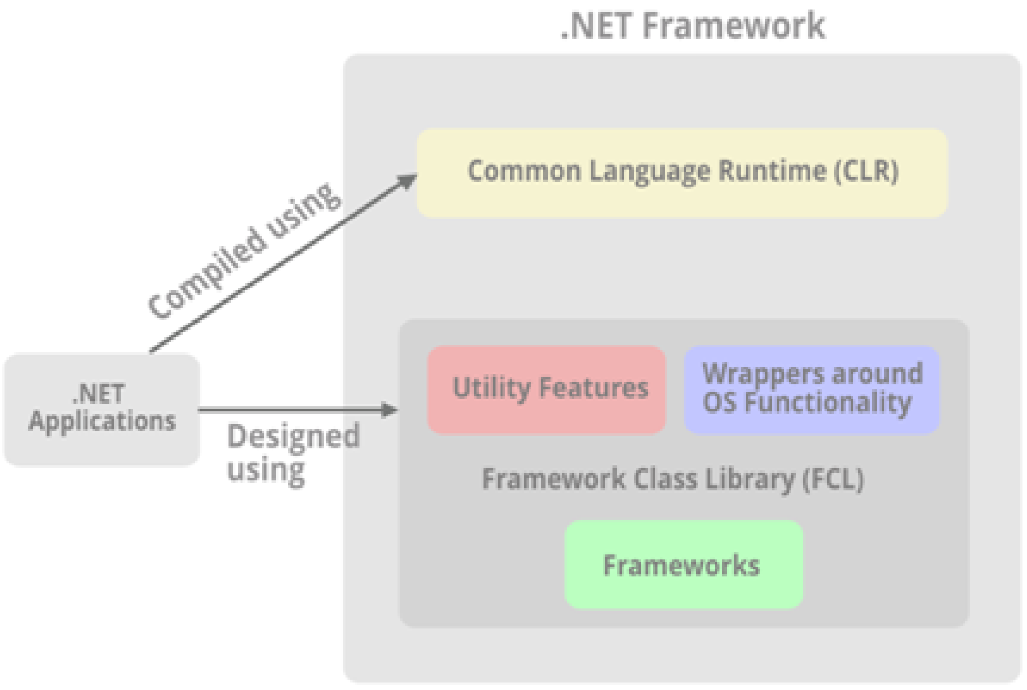
**Better Performance**:
Continued work to make the runtime and libraries more efficient.

## The Class Framework

The Framework Class Library or FCL provides the system functionality in the **.NET Framework** as it has various classes, data types, interfaces, etc. to perform multiple functions and build different types of applications such as desktop applications, web applications, mobile applications, etc. The Framework Class Library is integrated with the Common Language Runtime (CLR) of the .NET framework and is used by all the .NET languages such as Visual Basic .NET, **C#**, F#, etc.

**Categories in the Framework Class Library**

The functionality of the Framework Class Library can be broadly divided into **three** categories i.e *utility features written in .NET*, *wrappers around the OS functionality* and *frameworks*. These categories are not rigidly defined and there are many classes that may fit into more than one category.


.NET Framework
Common Language Runtime (CLR)
Compiled using
.NET Applications
Designed using
Utility Features
Wrappers around OS Functionality
Framework Class Library (FCL)
Frameworks

*Framework-Class-Library-FCL-in-Dot-NET*

Details about the Categories in the Framework Class Library are given as follows:

- **Utility Features:** The utility features in the FCL includes various collection classes such as list, stack, queue, dictionary, etc. and also classes for more varied manipulations.

- **Wrappers Around OS functionality:** Some of the features in the FCL are wrappers around the underlying Windows OS functionality. These include the classes for using the file system, the classes to handle the network features, the classes to handle I/O for console applications, etc.

- **Frameworks:** There are various frameworks available in the FCL to develop certain applications. For example, ASP.NET is used to develop web applications, Windows Presentation Foundation (WPF) is used to render user interfaces in Windows applications, and so on.

## Namespaces in the Framework Class Library

Namespaces in the Framework Class Library are a group of related classes and interfaces that can be used by all the *.NET framework* languages. Some of the namespaces in the FCL along with their description is given as follows:

| Namespace | Description |
|---|---|
| Accessibility | The Accessibility namespace is a part of the managed wrapper for the COM accessibility interface. |
| Microsoft.Activities | The Microsoft.Activities namespace provides support for Windows Workflow Foundation applications. |
| Microsoft.CSharp | The Microsoft.CSharp namespace has support for compilation and code generation for the C# source code. |
| Microsoft.JScript | The Microsoft.JScript namespace has support for compilation and code generation for the JScript source code. |
| Microsoft.VisualBasic | The Microsoft.VisualBasic namespace has support for compilation and code generation for the VisualBasic source code. |
| System | The System namespace has base classes for definition of interfaces, data types, events, event handlers, attributes, processing exceptions etc. |
| System.Activities | The System.Activities namespace handles the creation and working with activities in the Window Workflow Foundation using various classes. |
| System.Collections | The System.Collections namespace has multiple standard, specialized, and generic collection objects that are defined using various types. |
| System.Configuration | The System.Configuration namespace handles configuration data using various types. This may include data in machine or application configuration files. |
| System.Data | The System.Data namespace accesses and manages data from various sources using different classes. |
| System.Drawing | The System.Drawing namespace handles GDI+ basic graphics functionality. Various child namespaces also handle vector graphics functionality, advanced imaging functionality, etc. |
| System.Globalization | The System.Globalization namespace handles language, country, calendars used, format patterns for dates, etc. using various classes. |
| System.IO | The System.IO namespaces support IO like data read/write into streams, data compression, communicate using named pipes etc. using various types. |
| System.Linq | The System.Linq namespace supports Language-Integrated Query (LINQ) using various types. |
| System.Media | The System.Media namespace handles sound files and accessing the sounds provided by the system using various classes. |
| System.Net | The System.Net namespace provides an interface for network protocols, cache policies for web resources, composing and sending e-mail etc. using various classes. |
| System.Reflection | The System.Reflection namespace gives a managed view of loaded methods, types, fields, etc. It can also create and invoke types dynamically. |
| System.Security | The System.Security namespace has the .NET security system and permissions. Child namespaces provide authentication, cryptographic services etc. |
| System.Threading | The System.Threading namespace allows multithreaded programming using various types. |
| XamlGeneratedNamespace | The XamlGeneratedNamespace has compiler-generated types that are not used directly from the code. |

## Purpose of Namespaces

Namespaces are used to organize the classes. It helps to control the scope of methods and classes in larger .Net programming projects. In simpler words you can say that it provides a way to keep one set of names (like class names) different from other sets of names. The main advantage of using namespace is that the class names which are declared in one namespace will not clash with the same class names declared in another namespace. It is also referred as named group of classes having common features. The members of a namespace can be namespaces, interfaces, structures, and delegates.

**Defining a Namespace**
To define a namespace in C#, we will use the namespace keyword followed by the name of the namespace and curly braces containing the body of the namespace as follows:

**Syntax:**
```
namespace name_of_namespace
{
        // Namespace (Nested Namespaces)
        // Classes
        //Interfaces
        //structures
        //Delegates
}
```

**Example:**
```
// defining the namespace name1
namespace name1
{
  // C1 is the class in the namespace name1
  class C1
  {
     // class code
  }
}
```

**Accessing the Members of Namespace**
The members of a namespace are accessed by using dot(.) operator. A class in C# is fully known by its respective namespace.

Syntax:
```
[namespace_name].[member_name]
```

**Note:**
Two classes with the same name can be created inside 2 different namespaces in a single program.
Inside a namespace, no two classes can have the same name.
In C#, the full name of the class starts from its namespace name followed by dot(.) operator and the class name, which is termed as the fully qualified name of the class.

**Example:**

```
// C# program to illustrate the
// use of namespaces
// namespace declaration

namespace first
 {
   // name_1 namespace members
   // i.e. class

   class proj1
   {
      // function of class Proj1

      public static void display()
      {
         // Here System is the namespace
         // under which Console class is defined
         // You can avoid writing System with
         // the help of "using" keyword discussed
         // later in this article

         System.Console.WriteLine("Hello!!!   HEC   Students");
      }
     }

   /* Removing comment will give the error
     because no two classes can have the
     same name under a single namespace

     class Proj1
     {

     }      */
} // ending of first namespace

// Class declaration

class Proj2
{
   // Main Method

   public static void Main(String []args)
   {
      // calling the display method of
      // class Proj1 by using two dot
      // operator as one is use to access
      // the class of first namespace and
      // another is use to access the
      // static method of class Proj1.
      // Termed as fully qualified name

      first.Proj1.display();
    }
  }
```

**Output:**

```
Hello!!!  HEC  Students
```

In the above example:

In System.Console.WriteLine()" "System" is a namespace in which we have a class named "Console" whose method is "WriteLine()".

It is not necessary to keep each class in C# within Namespace but we do it to organize our code well.

Here "." is the delimiter used to separate the class name from the namespace and function name from the classname.

**The using keyword**

It is not actually practical to call the function or class(or you can say members of a namespace) every time by using its fully qualified name. In the above example, System.Console.WriteLine("Hello!!!  HEC Students"); and first.Proj1.display(); are the fully qualified name. So C# provides a keyword "using" which help the user to avoid writing fully qualified names again and again. The user just has to mention the namespace name at the starting of the program and then he can easily avoid the use of fully qualified names.

**Syntax:**

```
using [namespace_name][.][sub-namespace_name];
```

In the above syntax, dot(.) is used to include subnamespace names in the program.

**Example:**

```
// predefined namespace name
using System;

// user-defined namespace name
using name1
// namespace having subnamespace
using System.Collections.Generic;
```

**Program:**

```
// C# program to illustrate the
// use of using keyword
// predefined namespace
using System;

// user defined namespace
using first;

// namespace declaration
namespace first
{
   // name_1 namespace members
   // i.e. class
   class Proj1
   {
      // function of class Proj1
      public static void display()
      {
         // No need to write fully qualified name
         // as we have used "using System"
         Console.WriteLine("Hello!!!  HEC  Students ");
      }
   }
} // ending of first namespace
```

```
// Class declaration

class Proj2
{
    // Main Method
    public static void Main(String []args)
    {
        // calling the display method of
        // class Proj1 by using only one
        // dot operator as display is the
        // static method of class Proj1
        Proj1.display();
    }
}
```

**Output:**

```
Hello!!!  HEC  Students
```

**Nested Namespaces**

You can also define a namespace into another namespace which is termed as the nested namespace. To access the members of nested namespace user has to use the dot(.) operator.
For example, Generic is the nested namespace in the collections namespace
as System.Collections.Generic

**Syntax:**

```
namespace name_of_namespace_1
{
   // Member declarations & definitions
   namespace name_of_namespace_2
   {
        // Member declarations & definitions
        .
        .
   }
}
```

**Program:**

```
// C# program to illustrate use of
// nested namespace
using System;

// You can also use
// using Main_name.Nest_name;
// to avoid the use of fully
// qualified name

// main namespace
namespace Main_name
{

  // nested namespace
  namespace Nest_name
  {
    // class within nested namespace
    class Proj1
     {
        // Constructor of nested
        // namespace class Proj1
        public Proj1()
       {
          Console.WriteLine("Nested Namespace Constructor");

        }  //End of Proj1
     }  // End of namespace Nest_name
  }  // End of namespace Main_name

// Driver Class
class Driver
{
  // Main Method
  public static void Main(string[] args)
  {
    // accessing the Nested Namespace by
    // using fully qualified name
    // "new" is used as Proj1()
    // is the Constructor
    new Main_name.Nest_name.Proj1();
  }
 }
```

**Output:**

```
Nested Namespace Constructor
```

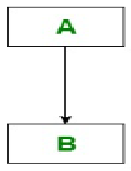
# Inheritance

Inheritance is the property that allows the reuse of an existing Class to built a New Class without changing existing one develop new thing. It enhance the enriches of Class.

- Inheritance allows you to create new classes (subclasses or derived classes) from existing classes (parent classes or base classes).
- Subclasses inherit the properties and methods of their parent class, but can also have their own unique characteristics.
- This promotes code reusability and reduces redundancy.

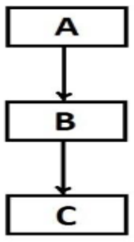
**Type of Inheritance**

- **Single Inheritance** is a type of inheritance in Object Oriented Programming (OOP) where a **child (subclass)** inherits from only **one parent (super class).** This allows the child class to reuse the attributes and methods of the parent class while also adding its own functionality.



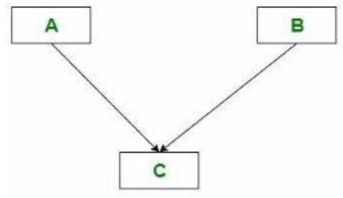
Single Inheritance

- **Multilevel Inheritance** is a type of inheritance in Object-Oriented Programming (OOP) where a class (child) inherits from another class (parent), and then another class (grandchild) inherits from the child class. This forms a **chain of inheritance** where each level builds upon the previous one.



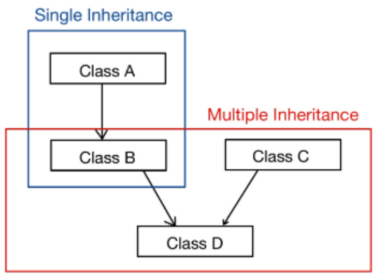
multi-level inheritance

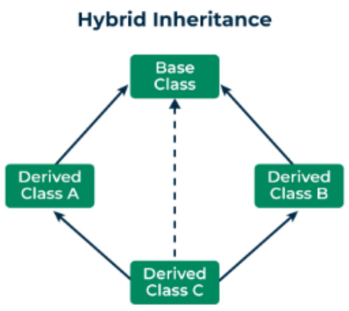
- **Multiple Inheritance** is a type of inheritance in Object-Oriented Programming (OOP) where a single child class inherits from **two or more parent classes.** This allows the child class to combine features and functionalities from multiple sources.



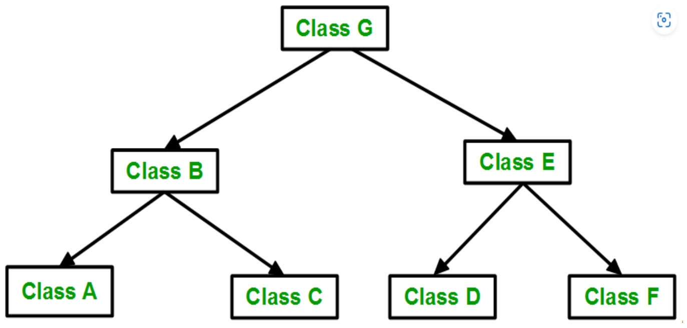
Multiple Inheritance

- **Hybrid Inheritance** is a combination of two or more types of inheritance in a single program. It is used to create a complex class hierarchy by combining **single, multiple, multilevel,** and **hierarchical inheritance.**





- **Hierarchical inheritance** is a type of inheritance in object-oriented programming (OOP) where a base class is inherited by multiple derived classes. This means that several classes (child classes) inherit properties and behaviors (methods) from a single parent class.



**Inheritance**: It's a mechanism of consuming the members of one class in another class by establishing parent/child relationship between the classes which provides re-usebility.

**How it is possible:**

```
class A
{
        -   Members
}


class B : A
{
        -   Consuming the members of A from here.
}
```

**Note**: In inheritance child class can consume members of it's parent class as if the owner of those members (expect private members of parent).

(e.g.> Children have right on his parent's properties (House, Car, Deposited Amount etc.), but children cannot take over parent's job because job is completely private to the parent.)

**A => Parent Class or Base Class or Super Class**

**B => Child Class or Derived  Class or Sub Class**

**Inheritance Program:**

Project Name is : ProjInherit

**A**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Claims;
using System.Text;
using System.Threading.Tasks;
namespace proj1
{
    class class1
    {
        public void Test1()
        {
            Console.WriteLine("Method 1");
        }
        public void Test2()
        {
            Console.WriteLine("Method 2");
        }
    }
}
```

**B**

**In solution tool Right Click on ProInherit, then Select Add, Now Select Class -> Click on Add**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace proj1
{
    class class2:class1
    {
        static void Main()
        {
            class2 c = new class2();
            c.Test1();
            c.Test2();
            Console.ReadLine();
        }
    }
}
```

**Output**:

```
Method 1
Method 2
```

**Now, New Method Test3 will be Add to Class2**

**C**

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

namespace proj1
{
    class class2:class1
    {
        public void Test3()
        {
            Console.WriteLine("Method 3");
        }
        static void Main()
        {
            class2 c = new class2();
            c.Test1();
            c.Test2();
             c.Test3();
            Console.ReadLine();
        }
    }
}
```

**Output**:

```
Method 1
Method 2
Method 3
```

Now, the question is how many methods Class2 contains
Answer is 3 methods.

How many method class1 contains
Answer is 2

**Important points to Remember**

(1) Parent class constructor must be accessible to child class otherwise inheritance will not be possible.
Every class contains a constructor implicitly define if not defined explicitly.
In this example class1 and class2 both these classes have constructor. If constructor is implicitly define that is a public constructor and that constructor is now accessible to class2.

**Now, In the class1 we defining a explicit constructor**.

**A**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Claims;
using System.Text;
using System.Threading.Tasks;
namespace proj1
{
    class class1
    {
        public class1()
        {
            Console.WriteLine("Class1 constructor is called");
        }
        public void Test1()
        {
            Console.WriteLine("Method 1");
        }
        public void Test2()
        {
            Console.WriteLine("Method 2");
        }
    }
}
```

**Run the Program**:

**Output**:
```
Class1 constructor is called
Method 1
Method 2
Method 3
```

(2) Whenever child class instance is created the child class constructor will implicitly call parent class constructor.

(Do not use private in this case (Change it `public class1() to private class1() let see want happen`)).

**Now, In the class2 we defining a explicit constructor**.

**C**

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

namespace proj1
{
    class class2:class1
    {
         public class2()
        {
            Console.WriteLine("Class2 constructor is called");
        }
        public void Test3()
        {
            Console.WriteLine("Method 3");
        }
        static void Main()
        {
            class2 c = new class2();
            c.Test1();
            c.Test2();
            c.Test3();
            Console.ReadLine();
        }
    }
}
```

**Output**:

```
Class1 constructor is called
Class2 constructor is called
Method 1
Method 2
Method 3
```

Execution always start with parent class constructor, when you create the instance of child class, child class constructor implicitly call the parent class constructor. It happens nested.
Parent class must be initialized to consume in child class. Child class is dependent on parent class, so parent class must be initialized

(2) In inheritance child class can access parent classes members but parent class can never access any member that is purely defined under the child class.

**C**

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

namespace proj1
{
    class class2:class1
    {
         public class2()
        {
            Console.WriteLine("Class2 constructor is called");
        }
        public void Test3()
        {
            Console.WriteLine("Method 3");
        }
        static void Main()
        {
            Class1 p = new class1();
            p.Test1();
            p.Test2();
            p.Test3();  // Here is an error parent class cannot access the child class member.
            Console.ReadLine();
        }
    }
}
```

(3) We can initialize a parent classes variable by using the child class instance to make it as a reference, so that the reference will be consuming the memory of child class instance, but in this case also we cannot call any pure child class members by using the reference.

**C**

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

namespace proj1
{
    class class2:class1
    {
         public class2()
        {
             Console.WriteLine("Class2 constructor is called");
        }
        public void Test3()
        {
            Console.WriteLine("Method 3");
        }
        static void Main()
        {
            Class1 p;//error p is a variable of class1 not an instance, it is a unassigned copy.
                    // we can call any members on a variable.

            p.Test1();
            p.Test2();
            p.Test3();  // Here is an error parent class cannot access the child class member.
            Console.ReadLine();
        }
    }
}
```

Another way to initialize parent instance in child class

**C**

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

namespace proj1
{
    class class2:class1
    {
         public class2()
        {
             Console.WriteLine("Class2 constructor is called");
        }
        public void Test3()
        {
            Console.WriteLine("Method 3");
        }
        static void Main()
        {
            Class1 p;// error p is a variable of class1 not an instance, it is a unassigned
                       // copy.
                    // we can call any members on a variable.

            Class2 c = new class2();
            P = c;
            p.Test2();
            p.Test2();  // Here is an error parent class cannot access the child class member.
            p.test3();  // error Class1 does not contain the definition for Test3.
            Console.ReadLine();
        }
    }
}
```

(4) Every class that is defined by us or pre-defined in the libraries of the language has a default parent class i.e. Object class of system namespace, so the member of Object class (**Equals, GetHashCode, GetType, ToString**)

These four methods are accessible from anywhere. Every class will contain these four methods. Because for every class the default class is Object Class.

**C**

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

namespace proj1
{
    class class2:class1
    {
         public class2()
        {
            Console.WriteLine("Class2 constructor is called");
        }
        public void Test3()
        {
            Console.WriteLine("Method 3");
        }
        static void Main()
        {
             Object obj = new Object(); //It will display four method
             Classs1 p = new class1();
             Console.ReadLine();
        }
    }
}
```

**Hierarchy sequence object always in top object is root of type Hierarchy, Object provides low level services to derived classes. This is the ultimate base class of all classes in the .Net Framework. Default parent class is object class present in the system namespace.**

Object
Class1
Class2
Class3

Class1 inherit from Object, class1 inherit from class2 and class3 inherit from class3
Notes : lower case show object in CSharo type and Object shows IL type, this object converted into IL type after compilation.

Q. **What is the default parent class for .Net?**
Ans. **Object class**

**C**

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

namespace proj1
{
    class class2:class1
    {
         public class2()
        {
            Console.WriteLine("Class2 constructor is called");
        }
        public void Test3()
        {
            Console.WriteLine("Method 3");
        }
        static void Main()
        {
             Object obj = new Object(); //It will display four method
             Console.WriteLine(obj.GetType()); //Return the type of obj
             Classs1 p = new class1();
             Console.WriteLine(p.GetType());//Return the type of p
             Classs2 c = new class2();
             Console.WriteLine(p.GetType());//Return the type of c
             Console.ReadLine();
        }
    }
}
```

**When I create the instance of Object class, it will start executing from the Object Class Constructor only, but I create the instance of Class1, it call two constructor Object Class Constructor and Class1 Class Constructor etc.**

**Type of Inheritance**

1. Single Inheritance
2. Multi-Level Inheritance
3. Hierarchical Inheritance
4. Hybrid Inheritance
5. Multiple Inheritance

But actually there is two type of inheritance.
1. Single Inheritance
2. Multiple Inheritance

If at all a class has 1 immediate parent class to it we call it as single inheritance and if it has more than 1 immediate parent class to it we call is as multiple Inheritance.

(5) In CSharp we don't have support for multiple inheritance thru classes, what we are provided is only single inheritance thru classes.

Hybrid and Multiple are inheritance not supported.

(6) In the first point we learnt when ever child class instance is created, child class constructor will implicitly call its parent classes constructor but only if the constructor of parent class has no parameter, where as if the constructor of parent class is parameterized, child class constructor can't implicitly call it's parent's constructor, so to overcome the problem it is the responsibility of the programmer to explicitly call parent classes constructor from child class constructor and pass values to those parameters. To call parent's constructor from child class we need to use the base keyword.

**A**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Claims;
using System.Text;
using System.Threading.Tasks;
namespace proj1
{
    class class1
    {
        public class1(int i) // parameterized constructor
        {
            Console.WriteLine("Class1 constructor is called"+i);
        }
        public void Test1()
        {
            Console.WriteLine("Method 1");
        }
        public void Test2()
        {
            Console.WriteLine("Method 2");
        }
    }
}
```

**C**

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

namespace proj1
{
    class class2:class1
    {
         public class2():base(10) //Explicit calling the parent function
        {
            Console.WriteLine("Class2 constructor is called");
        }
        public void Test3()
        {
           Console.WriteLine("Method 3");
        }
        static void Main()
        {
             Classs2 c = new class2();
             Console.ReadLine();
        }
    }
}
```

Here, we use static value 10 to pass to class1.

**A**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Claims;
using System.Text;
using System.Threading.Tasks;
namespace proj1
{
    class class1
    {
        public class1(int i) // parameterized constructor
        {
            Console.WriteLine("Class1 constructor is called"+i);
        }
        public void Test1()
        {
            Console.WriteLine("Method 1");
        }
        public void Test2()
        {
            Console.WriteLine("Method 2");
        }
    }
}
```

**C**

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

namespace proj1
{
    class class2:class1
    {
         public class2(int passa):base(passa) //Explicit calling the parent function
        {
            Console.WriteLine("Class2 constructor is called");
        }
        public void Test3()
        {
            Console.WriteLine("Method 3");
        }
        static void Main()
        {
             Classs2 c = new class2(50);
             Console.ReadLine();
        }
    }
}
```

Here, we use dynamic value to pass to class1, using ‘passa’ variable.
The value passa variable can also be used in class2 also.

*****

## Case study

How to use Inheritance in our applications? In heritance something that come into the picture not llike middle of project, when we start an application in initial stage only we plains this inheritance and we implement in our application.

In DBMS terminology, what is an **Entity**?
**Entity**: An Entity is something which is associated with a set of attributes, it can be living or non living object, but anything which is associated with a set of attributes that what we call as Entity in DBMS terminology.
Whenever we are going to develop an application, mainly we deal with Entity.
Suppose we are going to develop an application for Bank. In this case customer is an Entity, In case of School Application Student is an Entity.
e.g>
Pen is an Entity. (Company Name, Color, Price, Diameter, Height, Model Number)
Student is an Entity (Enrollment Number, Student's Name, Address, Mobile Number, Class)

Generally, when we developing an application. Process will be as following:

e.g> Lets we talk about School Application

**School Application**:

Step – 1 Identify the entities that are associated with the applications we are developing.

Student, teaching Staff, Non Teaching Staff are Entity.

Step – 2 Identify the attributes of each and every entity.

| Student | Teaching Staff | Non Teaching Staff |
|---|---|---|
| Id | Id | Id |
| Name | Name | Name |
| Address | Address | Address |
| Phone | Phone | Phone |
| Class | Designation | Designation |
| Marks | Salary | Salary |
| Grade | Qualification | Department |
| Fee | Subject | MgrId (Manager Id) |

There are a problem, here some common attributes in all the three Entity, If we start defining three classes separately it is code duplication (Id three time, Name three times, Address three times, Phone three times)
We have to eliminate the redundancy.

Step – 3 Identify the common attributes of each entity and put them in a hierarchical order. Here, Id, Name, Address, Phone attributes are some in these three Entities. Let us talk a class and define these common attributes in one class.

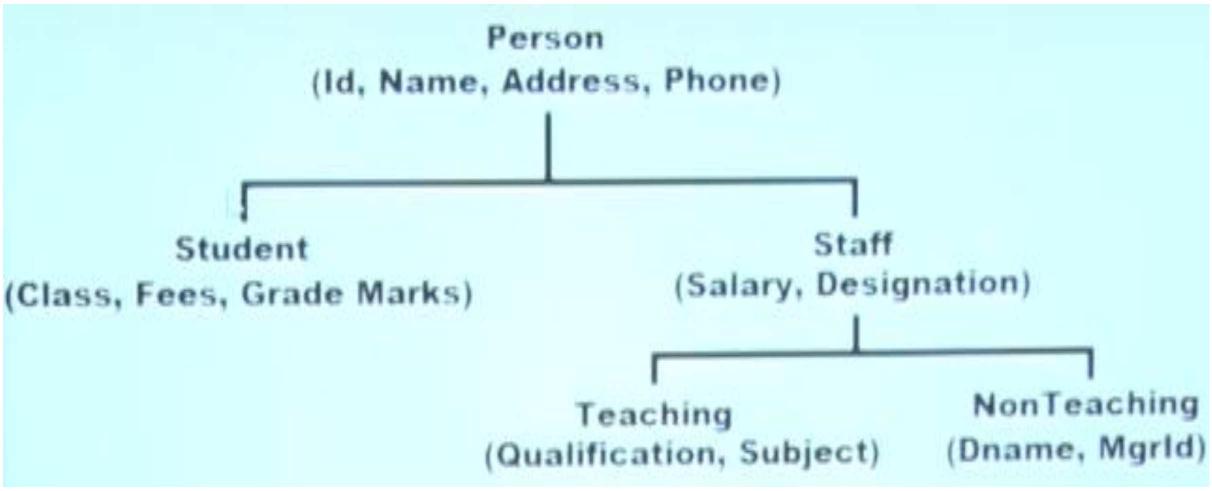
Generally, Parent classes use generic name to the classes.

Step – 4 Define the class representing the entities that are put in hierarchical order.

```
public class person
{
        String Id ;
        String Name, Address, Phone;
}

public class student : person
{
        String Class;
        Char Grade;
        float Marks, Fee;
}
```

Here, Teaching Staff and Non Teaching Staff again having two common attributes, these are Designation and Salary. To eliminate that redundancy we are going to define a class staff.

```
public class Staff : Person
{
        public string Designation;
        public double salary;
}

public class Teaching: Staff
{
        string Qualification, Salary;
}

public class Nonteaching : Staff
{
        string Department, MgrId;
}
```



*****