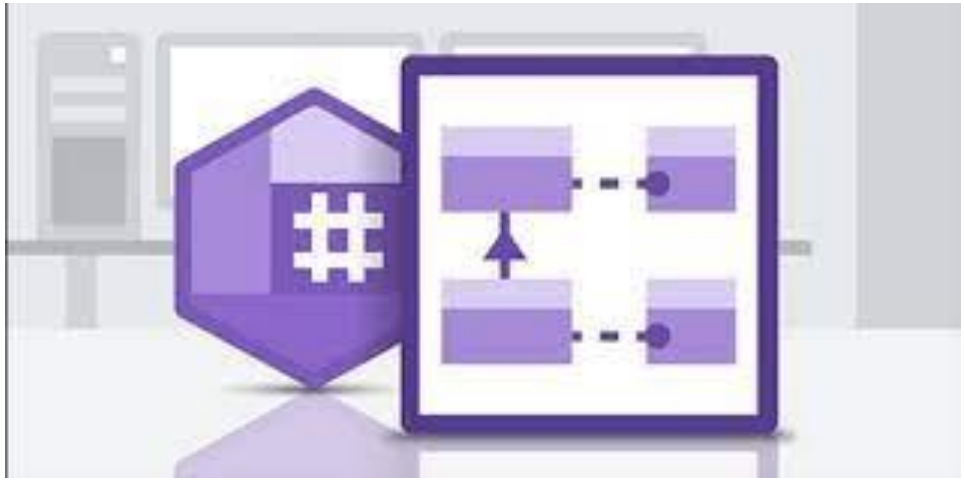
C# 13 – Generic và Iterator

Giảng viên: **ThS. Lê Thiện Nhật Quang**

Email: quangltn.dotnet.vn@gmail.com

Website: http://dotnet.edu.vn

Điện thoại: **0868.917.786**



MỤC TIÊU

- Tìm hiểu Generic
- Giải thích khởi tạo và sử dụng generic
- Tìm hiểu Iterator

1.1. GIỚI THIỆU

Generic là kiểu đại diện, nó cho phép tạo mã nguồn code không phụ thuộc vào kiểu dữ liệu cụ thể, chỉ khi code thực thi thì kiểu cụ thể mới xác định.

Trước đây, viết code trên những kiểu dữ liệu cụ thể như int, float, double ... hay các class do người dùng định nghĩa, tuy nhiên có những giải thuật giống nhau trên những kiểu dữ liệu khác nhau, để tránh việc viết nhiều lần code lặp lại thì lúc này áp dụng Generic - kiểu đại diện để xây dựng phương thức hoặc lớp.

Ví dụ cần chuyển đổi giá trị lưu trong hai biến kiểu int, có thể xây dựng phương thức theo cách thông thường như sau:

```
// phương thức này tráo đổi giá trị giữa hai biến kiểu int
public static void Swap(ref int a, ref int b) {
    int c = a;
    a = b;
    b = c;
}
```

Tuy nhiên, nếu cần tráo đổi giá trị giữa hai biến kiểu string thì phương thức trên không dùng được, bạn lại cần xây dựng một phương thức khác dành cho tham số kiểu string

```
// phương thức này tráo đổi giá trị giữa hai biến kiểu string
public static void Swap(ref string a, ref string b) {
    string c = a;
    a = b;
    b = c;
}
```

1.2. CÁC NAMESPACE, CLASS VÀ INTERFACE GENERIC

Có một số namespace trong .NET Framework rất thuận tiện cho việc tạo và sử dụng các generic.

Ví dụ như namespace **System.Collections.ObjectModel** cho phép tạo các tập hợp generic động và chỉ đọc, namespace **System.Collections.Generic** bao gồm các lớp và interface cho phép định nghĩa các tập hợp generic tùy chỉnh, ...

1.3. CÁC LỚP GENERIC

Namespace có chứa các lớp cho phép bạn tạo các tập hợp với kiểu an toàn; bảng dưới đây sẽ liệt kê các lớp được sử dụng phổ biến của namespace này:

Lớp	Mô tả
Comparer	Là một lớp trừu tượng cho phép ta tạo một tập hợp generic bằng cách thực thi các chức năng của interface IComparer
Dictionary.KeyCollection	Bao gồm các key của thể hiện của lớp Dictionary
Dictionary.ValueCollection	Bao gồm các value của thể hiện của lớp Dictionary
EqualityComparer	Là một lớp trừu tượng cho phép ta tạo một tập hợp generic bằng cách thực thi các chức năng của interface IEqualityComparer

1.4. CÁC INTERFACE GENERIC

Namespace **System.Collections.Generic** cũng chứa các interface cho phép bạn tạo các tập hợp với kiểu an toàn; bảng dưới đây sẽ liệt kê các lớp được sử dụng phổ biến của namespace này:

Interface	Mô tả
IComparer	Định nghĩa phương thức generic Comparer() dùng để so sánh các giá trị trong một tập hợp
INumerable	Định nghĩa phương thức GetEnumerator() dùng để lặp (iterate) trên một tập hợp
IEqualityComparer	Bao gồm các phương thức cho phép kiểm tra sự tương xứng giữa hai đối tượng bất kỳ

1.5. SYSTEM.COLLECTIONS.OBJECTMODEL

Namespace này bao gồm các lớp có thể được dùng để tạo các tập hợp generic tùy chỉnh; bảng dưới đây sẽ liệt kê một số lớp của namespace này:

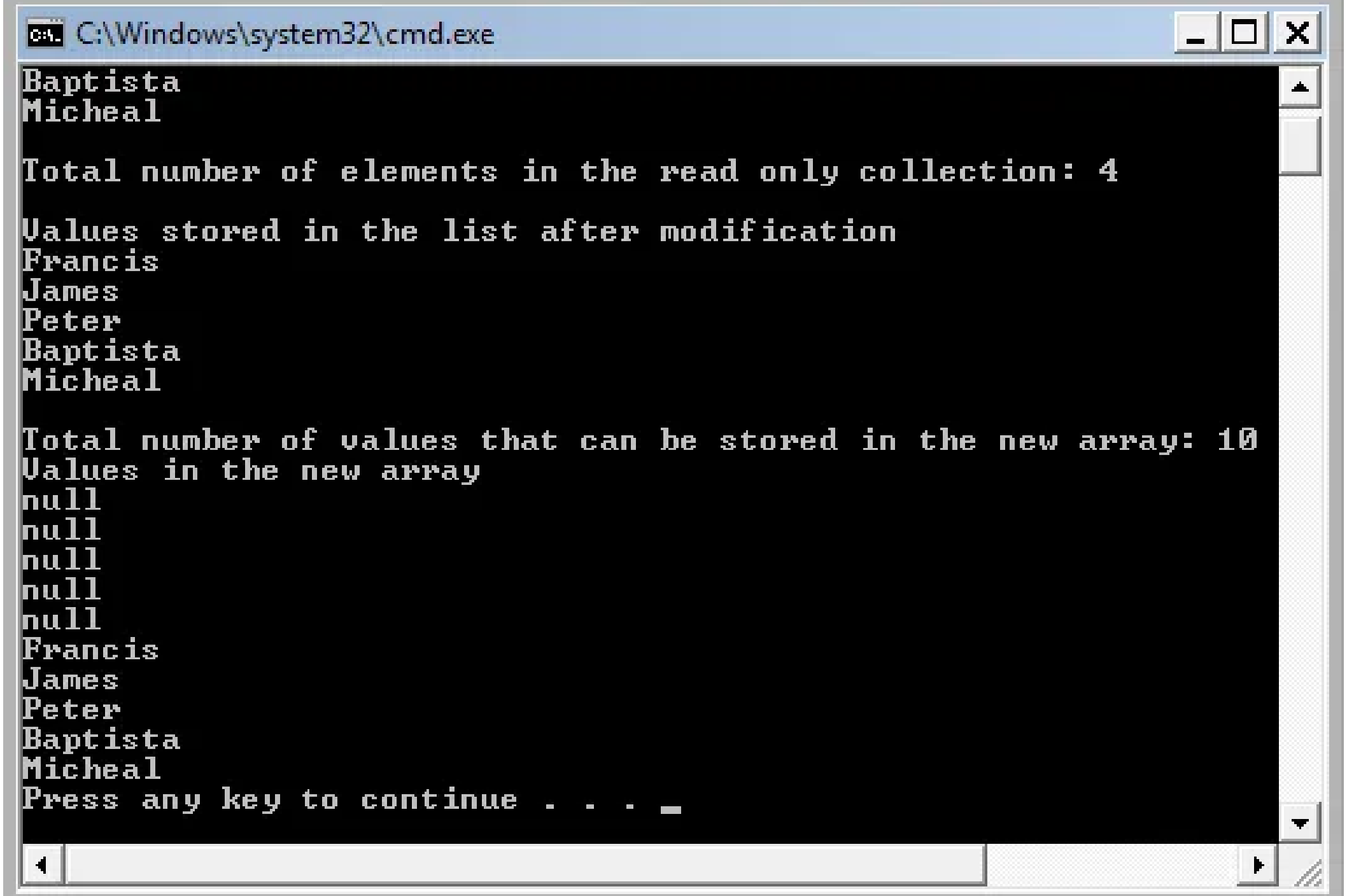
Lớp	Mô tả
Collection<>	Cung cấp lớp cơ sở cho các tập hợp generic
KeyedCollection<>	Cung cấp một lớp trừu tượng cho một tập hợp với key và value tương ứng
ReadOnlyCollection<>	Là một lớp generic cơ sở chỉ đọc dùng để ngăn ngừa sự thay đổi (modify) của tập hợp

1.5. VÍ DỤ

```
static void Main(string[] args)
{
  List<string> objList = new List<string>();
  objList.Add("Francis");
  objList.Add("James");
  objList.Add("Baptista");
  objList.Add("Micheal");
  ReadOnlyCollection<string> objReadOnly = new ReadOnlyCollection
    <string>(objList);
  Console.WriteLine("Values stored in the read only collection");
  foreach (string str in objReadOnly)
  {
    Console.WriteLine(str);
  }
  Console.WriteLine();
  Console.WriteLine("Total number of elements in the read only collection: " + objReadOnly.Count);
  if (objList.Contains("Francis"))
  {
    objList.Insert(2, "Peter");
  }
```

```
Console.WriteLine("\nValues stored in the list after modification");
foreach (string str in objReadOnly)
{
  Console.WriteLine(str);
}
string[] array = new string[objReadOnly.Count * 2];
objReadOnly.CopyTo(array, 5);
Console.WriteLine("\nTotal number of values that can be stored in the new array: " + array.Length);
Console.WriteLine("Values in the new array");
foreach (string str in array)
{
  if (str == null)
  {
    Console.WriteLine("null");
  }
  else
  {
    Console.WriteLine(str);
  }
}
```

- Trong ví dụ trên, phương thức Main() của lớp ReadOnly tạo thể hiện có tên objList với một tham số kiểu string của lớp generic List
- Phương thức Add() sẽ thêm các phần tử vào thể hiện objList
- Phương thức Main() cũng tạo thể hiện có tên objReadOnly của lớp ReadOnlyCollection với một tham số kiểu string và các phần tử của thể hiện objList của lớp List sẽ được copy tới thể hiện này
- Phương thức Contains() sẽ kiểm tra xem thể hiện objList của lớp List có chứa phần tử với giá trị "Francis“ hay không, nếu chứa thì sẽ thêm một phần tử mới có giá trị "Peter“ vào objList tại vị trí có chỉ số là 2
- Phương thức Main() cũng tạo một biến mảng có tên array và có kích thước gấp đôi kích thước của thể hiện objReadOnly
- Phương thức CopyTo() sẽ tiến hành copy các phần tử của thể hiện objReadOnly vào mảng array và đặt các phần tử copy được vào vị trí có chỉ số bằng 5 trở đi.
- Kết quả của ví dụ trên được thể hiện ở hình dưới đây:



1.6. TẠO KIỂU GENERIC

- Việc khai báo một generic luôn luôn phải có ít nhất một tham số kiểu và tham số kiểu sẽ được dùng để làm nơi giữ chỗ cho kiểu dữ liệu cụ thể sau này; kiểu dữ liệu cụ thể chỉ có thể được xác định khi kiểu generic được tham chiếu tới hoặc được khởi tạo trong chương trình.
- Quá trình tạo một kiểu generic bắt đầu bằng việc định nghĩa kiểu generic với các tham số kiểu đi kèm; việc định nghĩa này đóng vai trò như là một bản thiết kế. Sau đó, các kiểu generic sẽ được tạo từ định nghĩa bằng cách xác định kiểu thực sự và sẽ thay thế cho các tham số kiểu hay các nơi giữ chỗ.

1.7. ƯU ĐIỂM CỦA GENERIC

Generic đảm bảo tính an toàn kiểu trong thời gian biên dịch chương trình. Generic cho phép ta dùng lại được các mã lệnh một cách an toàn mà không cần phải gán hay boxing. Mỗi định nghĩa kiểu generic có thể được sử dụng lại với các kiểu khác nhau nhưng tại một thời điểm chỉ sử dụng được một kiểu; cùng với khả năng sử dụng lại thì generic còn có một số khả năng nữa như sau:

- Tăng hiệu năng của chương trình, bởi vì chương trình sẽ sử dụng ít bộ nhớ hơn do không cần phải sử dụng các biện pháp gán hay boxing.
- Đảm bảo được mô hình lập trình định kiểu mạnh mẽ.
- Giảm được lỗi run-time có thể gây ra bởi không cần sử dụng đến gán và boxing.

2.1. LỚP GENERIC

Các lớp generic định nghĩa những chức năng mà có thể được sử dụng cho bất kỳ loại dữ liệu nào, chúng được khai báo bằng cách sử dụng từ khóa class và theo sau là một tên lớp, phía sau đó là cặp ngoặc nhọn và bên trong có chứa tham số kiểu. Khi khai báo lớp generic ta nên đưa ra giới hạn hay ràng buộc cho tham số kiểu bằng cách sử dụng từ khóa where . Lưu ý rằng đây chỉ là một tuỳ chọn, vì vậy, khi tạo loại lớp này bạn phải khái quát hoá các kiểu dữ liệu vào phần tham số kiểu và tuỳ chọn này sẽ quyết định ràng buộc để áp dụng trên tham số kiểu.

Cú pháp:

```
Bổ_từ_truy_cập class Tên_lớp<Danh_sách_tham_số_kiểu> [where Mệnh_đề_ràng_buộc_tham_số_kiểu]
```

Bổ_từ_truy_cập: dùng để xác định tầm vực của lớp generic (đây là một tuỳ chọn).

Danh_sách_tham_số_kiểu: được dùng tạo nơi giữ chỗ cho kiểu dữ liệu thực sự sau này.

```
class ChungChung<T>
{
  T[] giaTri;
  int _boDem = 0;
  public ChungChung(int max)
  {
    giaTri = new T[max];
  }
  public void Them(T val)
  {
    if (_boDem < giaTri.Length)
    {
      giaTri[_boDem] = val;
      _boDem++;
    }
  }
  public void HienThi()
  {
    Console.WriteLine("Kieu cua lop co cau truc la: " + typeof(T));
    Console.WriteLine("Cac gia tri luu trong doi tuong cua lop co cau truc la:");
    for (int i = 0; i < giaTri.Length; i++)
    {
      Console.WriteLine(giaTri[i]);
    }
  }
}
class SinhVien
{
  static void Main(string[] args)
  {
    ChungChung<string> objCC1 = new ChungChung<string>(3);
    objCC1.Them("Phuong");
    objCC1.Them("Binh");
    objCC1.HienThi();
    ChungChung<int> objCC2 = new ChungChung<int>(2);
    objCC2.Them(325);
    objCC2.Them(57);
    objCC2.HienThi();
  }
}
```

2.1. LỚP GENERIC – PHÂN TÍCH (3)

- Trước tiên định nghĩa một lớp generic có tên ChungChung và có tham số kiểu là T
- Bên trong lớp này có một hàm tạo với một tham số kiểu int
- Phương thức Them() có một tham số cùng kiểu với lớp generic;
- Phương thức Display() hiển thị kiểu giá trị được xác định thông qua tham số kiểu và các giá trị được cung cấp bởi người dùng thông qua các đối tượng;
- Phương thức Main() của lớp SinhVien tạo một thể hiện có tên objGeneral của lớp ChungChung bằng cách cung cấp giá trị tham số kiểu string và tổng số các giá trị được lưu trữ là 3;
- Thể hiện objCC1 gọi phương thức Them() và truyền đi tên của các sinh viên và các tên sinh viên sẽ được hiển thị thông qua lời gọi phương thức HienThi().
- Sau đó, thể hiện objCC2 được tạo với tham số kiểu là int và khởi tạo tổng số giá trị lưu trữ là 2.
- Đến đây bạn sẽ thấy một điều rằng ta không cần phải thay đổi mã lệnh để phù hợp với kiểu dữ liệu mới khi ta sử dụng lớp generic.

2.2. RÀNG BUỘC CHO CÁC THAM SỐ KIỂU

Ta có thể sử dụng các ràng buộc cho kiểu tham số khi khai báo kiểu generic. Mỗi ràng buộc sẽ là một giới hạn áp đặt cho kiểu dữ liệu của tham số kiểu. Ràng buộc được tạo thông qua sử dụng từ khoá where và việc thiết lập ràng buộc cho tham số kiểu sẽ đảm bảo tính nhất quán và độ tin cậy của dữ liệu trong tập hợp.

Bảng dưới đây sẽ liệt kê các loại ràng buộc ta có thể được áp dụng cho tham số kiểu (giả sử T là tên của tham số kiểu):

Ràng buộc	Mô tả
T : struct	Tham số kiểu phải là một kiểu giá trị (value type) và chấp nhận giá trị null
T : class	Tham số kiểu phải là một kiểu tham chiếu (reference type), ví dụ như class, interface, delegate
T : new()	Tham số kiểu phải bao gồm một hàm tạo không tham số và có bổ từ truy cập là public
T : Tên_lớp	Tham số kiểu phải là một lớp cơ sở hoặc dược thừa kế từ lớp cơ sở
T : Tên_interface	Tham số kiểu phải là một interface hoặc được thừa kế từ interface

```
class NhanVien1
{
  string _tenNV;
  int _idNV;
  public NhanVien1(string ten, int id)
  {
    _tenNV = ten;
    _idNV = id;
  }
  public string Ten
  {
    get
    {
      return _tenNV;
    }
  }
  public int ID
  {
    get
    {
      return _idNV;
    }
  }
}
```

```
class Demo<T> where T : NhanVien1
  {
  T[] _ten = new T[3];
  int _boDem = 0;
public void Them(T val)
  {
  _ten[_boDem] = val;
  _boDem++;
  }
public void HienThi()
  {
  for (int i = 0; i < _boDem; i++)
  {
  Console.WriteLine(_ten[i].Ten + ", " + _ten[i].ID);
  }
  }
  }
```

```
class TestDemo
{
  static void Main(string[] args)
  {
    Demo<NhanVien1> objDemo = new Demo<NhanVien1>();
    objDemo.Them(new NhanVien1("Lan", 123));
    objDemo.Them(new NhanVien1("Long", 456));
    objDemo.Them(new NhanVien1("Minh", 789));
    objDemo.HienThi();
  }
}
```

- Trong ví dụ trên, lớp Demo được tạo với tham số kiểu là T và ràng buộc tham số kiểu cho T có kiểu class là lớp Employee ;
- Lớp Demo tạo một biến mảng kiểu T, tức là kiểu NhanVien;
- Phương thức Them() của lớp Demo có một tham số cũng là kiểu T và nó sẽ nhận lại giá trị từ lời gọi phương thức Them() ở phương thức Main(), điều này có nghĩa tham số của phương thức Them() ở Main() phải có kiểu NhanVien và hàm tạo của Demo được gọi ngay trong tham số của phương thức Them() của Main()
- Dưới đây là kết quả của ví dụ:

```
Lan, 123
Long, 456
Minh, 789
```

Chú ý: Khi ta sử dụng một kiểu cụ thể nào đó làm ràng buộc thì kiểu đó cần phải có mức truy cập cao hơn kiểu generic mã sẽ được áp dụng ràng buộc.

2.3. THỪA KẾ LỚP GENERIC

Lớp generic có thể được thừa kế giống như các lớp khác trong C#, điều này có nghĩa rằng lớp generic có thể đóng vai trò là lớp cơ sở hoặc lớp dẫn xuất.

Ta có quyền thừa kế các tham số kiểu từ lớp cha là lớp generic, điều này sẽ tránh được phải truyền đối số với kiểu dữ liệu cụ thể cho tham số. Lưu ý là nếu lớp dẫn xuất là lớp không generic thì trong lớp dẫn xuất ta phải cung cấp kiểu dữ liệu của tham số để thế vào tham số kiểu generic của lớp cơ sở, còn nếu lớp dẫn xuất là lớp generic thì ràng buộc áp đặt tại lớp cơ sở phải được đưa vào trong lớp generic dẫn xuất đó.

2.3. THỪA KẾ LỚP GENERIC (2)

Ví dụ khai báo một lớp generic thừa kế từ lớp generic khác:

```
// Generic thừa kế Generic
public class SinhVien<T>
{
}
public class DiemSo<T> : SinhVien<T>
{
}
```

Ví dụ khai báo một lớp không phải generic thừa kế từ lớp generic:

```
// Non-Generic thừa kế Generic
public class Student<T>
{
}
public class Mark : Student<string>
{
}
```

2.4. PHƯƠNG THỨC GENERIC

Chúng xử lý các dữ liệu trong đó kiểu của chúng chỉ được biết khi truy cập vào các biến chứa những dữ liệu đó, chúng được khai báo với các tham số kiểu nằm trong cặp ngoặc nhọn.

Việc định nghĩa các phương thức như vậy sẽ giúp ta có thể truyền các kiểu dữ liệu khác nhau mỗi lần gọi. Ta có thể khai báo phương thức generic trong một lớp generic hoặc non-generic, và khi chúng được khai báo bên trong lớp generic thì phần thân của chúng sẽ tham chiếu được đến cả các tham số kiểu của chúng lẫn của lớp chứa chúng.

2.4. PHƯƠNG THỨC GENERIC (2)

Phương thức generic có thể được khai báo cùng với những từ khoá sau:

- **virtual**: Từ khoá này sẽ giúp phương thức được ghi đè trong lớp dẫn xuất.
- **override**: Từ khoá này sẽ giúp phương thức ghi đè phương thức của lớp cơ sở.
- **abstract:** Phương thức chứa từ khoá này chỉ có thể được khai báo mà không được định nghĩa (không có phần thân).

Cú pháp:

```
Bổ_từ_truy_cập Kiểu_trả_về Tên_phương_thức<Danh_sách_tham_số_kiểu>(Danh_sách_tham_số)
```

2.4. PHƯƠNG THỨC GENERIC (3)

```
class HoanViSo
{
  static void HoanVi<T>(ref T so1, ref T so2)
  {
    T tam = so1;
    so1 = so2;
    so2 = tam;
  }
  static void Main(string[] args)
  {
    int so1 = 67;
    int so2 = 89;
    Console.WriteLine("Cac gia tri truoc khi hoan vi, so1 = " + so1 + ", so2 = " + so2);
    HoanVi<int>(ref so1, ref so2);
    Console.WriteLine("Sau khi hoan vi, so1 = " + so1 + ", so2 = " + so2);
  }
}
```

2.4. PHƯƠNG THỨC GENERIC (4)

- Giải thích ví dụ: Lớp HoanViSo có phương thức HoanVi() với tham số kiểu là T đặt trong cặp ngoặc nhọn và có hai đối số cũng có kiểu là T.
- Trong phương thức HoanVi() tạo một biến tên tam cũng có kiểu T và được gán giá trị của biến so1;
- Phương thức Main() khai báo hai biến so1 và so2 có cùng kiểu int và khởi tạo giá trị so1 = 67 và so2 = 89, sau đó gọi đến phương thức HoanVi() và truyền đi kiểu int trong cặp ngoặc nhọn và hai tham chiếu của hai biến so1 và so2

2.5. INTERFACE GENERIC

Loại interface này rất hữu ích đối với các lớp hoặc tập lớp generic trong việc thể hiện các mục trong tập hợp. Ta có thể sử dụng các lớp và interface generic để tránh các hoạt động boxing và unboxing trên các kiểu dữ liệu. Các lớp generic có thể thực thi các interface generic bằng cách truyền các tham số cụ thể tới interface. Interface generic cũng có thể thừa kế.

Cú pháp khai báo interface generic:

```
interface Tên_interface <Danh_sách_tham_số_kiểu> [where Mệnh_đề ràng_buộc_tham_số_kiểu]
{
}
```

2.5. INTERFACE GENERIC (2)

```
interface IToanHoc<T>
{
  T Cong(T valOne, T valTwo);
  T Tru(T valOne, T valTwo);
}
```

```
class ConSo : IToanHoc<int>
{
  public int Cong(int so1, int so2)
  {
    return so1 + so2;
  }
  public int Tru(int so1, int so2)
  {
    if (so1 > so2)
    {
      return (so1 - so2);
    }
    else
    {
      return (so2 - so1);
    }
  }
```

```
  static void Main(string[] args)
  {
    int so1 = 23;
    int so2 = 45;
    ConSo objCS = new ConSo();
    Console.Write("Cong {0} voi {1} duoc: ", so1, so2);
    Console.WriteLine(objCS.Cong(so1, so2));
    Console.Write("Tru {0} cho {1} duoc: ", so1, so2);
    Console.WriteLine(objCS.Tru(so1, so2));
  }
```

2.5. INTERFACE GENERIC (3)

Giải thích ví dụ: Trong interface generic IToanHoc với tham số kiểu là T khai báo hai phương thức Cong() và Tru() , mỗi phương thức cũng đều có hai tham số có kiểu là T. Lớp ConSo thực thi IToanHoc bằng cách cung cấp kiểu int trong cặp ngoặc nhọn và thực thi hai phương thức của interface này. Phương thức Main() tạo một thể hiện của lớp ConSo và gọi hai phương thức Cong() và Tru() để thực hiện các phép tính cộng và trừ. Kết quả của ví dụ như sau:

```
Cong 23 voi 45 duoc: 68
Tru 23 voi 45 duoc: 22
```

2.6. RÀNG BUỘC THAM SỐ KIỂU LÀ INTERFACE GENERIC

Ta có thể định nghĩa interface như là một tham số kiểu và điều này sẽ cho phép ta sử dụng các thành phần của interface trong một lớp generic. Mặt khác thì điều này cũng đảm bảo rằng chỉ có kiểu là interface mới được sử dụng trong lớp. Một lớp có quyền có nhiều tham số kiểu là interface. Ví dụ:

```
interface IChiTiet
{
  void ChiTiet();
}

class SinhVien : IChiTiet
{
  string _ten;
  int _id;
  public SinhVien(string ten, int id)
  {
    _ten = ten;
    _id = id;
  }
  public void ChiTiet()
  {
    Console.WriteLine(_id + "\t" + _ten);
  }
}
```

```
class ThuNghiem<T> where T : IChiTiet
{
  T[] _giaTri = new T[3];
  int _boDem = 0;
  public void Them(T val)
  {
    _giaTri[_boDem] = val;
    _boDem++;
  }
  public void HienThi()
  {
    for (int i = 0; i < 3; i++)
    {
      _giaTri[i].ChiTiet();
    }
  }
}
```

```
class ThuNghiem<T> where T : IChiTiet
{
  T[] _giaTri = new T[3];
  int _boDem = 0;
  public void Them(T val)
  {
    _giaTri[_boDem] = val;
    _boDem++;
  }
  public void HienThi()
  {
    for (int i = 0; i < 3; i++)
    {
      _giaTri[i].ChiTiet();
    }
  }
}
```

2.6. RÀNG BUỘC THAM SỐ KIỂU LÀ INTERFACE GENERIC (3)

Giải thích ví dụ: Trong interface IChiTiet khai báo phương thức ChiTiet(). Lớp SinhVien thực thi interface IChiTiet. Lớp ThuNghiem được tạo với tham số kiểu là T với ràng buộc tham số kiểu là IChiTiet, điều này có nghĩa rằng tham số kiểu chỉ có thể là IChiTiet. Phương thức Main() tạo một thể hiện của ThuNghiem bằng cách truyền giá trị của tham số kiểu là SinhVien vì lớp này thực thi interface IChiTiet

2.7. ĐẶC ĐIỂM GENERIC

Giúp định nghĩa thao tác dữ liệu với kiểu dữ liệu chung nhất nhằm hạn chế viết code và tái sử dụng.

Ứng dụng phổ biến nhất của Generic là tạo ra các Generic Collections.

• Các Collections phổ biến, giá trị lưu trữ bên trong đều là object.

• Điều này gây rất nhiều khó khăn nếu như ta muốn quản lý 1 danh sách có cùng kiểu.

Vì object có thể chứa được mọi kiểu dữ liệu nên khó kiểm soát rằng việc thêm phần tử có phải cùng kiểu dữ liệu mong muốn hay không.

• Từ đó Generic Collections ra đời để giúp vừa có thể sử dụng được các Collections vừa có thể hạn chế lỗi xảy ra trong quá trình thực thi.

Ngoài ra, Generic còn giúp hạn chế truy cập nếu như không truyền đúng kiểu dữ liệu.

2.8. MỘT SỐ LOẠI GENERIC THÔNG DỤNG

Các Generic Collections đều được xây dựng bắt nguồn từ 1 Collections nào đó có sẵn. Vì thế với mỗi Collections đã học sẽ có một Generic tương ứng.

2.8. MỘT SỐ LOẠI GENERIC THÔNG DỤNG (2)

LỚP	MÔ TẢ
List<T>	Là một Collections giúp lưu trữ các phần tử liên tiếp (giống mảng) nhưng có khả năng tự mở rộng kích thước. Generic Collections này là sự thay thế cho ArrayList.
Dictionary<Tkey, TValue>	Lớp lưu trữ dữ liệu dưới dạng cặp **Key – Value**. Khi đó ta sẽ truy xuất các phần tử trong danh sách này thông qua Key (thay vì thông qua chỉ số phần tử như mảng bình thường). Generic Collections này là sự thay thế cho Hashtable.
SortedDictionary<Tkey, TValue>	Là sự kết hợp giữa List và Dictionary. Tức là dữ liệu sẽ lưu dưới dạng **Key – Value**. Ta có thể truy xuất các phần tử trong danh sách thông qua **Key** hoặc thông qua chỉ số phần tử. Đặc biệt là các phần tử trong danh sách này luôn được sắp xếp theo giá trị của **Key**. Generic Collections này là sự thay thế cho SortedList.
Stack<T>	Lớp cho phép lưu trữ và thao tác dữ liệu theo cấu trúc **LIFO (Last In First Out)**. Generic Collections này là sự thay thế cho Stack.
Queue<T>	Lớp cho phép lưu trữ và thao tác dữ liệu theo cấu trúc **FIFO (First In First Out)**. Generic Collections là sự thay thế cho Queue .

2.9. KHAI BÁO PHƯƠNG THỨC GENERIC

Khai báo phương thức có sử dụng kiểu Generic, đâu là kiểu dữ liệu cụ thể thì thay bằng tên kiểu Generic, tên này là do người dùng đặt một cách thống nhất tùy chọn như A, B, T, ...

Trong đó sau phần tên hàm phải liệt kê ra tên những kiểu Generic mà người dùng sẽ sử dụng cho hàm.

2.9. KHAI BÁO PHƯƠNG THỨC GENERIC (2)

Một khai báo tổng quát như sạng sau:

```
X MyFunction<X, Y>(X x, Y y)
{
    return x;
}
```

Khai báo phương thức có tên MyFunction, sau tên này có ký hiệu <X, Y> có nghĩa phương thức này có sử dụng hai kiểu là kiểu X và kiểu Y (tất nhiên nó chưa cụ thể là kiểu gì, nó chỉ cụ thể khi phương thức được gọi).

Khi đã có kiểu Generic rồi thì dùng kiểu này cho các thành phần của phương thức - như kiểu trả về là X, tham số là kiểu X và Y, trong thân phương thức tương tự có thể khai báo sử dụng kiểu X và kiểu Y.

2.9. KHAI BÁO PHƯƠNG THỨC GENERIC (3)

Lúc này khi gọi, chỉ cần điền tên X và Y theo kiểu cụ thể trong ký hiệu < ... >:

```
int a = 1;
string b = 2;
int rs = MyFunction<int, string>(a, b);  // Phương thức chạy với vai trò X là kiểu int, Y là kiểu string.

double c = 2;
MyFunction<double, int>(c, a);           // Phương thức chạy với X là kiểu double, Y kiểu int
```

2.10. VÍ DỤ GENERIC

Áp dụng xây dựng hàm Swap ở trên:

```
static void Swap<T>(ref T a, ref T b)
{
    T c = a;
    a = b;
    b = c;
}
```

Khi sử dụng, gọi phương thức chỉ việc thay chữ T bằng kiểu cụ thể muốn áp dụng

```
public static void TestSwap()
{
  int a = 1;
  int b = 2;
  Swap<int>(ref a, ref b);                        // Hàm trên kiểu T = int
  System.Console.WriteLine($"a = {a}; b = {b}"); // a = 2; b = 1

  string a1 = "A";
  string b1 = "B";
  Swap<string>(ref a1, ref b1); // Hàm trên kiểu T = string
  System.Console.WriteLine($"a1 = {a1}; b1 = {b1}"); // a1 = B; b1 = C
}
```

3.1. GIỚI THIỆU ITERATOR

- Dùng để duyệt qua danh sách các giá trị của tập hợp. Iterator là một khối mã lệnh trong đó thường dùng vòng lặp foreach để tham chiếu liên tiếp tới tập hợp các giá trị. Ví dụ như nếu muốn sắp xếp các giá trị của tập hợp ta cần sử dụng iterator để lấy lần lượt từng giá trị một rồi đem so sánh.
- Iterator không phải là một thành phần dữ liệu, nó chỉ là cách truy cập các thành phần, chẳng hạn như là một phương thức (bộ truy cập get chẳng hạn) hoặc một toán tử cho phép ta điều hướng thông qua các giá trị của tập hợp.
- Iterator sẽ xác định cách mà các giá trị được tạo ra khi vòng lặp foreach truy cập vào các phần tử của tập hợp, và iterator sẽ lưu vết của các phần tử đó, điều này sẽ giúp tăng tốc độ truy cập các giá trị của các phần tử.

3.2. ƯU ĐIỂM ITERATOR

- Iterator cung cấp một cách đơn giản và nhanh hơn cách thức duyệt (lặp) trên các giá trị của tập hợp.
- Iterator sẽ giảm đi tính phức tạp bằng việc cung cấp bộ liệt kê cho tập hợp.
- Iterator có thể trả về một số lượng lớn các giá trị.
- Iterator có thể được dùng để định lượng các giá trị và chỉ trả lại những giá trị cần thiết.
- Iterator có thể trả lại những giá trị mà không hề dùng đến bộ nhớ bằng cách tham chiếu đến từng giá trị trong tập hợp.

3.3. THỰC THI ITERATOR

- Iterator có thể được tạo bằng cách thực thi phương thức GetEnumerator() của interface IEnumerator, phương thức này sẽ trả về một tham chiếu của interface này.
- Khối iterator sẽ sử dụng từ khoá yield để cung cấp các giá trị cho thể hiện của bộ liệt kê (enumerator) hoặc để kết thúc quá trình lặp, cụ thể, câu lệnh yield return sẽ trả về các giá trị, còn câu lệnh yield break sẽ kết thúc tiến trình lặp. Khi gặp câu lệnh yield return thì vị trí hiện thời sẽ được lưu trữ và lần kế khi iterator được gọi thì việc thực thi sẽ bắt đầu từ vị trí đã được lưu trữ đó.

3.4. VÍ DỤ ITERATOR

```
class Department : IEnumerable
{
  string[] departmentNames = { "Marketing", "Tai chinh", "Cong nghe thong tin", "Hanh chinh nhan su" };
  public IEnumerator GetEnumerator()
  {
    for (int i = 0; i < departmentNames.Length; i++)
    {
      yield return departmentNames[i];
    }
  }

  static void Main(string[] args)
  {
    Department objDepartment = new Department();
    Console.WriteLine("Ten cua cac phong ban:");
    foreach (string str in objDepartment)
    {
      Console.WriteLine(str);
    }
  }
}
```

```
Ten cua cac phong ban:
Marketing
Tai chinh
Cong nghe thong tin
Hanh chinh nhan su
```

3.5. ITERATOR GENERIC

C# cho phép tạo iterator generic, công việc này được thực hiện bằng cách trả về một đối tượng có kiểu interface generic IEnumerator<T> hoặc IEnumerable<T>. Iterator generic có thể duyệt dữ liệu với kiểu bất kỳ. Ví dụ:

```
class GenericDepartment<T>
{
  T[] item;
  public GenericDepartment(T[] val)
  {
    item = val;
  }
  public IEnumerator<T> GetEnumerator()
  {
    foreach (T value in item)
    {
      yield return value;
    }
  }
}
```

3.5. ITERATOR GENERIC (2)

```
class GenericIterator
{
  static void Main(string[] args)
  {
    string[] departmentNames = { "Marketing", "Finance", "Information Technology", "Human Resources" };
    GenericDepartment<string> objGeneralName = new GenericDepartment<string>(departmentNames);
    foreach (string val in objGeneralName)
    {
      Console.Write(val + "\t");
    }
    int[] departmentID = { 101, 110, 210, 220 };
    GenericDepartment<int> objGeneralID = new GenericDepartment<int>(departmentID);
    Console.WriteLine();
    foreach (int val in objGeneralID)
    {
      Console.Write(val + "\t\t");
    }
    Console.WriteLine();
  }
}
```

3.5. ITERATOR GENERIC (3)

Trong ví dụ trên, lớp generic GenericDepartment được tạo với tham số kiểu là T, trong lớp có một biến mảng và một hàm tạo có tham số để khởi tạo giá trị cho các phần tử của mảng; trong lớp GenericDepartment còn có phương thức GetEnumerator() với kiểu trả về là IEnumerator, phương thức này trả lại các phần tử được lưu trong biến mảng thông qua câu lệnh yield return; phương thức Main() trong lớp GenericIterator tạo một thể hiện của lớp GenericDepartment để tham chiếu tới các tên phòng ban khác nhau của mảng; một đối tượng khác của lớp GenericDepartment cũng được tạo và nó tham chiếu tới các ID của các phòng ban của mảng.

3.6. ITERATOR ĐƯỢC ĐẶT TÊN

Iterator có thể được tạo bằng cách tạo một phương thức với tên riêng, phương thức này có kiểu trả về là interface IEnumerable, người ta gọi là iterator được đặt tên. Loại iterator này có thể chấp nhận các đối số mà có thể được dùng để quản lý điểm đầu và điểm cuối của vòng lặp foreach. Kỹ thuật này giúp ta lấy những giá trị theo ý muốn trong tập hợp.

Cú pháp:

```
Access_modifier  IEnumerable  Iterator_name(Parameter_list){}
```

3.7. VÍ DỤ ITERATOR ĐƯỢC ĐẶT TÊN

```
class NamedIterators
{
  string[] cars = { "Ferrari", "Mercedes", "BMW", "Toyota", "Nissan" };
  public IEnumerable GetCarNames()
  {
    for (int i = 0; i < cars.Length; i++)
    {
      yield return cars[i];
    }
  }
  static void Main(string[] args)
  {
    NamedIterators objIterator = new NamedIterators();
    foreach (string str in objIterator.GetCarNames())
    {
      Console.WriteLine(str);
    }
  }
}
```

3.7. VÍ DỤ ITERATOR ĐƯỢC ĐẶT TÊN (2)

Trong ví dụ trên, lớp NamedIterators bao gồm một biến mảng và phương thức GetCarNames() (đây chính là iterator được đặt tên) có kiểu trả về IEnumerable, vòng lặp for của phương thức duyệt các giá trị trong biến mảng; phương thức Main() tạo một thể hiện của lớp NamedIterators và thể hiện này được dùng trong vòng lặp foreach để hiển thị tên của các xe hơi trong biến mảng.

```
Ferrari
Mercedes
BMW
Toyota
Nissan
```