Chương 5 Generic,Delegagte & Dynamic

Sau khi học xong bài này, học viên có khả năng :

- *Trình bày được kiểu Generic, Delegate*
- *Mô tả được cách sử dụng kiểu Dynamic với Lớp & Delegate*
- *Sử dụng được Method Generic, Collection Generic và Class Generic*
- *Trình bày được biểu thức Lambda và biểu thức Query*
- *Thực hiện được các thao tác trên Mảng , Tập Hợp sử dụng LINQ to Object*
- *Xây dựng được thư viện liên kết động (tập tin .dll) sử dụng Class Library project type*

5.1 Kiểu Generic

Generics là một phần kiểu hệ thống của .NET Framework cho phép bạn định nghĩa một kiểu bỏ đi một số chi tiết không xác định. Thay vì xác định rõ các kiểu của các tham số của phương thức , biến thành phần của lớp, bạn có thể cho phép mã lệnh sử dụng kiểu của bạn để chỉ định nó. Điều này cho phép mã lệnh sử dụng của bạn biến đổi thích ứng kiểu của bạn tới chỉ định cần thiết của chính nó.

Phiên bản 2.0 của .NET Framework chứa một số lớp generic khác nhau trong namespace có tên là System.Collections.Generic, bao gồm Dictionary, Queue, SortedDictionary, và SortedList. Các lớp này làm việc tương tự như đối với các phần tương ứng nongeneric của nó trong System.Collections, tuy nhiên chúng mang đến hiệu suất được cải thiện và tính an toàn kiểu.

- **Tại sao phải sử dụng Generics?**

  Phiên bản 1.0 và 1.1 của .NET Framework không hỗ trợ generics. Thay vào đó, lập trình viên sử dụng lớp Object với các tham số và thành viên sẽ phải chuyển đổi tới các lớp khác dựa trên lớp Object. Generics mang đến hai tính năng cải tiến đáng kể đối với việc sử dụng lớp Object :

- Giảm bớt lỗi vận hành (Reduced run-time errors): Trình biên dịch không thể kiểm tra các lỗi kiểu dữ liệu khi bạn chuyển đổi qua đựa trên lớp Object . Ví dụ, nếu bạn chuyển kiểu một chuỗi tới một lớp đối tượng và sau đó áp dụng việc chuyển đổi Object đó tới kiểu integer, trình biên dịch sẽ không catch lỗi. Thay vào đó, công cụ vận hành sẽ throw một exception (đưa ra một thông báo ngoại lệ).Việc sử dụng generics cho phép trình biên dịch catch loại lỗi này trước khi chương trình của bạn thi hành. Ngoài ra, bạn có thể chỉ định các ràng buộc để giới hạn các lớp sử dụng trong một generic, cho phép trình biên dịch kiểm tra phát hiện một kiểu không phù hợp.

- Hiệu suất được cải thiện (Improved performance) : Việc chuyển đổi đòi hỏi boxing và unboxing. Việc sử dụng generics không đòi hỏi phải casting hay boxing, nó cải thiện hiệu suất run-time.

5.2 Sử Dụng Class Generic

Để sử dụng Class Generic , ta khai báo các lớp sau: Lớp ObjClass và lớp GenClass , hai lớp này thực thi cùng các tác vụ như nhau, tuy nhiên ObjClass sử dụng lớp Object cho phép kiểu bất kỳ được gởi tới, trong khi đó lớp GenClass sử dụng generic.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace TestGeneric{
    //Khai bao voi kieu Object
    class ObjClass {
        public Object val_1;
        public Object val_2;
        public ObjClass(Object v1, Object v2)
        {
            val_1 = v1;
            val_2 = v2;
        }
    }
```

```
//Khai bao voi kieu Generic
class GenClass<T,U> {
    public T val_1;
    public U val_2;
    public GenClass(T v1, U v2)
    {
        val_1 = v1;
        val_2 = v2;
    }
}
```

Trong đó T và U là kiểu dữ liệu (string hay int..) được gửi tới, chúng ta có thể thấy, lớp ObjClass có hai thành viên kiểu Object. Lớp GenClass có hai thành viên kiểu T và U. Mã lệnh sử dụng sẽ kiểm tra kiểu đối với T và U. Với việc phụ thuộc vào cách mã lệnh sử dụng dùng lớp GenClass, T và U có thể là một string, một int, một lớp theo yêu cầu, hoặc sự kết hợp bất kỳ của những cái đó. Có một sự giới hạn đáng kể đối với việc tạo một lớp generic: mã lệnh generic chỉ hợp lệ nếu nó sẽ biên dịch mọi instance của generic có thể được khởi tạo, dù là một int, một string, hay bất kỳ một lớp nào khác. Trong trường hợp này chúng ta không thể sử dụng toán tử + , - , hay > cho kiểu generic .

- **Cách dùng một Generic Type**

  Khi chúng ta dùng một kiểu generic thì phải chỉ định các kiểu cho các generic bất kỳ được sử dụng. Để sử dụng 2 lớp trên ta viết lệnh sau :

```
class Program {
    static void Main(string[] args)
    {
        //su dung kieu object
        ObjClass obj1 = new ObjClass(1,2);
        //thuc hien chuyen kieu
        int result1  = (int)obj1.val_1 + (int)obj1.val_2;
        Console.WriteLine(result1);
        //Su dung kieu generic
        GenClass<int, int> obj2 = new GenClass<int, int>(2, 3);
        int result2 = obj2.val_2 + obj2.val_2;
        Console.WriteLine(result2);
    }
}
}
```

Khi thi hành các lệnh trong Main() , cho ta kết quả cộng 2 số nguyên. Tuy nhiên, mã lệnh sử dụng lớp GenClass trên thực tế sẽ làm việc nhanh hơn bởi vì nó không đòi hỏi boxing và unboxing xuất phát từ lớp Object . Ngoài ra, người lập trình cũng sẽ có nhiều thời gian thoải mái hơn với việc sử dụng lớp GenClass là chúng ta sẽ không phải cast (chuyển kiểu) thủ công từ lớp Object tới các kiểu tương thích.

5.3 Sử Dụng Collection Generic

Trong C#, khi cần sử dụng một mảng động chứa các phần tử kiểu int, string ,… chúng ta có thể sử dụng tập hợp: List<T>, tập hợp này có thể chứa bất kỳ kiểu gì.
Để sử dụng List<T>, ta tạo mới ứng dụng Console và viết code như sau :

```
class Program {
    static void Main(string[] args) {
        //Su dung Collection Generic : List<>
        //Khai báo List chua cac phan tu kieu int
        List<int> lstInt = new List<int>();
        //gan gia tri cho cac phan tu
        lstInt.Add(1); lstInt.Add(2); lstInt.Add(3);
        //Xuat cac phan tu trong listInt
        Console.WriteLine(" Cac phan tu trong lstInt");
        foreach (var phantu in lstInt) {
            Console.Write("{0,5}",phantu);
        }
        //Khai báo List chua cac phan tu kieu string
        List<string> lstString = new List<string>();
        //gan gia tri cho cac phan tu
        lstString.Add("Lap trinh");
        lstString.Add("ASP.NET");
        lstString.Add("4.0");
        //Xuat cac phan tu trong lstString
        Console.WriteLine("\n Cac phan tu trong lstString");
        foreach (var phantu in lstString) {
            Console.Write("{0,10}", phantu);
        }
        Console.WriteLine("\n-------------****-------------");
    }
}
```

Kết quả thực thi chương trình như sau :

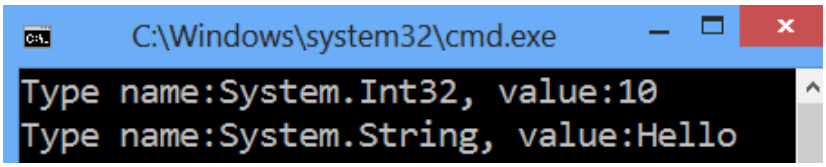
```
Cac phan tu trong lstInt
   1    2    3
Cac phan tu trong lstString
Lap trinh   ASP.NET       4.0
------------****------------
```

5.4 Sử Dụng Method Generic

Trong C#, thông thường chúng ta sử dụng phương thức với các tham số và kiểu trả về cố định như : int, string ,… Bây giờ chúng ta có thể sử dụng các phương thức generic với danh sách tham số và kiểu trả về bất kỳ .
Để sử dụng phương thức generic, ta tạo mới ứng dụng Console và viết code như sau :

```
class Program{
    //Khai bao phuong thuc generic xuat kieu va gia tri mot bien
    static void Print<T>(T param){
        Console.WriteLine("Type name:{0}, value:{1}",
            param.GetType(),param);
    }
    static void Main(string[] args)
    {
        int a = 10;
        string s = "Hello";
        //xuat gia tri kieu int
        Print<int>(a);
        //xuat gia tri kieu string
        Print<string>(s);
    }
}
```

Kết quả thực thi chương trình như sau :



5.5 Delegate trong C#

Delegate là đối tượng dùng tạo để tham chiếu đến các phương thức và thực thi các phương thức mà delegate đã tham chiếu.

5.5.1. Khai báo Delegate

- Đầu tiên ta phải định nghĩa delegate mà ta muốn dùng ,nghĩa là bảo cho trình biên dịch biết loại phương thức mà delegate sẽ trình bày. Sau đó ta tạo ra các thể hiện của delegate.

Cú pháp:

```
delegate void VoidOperation(uint x);
```

- Ta chỉ định mỗi thể hiện của delegate có thể giữ một tham chiếu đến 1 phương thức mà chứa một thông số uint và trả về kiểu void.

- Ví dụ khác: nếu bạn muốn định nghĩa 1 delegate gọi là twolongsOp mà trình bày 1 hàm có 2 thông số kiểu long và trả về kiểu double. ta có thể viết :

```
delegate double TwoLongsOp(long first, long second);
```

hay 1 delegate trình bày phương thức không nhận thông số và trả về kiểu string.

```
delegate string GetAString();
```

- Cú pháp cũng giống như phương thức, ngoại trừ việc không có phần thân của phương thức, và bắt đầu với delegate, ta cũng có thể áp dụng các cách thức truy nhập thông thường trên một định nghĩa delegate – public, private, protected …

```
public delegate string GetAString(string s);
```

- Mỗi lần ta định nghĩa một delegate chúng ta có thể tạo ra một thể hiện của nó mà ta có thể dùng để lưu trữ các chi tiết của 1 phương thức cụ thể. *Lưu ý*: với lớp ta có 2 thuật ngữ riêng biệt: lớp dùng để chỉ định nghĩa chung và đối tượng dùng để chỉ một thể hiện của 1 lớp, tuy nhiên đối với delegate ta chỉ có một thuật ngữ là "1 delegate" khi tạo ra một thể hiện của delegate ta

cũng gọi nó là delegate. Vì vậy cần xem xét ngữ cảnh để phân biệt.

Đoạn mã sau minh hoạ cho 1 delegate:

```
static string Msg(string s){
   return "Hello :"+s;
}
static void Main(string[] args)
{
   string s = "Tom";
   GetAString firstStringMethod = new GetAString(Msg);
   Console.WriteLine("String is" + firstStringMethod(s));
}
```

- Trong đoạn mã này, ta tạo ra delegate GetAString, và khởi tạo nó để nó tham khảo đến phương thức Msg() .Chúng ta sẽ biên dịch lỗi nếu cố gắng khởi tạo FirstStringMethod với bất kì phương thức nào có tham số và kiểu trả về khác kiểu chuỗi.

- Một đặc tính của delegate là an kiểu an toàn (type-safe) để thấy rằng chúng phải đảm bảo dấu ấn (signature) của phương thức được gọi là đúng và khi sử dụng delegate, chúng ta có thể tham chiếu đến phương thức static hay là phương thức của đối tượng.

5.5 Dynamic trong C#

Trong C# 4.0, cũng như các phiên bản trước đây, hầu hết các kiểu dữ liệu (có sẵn, do người dùng định nghĩa, kiểu tham chiếu, kiểu giá trị) đều thừa kế một cách trực tiếp hay gián tiếp từ Object. Do vậy, một cách gần đúng, ta có thể gán một giá trị có kiểu bất kỳ cho một biến kiểu Object.

Ví dụ:

```
int temp = 123;
Object obj = temp;
Console.Write(obj.ToString());
```

Tuy nhiên, việc kiểm tra kiểu của các đối tượng sẽ được tiến hành tại thời điểm biên dịch chương trình. Do đó nếu sử dụng không đúng cách sẽ gây ra những phiền toái. Bạn hãy xem ví dụ sau:

```
object obj = 10;
Console.WriteLine(obj.GetType());
```

Tất nhiên dòng lệnh này sẽ xuất ra System.Int32, vì đây là kiểu của giá trị đang được chứa trong obj.

Bây giờ giả sử ta muốn cộng một số vào obj:

```
obj = (int)obj + 10;
```

Điều này hoàn toàn hợp lệ!

Tuy nhiên, vì một bất cẩn (hay cố tình?) nào đó có thể bạn sử dụng

```
obj = (double)obj + 10;
```

Và trình biên dịch hoàn toàn không phát hiện được lỗi này.

Do đó C#4.0 đưa vào khái niệm Dynamic. Đối tượng có kiểu Dynamic không được xác định cho đến khi chạy chương trình (run time). Do đó khi sử dụng Dynamic. Bạn không cần quan tâm cái gì được gán cho đối tượng Dynamic.

Giả sử ta có

```
dynamic dyn = 10;
Console.WriteLine(dyn.GetType());
```

Việc này chưa có gì khác biệt với cách sử dụng object ở trên

Và việc gán

```
dyn = dyn + 10;
```

hoàn toàn hợp lệ. Không những thế nó còn đúng cho tất cả các kiểu dữ liệu khác có hỗ trợ phép toán "+"

```
dyn = 10.0;
dyn = dyn + 10;
dyn = "10";
```

dyn = dyn + 10**;**

Một ví dụ khác minh họa dynamic:

```
static void Main(string[] args)
{
  dynamic temp = 1;  //temp co kieu int
  //result = 3
  Console.WriteLine("result = {0}",temp + 2);
  temp = "Hello";    //temp co kieu string
  //result = Hello World
  Console.WriteLine("result = {0}", temp + " World");
}
```

5.6 Lambda Expressions C#

Biểu thức lambda được chỉ định như là các tham số được phân cách bởi dấu "," theo sau là toán tử lambda "**=>**" hoặc một biểu thức hoặc một khối lệnh.

Cú pháp :

```
(param1, param2, ...paramN) => expr
hay :
(param1, param2, ...paramN) =>
{
 statement1;
 statement2;
 ...
 statementN;
 return(lambda_expression_return_type);
}
```

```
class Program
{
  //Khai báo delegate gọi đến phương thức cộng
  public delegate int GoiPhuongThuc(int a, int b);
  public static int Calc(GoiPhuongThuc f, int a, int b)
  {
      return f(a, b);
  }
  public static int Cong(int a, int b)
  {
      return a + b;
  }
  static void Main(string[] args)
  {
      Console.WriteLine("Minh hoa Lambda Expressions");
      int x = 1,y = 2;
      //Khai báo biểu thức lambda
      int ketqua = Calc((a, b) => a + b, x, y);
      Console.WriteLine("Ket qua : {0} + {1} = {2}", x, y, ketqua);
  }
}
```

- Kết quả thi hành

5.7 Query Expression

Biểu thức Query cho phép chúng ta viết các lệnh truy vấn trong LINQ gần giống với các lệnh truy vấn trong T-SQL.

Cú pháp :

from itemName in srcExpr

join itemName in srcExpr on keyExpr equals keyExpr

(into itemName)?

let itemName = selExpr

where predExpr

*orderby (keyExpr (ascending | descending)?)**

select selExpr

group selExpr by keyExpr

into itemName query-body

Ví dụ minh họa : Biểu thức Query trong LINQ

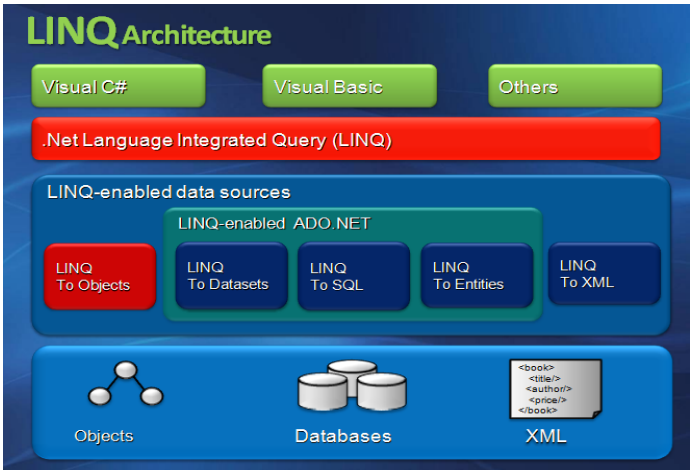
```
class Program
{
    static void Main(string[] args)
    {
        //Khai báo mảng chứa danh sách các tên
        string[] danhsach = { "An", "Tuan",
                              "Hung", "Lan",
                              "Khanh", "Toan",
                              "Vu", "Huy" };
        var danhsachchonloc = from s in danhsach
                    //lấy những tên có chiều dài = 2
                   where s.Length == 2
                   orderby s ascending // sắp xếp tăng dần
                   // chuyển tên được chọn sang chữ HOA
                   select s.ToUpper();
        //Xem danh sách các tên được chọn
        foreach (string ten in danhsachchonloc)
            Console.WriteLine(ten);
        Console.ReadLine();
    }
}
```

- Kết quả thi hành:

5.8 LINQ to Objects

LINQ to Object dùng để truy vấn, sắp xếp ,lọc dữ liệu trong các đối tượng như : mảng , tập hợp

Ví dụ . Minh họa sử dụng LINQ to Object để chuyển 1 mảng các chuỗi số sang 1 mảng các số nguyên.



```
class Program
{
    static void Main(string[] args)
    {
        //Khai báo mảng các chuỗi số
        string[] MangChuoiSo = { "0042", "010", "9", "27" };
        //Chuyển sang mảng các số nguyên
        int[] MangSoNguyen = MangChuoiSo
                    .Select(s => Int32.Parse(s))
                    .OrderBy(s => s).ToArray();
        //Xem danh sách mảng sau khi chuyển
        foreach (int SoNguyen in MangSoNguyen)
            Console.Write(" {0} ", SoNguyen);

        Console.ReadLine();
    }
}
```

◈ Kết quả thi hành

5.8 Xây Dựng Thư Viện Liên Kết Động

5.8.1 Dll Là Gì ?

- DLL là viết tắt của Dynamic Link Library (thư viện liên kết động).
- DLL chứa các đoạn code (đã được biên dịch) mà có thể được sử dụng bởi hơn một ứng dụng tại cùng một thời điaểm.
- Trong Windows thì có rất rất nhiều các DLL cung cấp các tính năng được xây dựng sẵn để cho phép các ứng dụng khác có thể sử dụng nó.
- Một vài ưu điểm khi sử dụng DLL:
    - Các DLL sử dụng ít tài nguyên hơn.
    - DLL giúp cho việc tạo các chương trình module.
    - Các DLL là dễ dàng triển khai và cài đặt.

❖ **Các Hạn Chế Của Dll:**

- Không hỗ trợ phiên bản hóa (các phiên bản khác nhau của DLL): hạn chế này gây ra rất nhiều khó khăn cho người quản trị hệ thống trong việc thay đổi (cập nhật) một chức năng nào đó cho một ứng dụng bất kì.

- Triển khai và cài đặt: khi chúng ta cài đặt các components (DLL) thì có rất nhiều thông tin được tạo ra và được ghi vào Registry và khi các ứng dụng thay đổi các tính năng thì các DLL này cần phải được gỡ bỏ. Khi DLL được gỡ bỏ thì chúng ta phải xác định được các thông tin tương ứng cần phải gỡ bỏ trong Registry và điều này gây ra rất nhiều khó khăn.

❖ Tất cả những khó khăn (hạn chế) trên của DLL được các những người quản trị hệ thống gọi là DLL Hell (địa ngục DLL). Và để giải quyết các vấn đề liên quan đến DLL Hell thì Assembly ra đời.

5.8.2 Assembly Là Gì ?

- Assembly là một khối mã được sử dụng để giải quyết vấn đề phiên bản hóa và các vấn đề khi triển khai các DLL.
- Một Assembly là một tập hợp các thông tin được yêu cầu bởi trình quản lý code để thực hiện các ứng dụng.
- Một Assembly là một tập hợp các đoạn code được sử dụng để tạo nên các ứng dụng.
- Một Assembly là một cách thức tổ chức code dưới dạng vật lý (được lưu trữ trên đĩa).
- **Có 2 loại Assembly**:
  - ***Private Assembly***: là một Assembly mà được triển khai cùng với một ứng dụng nào đó và chỉ được sử dụng bởi ứng dụng đó mà thôi chứ ko thể chia sẻ cho các ứng dụng khác. Private Assembly thường nằm trong thư mục cài của ứng dụng hoặc nằm trong thư mục bin của ứng dụng.
  - ***Shared Assembly***: là một assembly được cài đặt (triển khai) tại một vị trí được chia sẻ nào trên máy tính và có thể được sử dụng bởi tất cả các ứng dụng khác. Nơi lưu trữ các shared assembly là một thư mục[ổ cài Window]\WINDOWS[hoặc WINNT tùy vào hệ điều hành]\assembly.

5.8.3. Các Thuộc Tính Của Assembly

Trong nền tảng .NET thì một Assembly là một đơn vị code có khả năng sử dụng lại, có khả năng phiên bản hóa và được triển khai. Một Assembly có các thuộc tính:

- Các Assembly có thể được thực hiện như là một file .exe hoặc .dll. Nói cách khác Assembly có thể tồn tại dưới dạng file .exe hoặc .dll.
- Các Assembly có thể được chia sẻ, sử dụng bởi các ứng dụng khác nhau bằng cách lưu trữ trong một vùng gọi là Global Assembly Cache (GAC).
- Các Assembly cần phải có tên nổi bật (strong-named) trước khi được đặt vào trong GAC.
- Các Assembly chỉ được nạp vào bộ nhớ của các ứng dụng khi nó được yêu cầu.
- Các thông tin về Assembly có thể đạt được sử dụng cơ chế Reflection (ánh xạ
- ngược).
- Nhiều phiên bản của Assembly có thể được sử dụng bởi một ứng dụng.
- Các assembly có thể lưu trữ một hoặc nhiều module.

5.8.4 Cấu Trúc Của Assembly

Mỗi Assembly bao gồm 4 phần:

- Assembly MetaData (siêu dữ liệu về Assembly): là một tập hợp các thông tin tồn tại dưới dạng nhị phân trong file .exe hoặc .dll. Phần này có thể lưu trữ bất kì một loại thông tin nào. VD: tên tác giả, tên tổ chức, ngày tạo, số phiên bản của Assembly…
- Type MetaData (siêu dữ liệu về các kiểu): phần này lưu trữ các thông tin dữ liệu và các kiểu (class, interface, struct, enum…) của nó. Các thông tin về các phương thức (methods), các trường dữ liệu (fields), các thuộc tính (properties), các hàm khởi tạo (constructors) …
- MSIL Code (Microsoft Intermediate Language – ngôn ngữ trung gian của Microsoft): phần này lưu trữ các đoạn mã IL được biên dịch từ mã nguồn, các code này sau đó sẽ được thông dịch bởi CLR và biến đổi thành mã máy để thực hiện vào thời điểm runtime.
- Resources (các tài nguyên): phần này sẽ lưu trữ các thông tin tài nguyên được sử dụng trong các ứng dụng. Các thông tin có thể là các hình ảnh, các đoạn văn bản, các file XML hoặc các kiểu dữ liệu khác. Phần này có thể được lưu trữ trong cùng một file assembly hoặc được lưu trữ trong một file tách biệt.

5.8.5 Các Bước Tạo Một Assembly

Để tạo một Assembly thì chúng ta có thể sử dụng 2 cách:

- Cách 1: sử dụng Visual Stutio .NET 2010.
- Cách 2: sử dụng trình biên dịch (Csharp Compilier) csc.exe.

❖ **Cách 1:**

- Sử dụng một trình soạn thảo code (Code Editor) để viết code cho Assembly, lưu với tên FileName.cs (nếu dùng ngôn ngữ C#)
- Biên dịch mã nguồn thành Assembly:

Mở Command Prompt của VS 2010, chúng ta gõ lệnh sau:

csc.exe /target:library /out:[tên assembly cần tạo]

Ví dụ :

csc.exe /target:library /out:MyLib.dll FileName.cs

❖ **Cách 2**:

Mở VS.NET 2010 , vào menu File | New Project, trên hộp thoại New Project chọn Class Library. Viết lệnh cho Assembly và nhấn Shift+F6 để biên dịch thành Assembly (.dll).