INDEX

[INheritance Mechanisms 3](#_Toc499155216)

[Syntax 3](#_Toc499155217)

[Override 4](#_Toc499155218)

[Abstract 4](#_Toc499155219)

[Interface 5](#_Toc499155220)

[Polymorphism 5](#_Toc499155221)

[Generics 7](#_Toc499155222)

[Constraints On Type Parameters 7](#_Toc499155224)

[Multiple Parameters 8](#_Toc499155240)

[Generic Methods 9](#_Toc499155241)

[Collection Framework 11](#_Toc499155242)

[Collection Classes In C# 11](#_Toc499155243)

[Collection Of Key/Value Pairs 13](#_Toc499155244)

[LINQ 13](#_Toc499155245)

[Iterators 14](#_Toc499155246)

[Functional Programming 15](#_Toc499155247)

[More About LINQ 16](#_Toc499155248)

[LINQ With Characters 22](#_Toc499155294)

[Meta‐Programming 23](#_Toc499155302)

[Attribute 25](#_Toc499155303)

[Predefined Annotations Types 25](#_Toc499155304)

[Attributeusage 26](#_Toc499155305)

[Conditional 26](#_Toc499155306)

[Obsolete 26](#_Toc499155308)

[Defining Your Own Attribute Type 27](#_Toc499155309)

[References 30](#_Toc499155310)

INheritance Mechanisms

# Inheritance gives ability to a child class to inherit behaviours from the parent class. The class whose members are inherited is called the base class. The class that inherits the members of the base class is called the derived class.

# C# support single inheritance. It means that a class can only inherit from a single class which is also valid in Java. However, inheritance is transitive, therefore we can define an inheritance hierarchy. For example, Z can inherit from Y, which inherits from X, which inherits from the base class W. The members of W are available to Z.

Syntax

# class NameofSubClass : NameOfSuperclass {….}

# One advantage of C# over Java in Inheritance mechanism is that while inheriting from other class or interface we do have to take care of the keyword “extends” and “implements”, the syntax is simple in C# i.e., always use “ : ” in both cases.

# Example:

### class Employee{

### public int salary= 10;

### }

### class Lawyer : Employee{

### public int GetSalary (){

### return this. salary;

### }

### }

### class Program{

### static void Main(string[] args){

### Lawyer b = new Lawyer();

### Console.WriteLine(b.GetSalary());

### Console.ReadKey();

### }

### }

# [File: Employ.cs]

# Derived classes methods can use base class methods/constructors:

# base.method(parameters) // method

# base(parameters); // constructor

### class Lawyer : Employee {

### // give Lawyers a $5K raise (better)

### public double getSalary() {

### double baseSalary= base.getSalary();

### return baseSalary + 5000.00;

### }

### }

# [File: Employ.cs]

Override

# The override modifier can be used to extend or modify the abstract or virtual implementation of an inherited method. When a base class declares a method as [virtual](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual), a derived class can [override](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/override) the method with its own implementation [C# Guide - https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/override].

# For example,

### class Employee{

### public int salary = 10;

### public virtual double Bonus(){

### return 10.00;

### }

### }

### class Lawyer : Employee{

### public int GetSalary(){

### return this.salary;

### }

### public override double Bonus(){

### return 20.00;

### }

### }

Abstract

# The abstract modifier indicates that the thing being modified has a missing or incomplete implementation. The abstract modifier can be used with classes, methods. Use the abstract modifier in a class declaration to indicate that a class is intended only to be a base class of other classes. Members marked as abstract, or included in an abstract class, must be implemented by classes that derive from the abstract class [C# Guide - <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract>].

# For Example,

### abstract class Employee{

### abstract public double Bonus();

### }

# [File: Employ.cs]

### class Lawyer : Employee{

### public override double Bonus(){

### return 20000.00;

### }

### }

# [File: Employ.cs]

Interface

# An interface contains elements and methods that a [class](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/class) can implement and change it according to its purpose.

# By using interfaces, we can implement behaviour from multiple interfaces in a class. Same is valid for Java. C# doesn't support multiple inheritance of classes.

# You define an interface by using the “[interface](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface)” keyword, as the following example shows.

### interface Shape{

### int Area(int a, int b);

### }

### class Square : Shape{

### public int Area(int side1,int side2){

### return side1\*side2;

### }

### }

# [File: Employ.cs]

# For inheriting from multiple interfaces, the syntax is as follows:

# class ClassName : InterfaceName1, InterfaceName2

Polymorphism

# Let’s understand this concept with the help of an example. Let’s create a base class called Shape, and derived classes Rectangle, Circle, and Triangle. “Shape” has a virtual method “Draw”, override it in each derived class to draw the particular shape that the class represents.

# Create a List<Shape> object and add all the derived class like Circle, Triangle and Rectangle to it.

### class Shape{

### public virtual void Draw(){

### Console.WriteLine("Performing base class drawing tasks");

### }

### }

### class Circle : Shape{

### public override void Draw(){

### Console.WriteLine("Drawing a circle");

### base.Draw();

### }

### }

### class Rectangle : Shape{

### public override void Draw(){

### Console.WriteLine("Drawing a rectangle");

### base.Draw();

### }

### }

# [File: Employ.cs]

### class Triangle : Shape{

### public override void Draw(){

### Console.WriteLine("Drawing a triangle");

### base.Draw();

### }

### }

### class Program{

### static void Main(string[] args){

### var shapes = new List<Shape>{

### new Rectangle(),

### new Triangle(),

### new Circle()

### };

### 

### foreach (var shape in shapes){

### shape.Draw();

### }

### Console.ReadKey();

### }

### }

# [File: Program.cs]

### Output:

### Drawing a rectangle

### Performing base class drawing tasks

### Drawing a triangle

### Performing base class drawing tasks

### Drawing a circle

### Performing base class drawing tasks

# **Note: Rest assured is same as Inheritance in Java.**

Generics

# Generics specify use of one or more type parameter until the class or method is declared and instantiated. Generic classes and methods combine reusability, type safety and efficiency in a way that the non-generic cannot. The .NET Framework has [System.Collections.Generic](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic) namespace which contains several generic-based collection classes.

# We can also create custom generic types and methods. By using generics, a collection becomes type-safe at compile-time. There is no wild card concept in C# because Generics in C# make stronger guarantees than generics in Java.

### List<int> list1 = new List<int>();

### list1.Add(3);

Example

### public class Box<T> {

### private T t;

### public void set(T t) { this.t = t; }

### public T get() { return t; }

### }

# // Instantiate a Generic Class

### Box<Integer> box=new Box<Integer>(); //Error: Typed namespace could not be found

### Box<int> box=new Box<int>(); //Correct

### Box<int> box=new Box<>(); //Error: Type Expected

### Box rawbox=new Box(); // Error: Using generic type 'Box<T>' requires 1 type arguments

# [File: Box.cs]

# Raw types are not supported by C#.

Constraints on Type Parameters

When we define a generic class, we can create bounds type that. If others tries to instantiate your class by using a type that is not allowed by the bound, the result is a compile-time error.

# These bounds are known as constraints in C#. Constraints can be specified by using “where”. The following table lists the six types of constraints:

| Constraint | Description |
| --- | --- |
| where T: struct | The type argument must be a value type. Any value type except [Nullable](https://docs.microsoft.com/en-us/dotnet/api/system.nullable) can be specified. |
| where T : class | The type argument must be a reference type; this applies also to any class, interface, delegate, or array type. |
| where T : new() | The type argument must have a public parameterless constructor. When used together with other constraints, the new() constraint must be specified last. |
| where T : <base class name> | The type argument must be or derive from the specified base class. |
| where T : <interface name> | The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic. |
| where T : U | The type argument supplied for T must be or derive from the argument supplied for U. |

# Source: C# Guide - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters

# There are no constraints for primitive data types in generics in C#. If we want to create a constraint for this, we can directly mention like List<string> instead of List<T>.

Multiple Parameters

# Applying constraints to multiple parameters, and multiple constraints to a single parameter.

### class Box { }

### class Test<P, Q>

### where Q : struct

### where P : Box, new() { }

# Example:

### interface Pair<K,V>{

### K getKey();

### V getValue();

### }

# [File: Pair.cs]

### class OrderedPair<K, V> : Pair<K, V> {

### private K key;

### private V value;

### public OrderedPair(K key, V value){ this.key= key; this.value= value; }

### public K getKey() { return key;}

### public V getValue(){ return value;}

### }

# // Instantiate a Multi Parameter Generic Class

### Pair<string, int> p1; p1 = new OrderedPair<string, int>("Even", 8);

### Pair<string, string> p2; p2 = new OrderedPair<string, string>("hello", "world");

# [File: OrderedPair.cs]

Generic Methods

# A generic method is a method that is declared with type parameters, as follows:

### class GenericMethod{

### public void Swap<T>(ref T[] array, int i, int j){

### T temp;

### temp = array[i];

### array[i] = array[j];

### array[j] = temp;

### }

### }

# [File: GenericMethod.cs]

### int[] array = new int[] { 1, 2, 3, 4, 5, 6 };

### GenericMethod gm = new GenericMethod();

### //gm.Swap<int>(ref array, 0, 3); //Correct

### gm.Swap(ref array, 0, 3); //Correct - We can also omit the type argument and the

### compiler will infer it.

### for (int i = 0; i < 6; i++) { Console.WriteLine(array[i]); }

# The above program will also work if we remove the “ref” from both method and calling, it is just a programming etiquette.

# **Note: Instead of Comparable<T>, C# has IComparable<T>, both works the same way. The inheritance property is similar as Java, still refer to the Constraints on Type Parameters Table.**

# Example: Compare the student name with the given name.

# Student.cs

### class Student : IComparable<Student>

### {

### public string Name { get; set; }

### public Student(string name)

### {

### Name = name;

### }

### public int CompareTo(Student other)

### {

### return this.Name.CompareTo(other.Name);

### }

### }

# [File: Student.cs]

# Program.cs

### static int myComparator<T>(T value) where T : IComparable<T>

### {

### return value.CompareTo(default(T));

### }

### Student st = new Student("Mark");

### Console.WriteLine(myComparator("x"));

### //Output: 1

# [File: Program.cs]

Collection Framework

# There are several ways for storing a data, i.e., using an array or using a collection. The problem with array is that they are of fixed size. But collection can be of dynamic size. Like java collection is a class in C#.

# Example:

### var c=new List<string>();

### List<string> list=new List<string>();

### ArrayList arraylist = new ArrayList(); // ArrayList is non-generic in C#

### arraylist.Add(1);

### arraylist.Add("hello");

### foreach (var al in arraylist)

### {

### Console.Write(al + " ");

### }

### Int32 i = (Int32)arraylist[0];

### Console.WriteLine(i);

### string s = (string)arraylist[1];

### Console.WriteLine(s);

# [File: GenericMethod.cs]

# ArrayList is non-generic in C#. Therefore one cannot restrict on what type of elements can get stored in an ArrayList. On the other hand List perform operation same as it does in Java. “.Add(any\_Element)” method adds the element in the list. We can also assign elements at the time of collection initialization like “List<string> list2 = new List<string> { "Hello", "How", "Are", "You" };”. If you want to remove an element from an array, C# provide two methods: list.Remove(element value) and list.RemoveAt(index value). We can also create a List<T> by defining our own class like:” var emp = new List<Employee>{};”.

# Unlike java, we don’t have to use Collections class for performing operation in a list. In C#, we can directly use the dot operator after the list name to perform different task. For example: Sort a list

# In Java: Colections.sort(collection\_name);

# In C#: collection\_name.Sort();

Collection Classes in C#

# Some of the most commonly used Collection class are:

# [System.Collections.Generic](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic) classes

# [System.Collections](https://docs.microsoft.com/en-us/dotnet/api/system.collections) classes

# The following table lists some of the frequently used classes of the [System.Collections.Generic](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic) namespace:

| **Class** | **Description** |
| --- | --- |
| [Dictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2) | Represents a collection of key/value pairs that are organized based on the key. |
| [List<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1) | Represents a list of objects that can be accessed by index. Provides methods to search, sort, and modify lists. |
| [Queue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1) | Represents a first in, first out (FIFO) collection of objects. |
| [SortedList<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedlist-2) | Represents a collection of key/value pairs that are sorted by key based on the associated [IComparer<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.icomparer-1) implementation. |
| [Stack<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.stack-1) | Represents a last in, first out (LIFO) collection of objects. |

# The following table lists some of the frequently used classes in the System.Collections namespace:

| **Class** | **Description** |
| --- | --- |
| [ArrayList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.arraylist) | Represents an array of objects whose size is dynamically increased as required. |
| [Hashtable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.hashtable) | Represents a collection of key/value pairs that are organized based on the hash code of the key. |
| [Queue](https://docs.microsoft.com/en-us/dotnet/api/system.collections.queue) | Represents a first in, first out (FIFO) collection of objects. |
| [Stack](https://docs.microsoft.com/en-us/dotnet/api/system.collections.stack) | Represents a last in, first out (LIFO) collection of objects. |

# Source: C# Guide - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/collections

Collection of Key/Value Pairs

# Like in java, we have a Collection that store elements in Key – Value format. In java, it is “Map”, in C# it is “Dictionary”. Names are different, both perform same functionality. In brief Dictionary is Map of C#. For example:

### var map = new Dictionary<int, string>();

### map.Add(1,"hello");

### map.Add(2, "how");

### map.Add(3, "are");

### map.Add(4, "you");

### map.Add(5, "!");

### //prinrting the dictionary in default manner

### foreach (var dict in map) { Console.Write(dict + " "); }

### //printing the dictionary using Key and Value method

### foreach (var dict in map) { Console.Write(dict.Key + " " +dict.Value + " "); }

# [File: GenericMethod.cs]

LINQ

# LINQ (Language-Integrated Query) is a non-platform based query that can be applied on any type of data stored. LINQ can be used to do operation on collection or on any database. The beauty of LINQ is that the syntax always remains the same whether you interact with collection or with any database.

# Example:

### //stores all the elements whose value == "hello"

### var linq = from e in map

### where e.Value == "hello"

### select e;

### foreach (var dict in linq) { Console.Write(dict.Key + " " + dict.Value + " "); }

### //Output: 1 hello

### //sort all elements based on Key

### var linq2 = from e in map

### orderby e.Value

### select e;

### foreach (var dict in linq2) { Console.Write(dict.Key + " " + dict.Value + " "); }

### //Output: 5 ! 3 are 1 hello 2 how 4 you

# [File: GenericMethod.cs]

Iterators

# An Iterators can be used to traverse through a collection.

# An iterator method or get accessor performs a custom iteration over a collection. An iterator method uses the [yield return](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/yield) statement to return each element one at a time. When a yield return statement is reached, the current location in code is remembered. Execution is restarted from that location the next time the iterator function is called.

# You consume an iterator from client code by using a [foreach](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/foreach-in) statement or by using a LINQ query [C# Guide - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/iterators].

# The return type of an iterator method or get accessor can be following:

# [IEnumerable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerable) – Exposes the enumerator, which supports a simple iteration over a non-generic collection.

# [IEnumerable<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1) - Exposes the enumerator, which supports a simple iteration over a collection of specified type.

# [IEnumerator](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerator) – Supports a simple iteration over a nongeneric collection.

# [IEnumerator<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerator-1) - Supports a simple iteration over a generic collection.

# For example:

### public static System.Collections.IEnumerable EvenNumber(List<int> list){

### foreach (var en in list){

### if (en % 2 == 0) {

### yield return en;

### }

### }

### }

### static void Main(string[] args){

### List<int> intlist = new List<int> { 5,4,3,2,1 };

### foreach (var i in EvenNumber(intlist)){

### Console.Write(i + " ");

### }

### }

### //Output: 4 2

# [File: Program.cs]

# We cannot convert one type of collection to another in C# as, for example, “List<T>” is a Generic but “ArrayList” is non generic.

Functional Programming

# There are several ways of using functional programming in C#. One of them is LINQ and another way is using “Func”.

# For example:

### //Prints out the squre of the given number

### Func<int, int> sq = x => x\*x;

### Console.WriteLine(sq(5));

# We can also apply this function on a list.

### List<int> intlist = new List<int> { 5,4,3,2,1 };

### Func<int, int> sq = x => x\*x;

### foreach(var i in intlist){Console.Write(sq(i)+” ”);}

### foreach (var i in intlist) { Console.Write(i+” ”); }

### //Output: 25 16 9 4 1

### 5 4 3 2 1

# [File: Program.cs]

# As we can see in the above example the original list does not get modified. The first “int” in Func<int, int> stands for the input type, second one is for return type. In case there are more than one input parameter: Func<input1 type, input2 type, return type>

# For example, we want to print the strings whose length is greater than the given length:

### List<string> friends = new List<string> {"Brian", "Nate", "Neal", "Raju", "Sara", "Scott" };

### Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;

### foreach (var i in friends) { Console.Write(isTooLong(4,i)?i:null); }

### //Output: BrainScott

# [File: Program.cs]

# Let’s modify the above example and count the numbers of strings whose length is greater than four (4):

### List<string> friends = new List<string> { "Brian", "Nate", "Neal", "Raju", "Sara", "Scott" };

### Console.WriteLine("The are "+friends.Count(a=>a.Length>4)+" strings in the friends list whose length is greater than 4");

### //Output: The are 2 strings in the friends list whose length is greater than 4

# [File: Program.cs]

# Let’s see some of the easier lambda in C#:

### friends.ForEach(i=>Console.WriteLine(i));

### // prints the whole list

### friends.ForEach(Console.WriteLine);

### //prints the whole list

### List<string> uppercaseNames = new List<string> {};

### friends.ForEach(name=>uppercaseNames.Add(name.ToUpper()));

### uppercaseNames.ForEach(Console.WriteLine);

### //convert the elements of friends to uppercase and stores them in another list

# [File: Program.cs]

# There is no concept of map and stream in C#. Instead C# has LINQ.

# For example, let’s try the above example of Uppercase

### List<string> uppercaselist = friends.Select(x => x.ToUpper()).ToList();

### uppercaselist.ForEach(Console.WriteLine);

More about LINQ

# There are two ways of writing LINQ Queries:

# LINQ query using Lambda Expression

### IEnumerable<string> f1 = friends.Where(f => f == "Nate");

# LINQ query using SQL like query expressions

### IEnumerable<string> f2 = from f in friends

### where f == "Nate"

### select f;

# To write LINQ queries we use the LINQ Standard Query Operators. The following are a few Examples of Standard Query Operators:

# select

# from

# where

# orderby

# join

# groupby

# Let’s see some of the aggregation operators provided by LINQ:

# Min

# Max

# Sum

# Count

# Average

# Aggregarte

# For example:

### int[] numbers = { 1,2,3,4,5,6,7,8,9,10};

### int smallestEvenNumber = numbers.Where(n => n % 2 == 0).Min(); Console.WriteLine(smallestEvenNumber); //2

### int largestEvenNumber = numbers.Where(n => n % 2 == 0).Max(); Console.WriteLine(largestEvenNumber); //10

### int sumOfAllEvenNumbers = numbers.Where(n => n % 2 == 0).Sum(); Console.WriteLine(sumOfAllEvenNumbers); //30

### int countAllEvenNumbers = numbers.Where(n => n % 2 == 0).Count(); Console.WriteLine(countAllEvenNumbers); //5

### double averageOfAllEvenNumbers = numbers.Where(n => n % 2 == 0).Average(); Console.WriteLine(averageOfAllEvenNumbers); //6

### int aggreagate = numbers.Aggregate((n1, n2) => n1 \* n2); Console.WriteLine(aggreagate); //3628800

# [File: Program.cs]

# As we can observe from above example that Aggregate might be looking similar to reduce in Java, and yes Aggregate is reduce of C#.

# Let’s print our friends list with each element separated by “, ” : Console.WriteLine(friends.Aggregate((name1,name2)=>name1+", "+name2));

# Now print the index position of all the even numbers from our numbers list:

### IEnumerable<int> evenIndexPositions = numbers

### .Select((num, index) => new { Number = num, Index = index })

### .Where(x => x.Number % 2 == 0)

### .Select(x => x.Index);

### foreach (int even in evenIndexPositions){Console.WriteLine(even);}

### //Output: 1 3 5 7 9

# [File: Program.cs]

# Let’s create an Employee class that has details about an employee and apply different task on Employee using LINQ:

# Employee.cs

### class Employee

### {

### public int EmployeeID { get; set; }

### public string FirstName { get; set; }

### public string LastName { get; set; }

### public string Gender { get; set; }

### public int AnnualSalary { get; set; }

### public static List<Employee> GetAllEmployees()

### {

### List<Employee> listEmployees = new List<Employee>

### {

### new Employee

### {

### EmployeeID = 100,

### FirstName = "Mary",

### LastName = "Muller",

### Gender = "Female",

### AnnualSalary = 45000

### },

### new Employee

### {

### EmployeeID = 101,

### FirstName = "Tom",

### LastName = "Daely",

### Gender = "Male",

### AnnualSalary = 60000

### },

### new Employee

### {

### EmployeeID = 102,

### FirstName = "Mike",

### LastName = "Mist",

### Gender = "Male",

### AnnualSalary = 72000

### },

### new Employee

### {

### EmployeeID = 103,

### FirstName = "Mary",

### LastName = "Lambeth",

### Gender = "Female",

### AnnualSalary = 48000

### },

### new Employee

### {

### EmployeeID = 104,

### FirstName = "Pam",

### LastName = "Penny",

### Gender = "Female",

### AnnualSalary = 84000

### },

### };

### return listEmployees;

### }

### }

# [File: Employee.cs]

# Display FirstName & Gender properties of all employees

### var result = Employee.GetAllEmployees().Select(emp => new

### {

### FirstName = emp.FirstName,

### Gender = emp.Gender

### });

### foreach (var emp in result)

### {

### Console.WriteLine(emp.FirstName + " - " + emp.Gender);

### }

### //Output:Mary - Female

### // Tom - Male

### // Mike - Male

### // Mary - Female

### // Pam – Female

# Display employees by FirstName in ascending order

### //Method 1

### IEnumerable<Employee> result2 = Employee.GetAllEmployees().OrderBy(emp => emp.FirstName);

### foreach (Employee emp in result2)

### {

### Console.WriteLine(emp.FirstName);

### }

### //Method 2

### IEnumerable<Employee> result3 = from emp in Employee.GetAllEmployees()

### orderby emp.FirstName

### select emp;

### foreach (Employee emp in result3)

### {

### Console.WriteLine(emp.FirstName);

### }

### //Output: Mary Mary Mike Pam Tom

### 

# Display employees by FirstName in descending order

### //Method 1

### IEnumerable<Employee> result4 = Employee.GetAllEmployees().OrderByDescending(emp => emp.FirstName);

### foreach (Employee emp in result4)

### {

### Console.WriteLine(emp.FirstName);

### }

### //Method 2

### IEnumerable<Employee> result5 = from emp in Employee.GetAllEmployees()

### orderby emp.FirstName descending

### select emp;

### foreach (Employee emp in result5)

### {

### Console.WriteLine(emp.FirstName);

### }

### //Output: Tom Pam Mike Mary Mary

# Order by FirstName and then by EmployeeID

### //Method 1

### IEnumerable<Employee> result6 = Employee.GetAllEmployees()

### .OrderBy(s => s.FirstName).ThenBy(s => s.EmployeeID);

### foreach (Employee emp in result6)

### {

### Console.WriteLine(emp.FirstName + "\t" + emp.EmployeeID);

### }

### //Method 2

### IEnumerable<Employee> result7 = from emp in Employee.GetAllEmployees()

### orderby emp.FirstName, emp.EmployeeID

### select emp;

### foreach (Employee emp in result7){

### Console.WriteLine(emp.FirstName + "\t" + emp.EmployeeID);

### }

### //Output: Mary 100

### // Mary 103

### // Mike 102

### // Pam 104

### // Tom 101

# Takemethod returns a specified number of elements from the start of the collection. The number of items to return is specified using the count parameter this method expects (PRAGIM Technology - http://csharp-video-tutorials.blogspot.com/2014/07/linq-tutorial.html).

//Take first 3 employee

var result8 = Employee.GetAllEmployees().Select(emp => emp.FirstName).Take(3);

foreach (var emp in result8)

{

Console.WriteLine(emp);

}

//Output: Mary Tom Mike

# Skipmethod skips a specified number of elements in a collection and then returns the remaining elements. The number of items to skip is specified using the count parameter this method expects (PRAGIM Technology - http://csharp-video-tutorials.blogspot.com/2014/07/linq-tutorial.html).

//Skip first 3 employee

var result9 = Employee.GetAllEmployees().Select(emp => emp.FirstName).Skip(3);

foreach (var emp in result9)

{

Console.WriteLine(emp);

}

//Output: Mary Pam

# Group

//count employee by gender

var result10 = from emp in Employee.GetAllEmployees()

group emp by emp.Gender;

foreach (var emp in result10)

{

Console.WriteLine("{0} - {1}", emp.Key, emp.Count());

}

//Output: Female - 3

// Male - 2

//count employee by gender and display their firstname

var result11 = from emp in Employee.GetAllEmployees()

group emp by emp.Gender;

foreach (var emp in result10)

{

Console.WriteLine("{0} - {1}", emp.Key, emp.Count());

Console.WriteLine("----------");

foreach (var employee in emp)

{

Console.WriteLine(employee.FirstName + "\t" + employee.Gender);

}

Console.WriteLine();

}

//Output: Female - 3

// ----------

// Mary Female

// Mary Female

// Pam Female

//

// Male - 2

// ----------

// Tom Male

// Mike Male

# [File: Program.cs]

LINQ with Characters

string str ="w00t";

IEnumerable<char> strchar = str.Select(s => s);

foreach (char c in strchar)

{

Console.Write(c+" ");

}

//Output: w 0 0 t

# [File: Program.cs]

Meta‐Programming

# Reflection provides us power to analyse the metadata in the assembly at runtime.

# Reflection is useful in the following situations:

# Need to access attributes in your program's metadeta.

# To examine and instantiate types in an assembly.

# To build new types at runtime using classes in System.Reflection.Emit.

# To perform late binding, accessing methods on types created at run time.

# .NET assembly consists of four parts

# Assembly manifest

# Type Metadeta

# MSIL code

# Resource

# [Source: C Shrap Corner - <http://www.c-sharpcorner.com/UploadFile/d0a1c8/collection-framework-reflection-in-C-Sharp/>]

# The metadata draws data from the program stored in main memory (during execution) or any library file, the runtime loads the metadata into memory and it detect infromation about the classes, members, methods, constructors, inheritance, and so on. Reflection provides ability to examine the metadata at runtime. It is used dynamically to create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its field and properties.

# The package that helps developers to perform Reflection in C# is “System.Reflection”.

# Let’s understand reflection in C# with some simple examples: In this example we will be printing out all the information about our project.

### Type[] type;

### Assembly assembly = Assembly.GetExecutingAssembly(); // Gets the assembly that contains the code that is currently executing

### type = assembly.GetTypes();

### foreach (Type t in type)

### {

### Console.WriteLine(t.Name);

### MemberInfo[] members = t.GetMembers();

### MemberTypes memberType;

### foreach (MemberInfo m in members)

### {

### memberType = m.MemberType;

### switch (memberType)

### {

### case MemberTypes.Property: Console.WriteLine("Properties:");

### Console.WriteLine(m.Name);

### break;

### case MemberTypes.Method: Console.WriteLine("Methods:");

### Console.WriteLine(m.Name);

### break;

### case MemberTypes.Constructor: Console.WriteLine("Constructors:");

### Console.WriteLine(m.Name);

### break;

### }

### }

### }

# [File: Program.cs]

# In the next example we will performing the same task as above but at Runtime.

### Assembly assembly2 = Assembly.GetExecutingAssembly();

### MemberInfo method;

### object resulta = new object();

### object[] arg = new object[] { 30, 20 };

### Type[] types = assembly2.GetTypes();

### foreach (Type t in types)

### {

### method = t.GetMethod("sub");

### if (method != null)

### {

### string typeName = t.FullName;

### object obj = assembly2.CreateInstance(typeName);

### resulta = t.InvokeMember(method.Name, BindingFlags.Public | BindingFlags.InvokeMethod | BindingFlags.Instance, null, obj, arg);

### break;

### }

### }

### Console.WriteLine("Result is : {0}", resulta);

# [File: Program.cs]

# In java, the class on which reflection is to be performed is specified as “Class” type. In C#, same can be done using “Assembly” type. In java, ClassMember is used to get all the members of a class. In C#, we have to use “GetMembers()” method and store its value in a “MemberInfo” object. And for accessing different members, we can specify MemberTypes.Class\_memeber\_Type that we require. Like MemberTypes.Method and MemberTypes.Constructor.

Attribute

# Attribute is a form of metadata which provides information about a program that is not part of the program itself. Attribute is Annotation of C#. Attribute use “System.Diagnostics” package.

# Syntax:

# [attribute(target\_parameter, parameter2 = value, ...)]

# The attribute[] performs same task as Annotation of Java.

# The list of possible target values is shown in the following table.

| **Target value** | **Applies to** |
| --- | --- |
| assembly | Entire assembly |
| module | Current assembly module |
| field | Field in a class or a struct |
| event | Event |
| method | Method or get and set property accessors |
| param | Method parameters or set property accessor parameters |
| property | Property |
| return | Return value of a method, property indexer, or get property accessor |
| type | Struct, class, interface, enum, or delegate |

# Source: C# Guide - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/

Predefined Annotations Types

# The .Net Framework provides three pre-defined attributes −

# AttributeUsage - describes how an attribute class can be used

# Conditional – the specifede code execution depends on a specified condition

# Obsolete - marks a program entity that should not be used, just like Deprecated in java

AttributeUsage

# Example: [AttributeUsage (AttributeTargets.ALL, Inherited = inherited)]

# The “AttributeTargets.ALL” consider all the members of a class. If we want to specify the class member, we can specify by “AttributeTargets.Member\_Type” like AttributeTargets.Constructor, AttributeTargets.Field, AttributeTargets.Method. The inherited is same as of Java’s inherited annotation.

Conditional

Example: [Conditional("DEBUG")]

# The Conditional attribute only execute if the condition is true. In our example, we have specified the configuration. .NET mainly has two configuration: Debug and Release. Debug is used while developing the project and once the project becomes live the configuration is changed to Release.

### class attribute

### {

### [Conditional("DEBUG")]

### public void DebugConfig(){

### Console.WriteLine("DEBUG");

### }

### }

### 

### attribute attr = new attribute();

### attr.DebugConfig();

# [File: attribute.cs]

# The following code will output “DEBUG” only if we are in a debug configuration.

Obsolete

# Two variant:

# [Obsolete (message)] – Replace “message” by describing why the block of code is obsolete

# [Obsolete (message , iserror)] – “iserror” is a Boolean value, by default its false

# Example:

### class Obsolete

### {

### [Obsolete("Don't use the ObsoleteMethod, instead use the ReplacementOfObseleteMethod", true)]

### public void ObsoleteMethod() {

### Console.WriteLine("Obsolete Method");

### }

### public void ReplacementOfObseleteMethod(){

### Console.WriteLine("Replacement Of Obselete Method");

### }

### }

# [File: Obsolete.cs]

### Obsolete o = new Obsolete();

### o.ObsoleteMethod();

# [File: Proram.cs]

# The above code generates a compile time error saying “[deprecated] void Obsolete.ObsoleteMethod(). Don't use the ObsoleteMethod, instead use the ReplacementOfObseleteMethod”.

Defining your own Attribute Type

# The logic behind creating our own attribute and using it is same as it is for annotation in Java. If there is a difference, that will be in Syntax. Which we will discuss in this section.

# Syntax:

### public class MyAttribute : Attribute

### {...}

### [MyAttribute]

### public class ClassForImplementingMyAttribute

### {...}

# Example: Let’s create an Attribute class that displays the name of the person that wrote the method of another class and combine it with Reflection.

# MyAttribute.cs

### [AttributeUsage(AttributeTargets.All, AllowMultiple = true)]

### class MyAttribute : Attribute

### {

### private string name;

### private int age;

### private string gender;

### public string remark;

### 

### public MyAttribute(string name, int age, string gender) {

### this.name=name;

### this.age=age;

### this.gender=gender;

### }

### public string Name { get { return name; } }

### public int Age { get { return age; } }

### public string Gender { get { return gender; } }

### public string Remark { get { return remark; } set { remark = value; } }

### }

# [File: MyAttribute.cs]

# ClassForImplementingMyAttribute.cs

### [MyAttribute("Mayank",21,"Male",Remark="Hello from Mayank")]

### [MyAttribute("Mary", 20, "Female", Remark = "Hello from Mary")]

### class ClassForImplementingMyAttribute

### {

### protected int a;

### protected int b;

### public ClassForImplementingMyAttribute(int a, int b)

### {

### this.a = a;

### this.b = b;

### }

### [MyAttribute("Mayank", 21, "Male", Remark = "Add Method written")]

### public int Add()

### {

### return a + b;

### }

### [MyAttribute("Mayank", 21, "Male")]

### public void Print()

### {

### Console.WriteLine("a= "+ a);

### Console.WriteLine("b= "+ b);

### Console.WriteLine("c= "+ Add());

### }

### }

# [File: ClassForImplementingMyAttribute.cs]

# Program.cs

### ClassForImplementingMyAttribute cfima = new ClassForImplementingMyAttribute(10,20);

### cfima.Print();

### Type type = typeof(ClassForImplementingMyAttribute);

### foreach (Object attributes in type.GetCustomAttributes(false))

### {

### MyAttribute ma = (MyAttribute)attributes;

### if (null != ma)

### {

### Console.WriteLine("Name: " + ma.Name+"; Age: "+ma.Age+"; Gender: "+ ma.Gender+"; Remark: "+ma.Remark);

### }

### }

### foreach (MethodInfo m in type.GetMethods())

### {

### foreach (Attribute a in m.GetCustomAttributes(true))

### {

### MyAttribute ma = (MyAttribute)a;

### if (null != ma)

### {

### Console.WriteLine("Name: {0}; for Method: {1}", ma.Name, m.Name);

### Console.WriteLine("Age: {0}; Gender: {1}; Remark: {2}", ma.Age, ma.Gender, ma.Remark);

### }

### }

### }

# [File: Program.cs]

# The Output of the following example is:

# a= 10

# b= 20

# c= 30

# Name: Mary; Age: 20; Gender: Female; Remark: Hello from Mary

# Name: Mayank; Age: 21; Gender: Male; Remark: Hello from Mayank

# Name: Mayank; for Method: Add

# Age: 21; Gender: Male; Remark: Add Method written

# Name: Mayank; for Method: Print

# Age: 21; Gender: Male; Remark:

References

# Microsoft CLR via C# Fourth Edition – Jeffrey Richter

# C# Guide - <https://docs.microsoft.com/en-us/dotnet/csharp/index>

# LINQ Tutorials by PRAGIM Technology - <http://csharp-video-tutorials.blogspot.com/2014/07/linq-tutorial.html>

# Collection Framework and Reflection in C# - http://www.c-sharpcorner.com/UploadFile/d0a1c8/collection-framework-reflection-in-C-Sharp/

# Tutorials Point - https://www.tutorialspoint.com/csharp/csharp\_attributes.htm