ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN
Vietnam - Korea University of Information and Communication Technology

Chapter 4

Collections and Generics

ĐẠI HỌC ĐÀ NẴNG

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN

Vietnam - Korea University of Information and Communication Technology

Generics in C#

http://vku.udn.vn/

The Issue of Performance

- **A primary limitation of collections is the absence of effective type checking. This means that we can put any object in a collection because all classes in the C# extend from the object base class and this compromises type safety in C# language**

- **In addition, using collections involves a significant performance overhead in the form of implicit and explicit type casting(boxing and unboxing) that is required to add or retrieve objects from a collection**

The Issue of Performance

```csharp
public class IntCollection
{
    private ArrayList arInts = new ArrayList();
    // Get an int (performs unboxing)!
    public int GetInt(int pos) => (int)arInts[pos];
    // Insert an int (performs boxing)!
    public void AddInt(int n)=>arInts.Add(n);
    public void ClearInts()=> arInts.Clear();
    public int Count => arInts.Count;
}
```

```csharp
class Program {
    static void Main(string[] args){
        int s = 0, number;
        IntCollection collection = new IntCollection();
        collection.AddInt(10);
        collection.AddInt(20);
        collection.AddInt(30);
        for (int i = 0; i < collection.Count; i++){
            number = collection.GetInt(i);
            s += number;
            Console.Write($" {number} " +
                $"{(i == collection.Count - 1 ? " =" : "+")}");
        }
        Console.WriteLine($" {s}");
    }
}
```

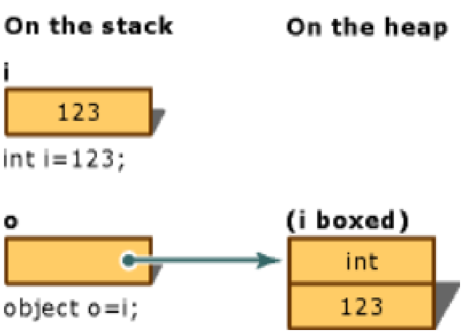
Microsoft Visual Studio Debug Console

```
10 + 20 + 30  = 60
```

The Issue of Performance

- **Problem with Boxing and UnBoxing Operations**

1) **A new object must be allocated on the managed heap**
2) **The value of the stack-based data must be transferred into that memory location**
3) **When unboxed, the value stored on the heap-based object must be transferred back to the stack**
4) **The now unused object on the heap will (eventually) be garbage collected**



What is the Generics?

- **Generics introduce the concept of type parameters to .NET, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code**
- **Generic allows type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes, and interfaces**
- **Generics are commonly used to create type-safe collections for both reference and value types. The .NET provides an extensive set of interfaces and classes in the System.Collections.Generic namespace for implementing generic collections**

Benefits of Generics

- **Ensure type-safety at compile-time: Ensure strongly-typed programming model**
- **Allow to reuse the code in a safe manner without casting or boxing:**
  - **Reduce run-time errors**
  - **Improve performance because of low memory usage as no casting or boxing operation is required**
- **Can be reusable with different types but can accept values of a single type at a time**
- **Generic delegates enable type-safe callbacks without the need to create multiple delegate classes**
  - **The Predicate<T> generic delegate allows us to create a method that implements our search criteria for a particular type and to use our method with methods of the Array type such as Find, FindLast, and FindAll**

Generic Classes

- **Generic classes encapsulate operations that are not specific to a particular data type**
- **The most common use for generic classes is with collections like linked lists, hash tables, stacks, queues, trees, and so on**
- **When creating our generic classes, important considerations include the following:**
  - **Which types to generalize into type parameters**
  - **What constraints, if any, to apply to the type parameters**
  - **Whether to factor generic behavior into base classes and subclasses**
  - **Whether to implement one or more generic interfaces**

Generic Classes

```
// Using <> to specify Parameter type
public class MyClass<T>{
    private T data;
    public T Value {
        get => data;
        set => data = value;
    }
    public override string ToString() => $"Value:{data}";
}
```

```
class Program {
    static void Main(string[] args) {
        // Instance of string type
        MyClass<string> name = new MyClass<string>() { Value="Jack" };
        Console.WriteLine(name);
        // Instance of float type
        MyClass<float> version = new MyClass<float>() { Value = 5.5f };
        Console.WriteLine(version);
        // Instance of dynamic type
        dynamic obj = new { Id = 1, Name = "David" };
        MyClass<dynamic> myClass = new MyClass<dynamic> { Value = obj };
        Console.WriteLine(myClass);
        Console.ReadLine();
    }
}
```

D:\Demo\FU\Basic.NET\Slot_06\Demo_Generic_Classes

```
Value:Jack
Value:5.5
Value:{ Id = 1, Name = David }
```

Generic Methods

- With a generic method, the generic type is defined with the method declaration
- Generic methods can be defined within non-generic classes

D:\Demo\FU\Basic.NET\Slot_06\...

```
Integer : 2050
155.9 : A
358.9 : 255.67
```

```
public class MyClass{
    // Generics method with two types T and U
    public void Display<T,U>(T msg, U value){
        Console.WriteLine($"{msg} : {value}");
    }
}
class Program {
    static void Main(string[] args){
        // Creating object of MyClass
        MyClass obj = new MyClass();
        // Calling Generics method
        obj.Display<string,int>("Integer", 2050);
        obj.Display<double,char>(155.9, 'A');
        obj.Display<float,double>(358.9F, 255.67);
        Console.ReadLine();
    }
}
```

Constraints on Type Parameters

- Constraints **inform the compiler** about the capabilities **a type argument must have**
- Without any constraints, the type argument could be any type. The compiler can only assume the members of System.Object, which is the ultimate base class for any .NET type
- Constraints are specified by using the **where** contextual keyword
- The following table lists the various types of constraints:

Why use constraints ?

Read by yourself

Constraint	Description
where T : struct	The type argument must be a non-nullable value type. The struct constraint implies the new() constraint and can't be combined with the new() constraint
where T : class	The type argument must be a reference type. This constraint applies also to any class, interface, delegate, or array type. T must be a non-nullable reference type
where T : class?	The type argument must be a reference type, either nullable or non-nullable. This constraint applies also to any class, interface, delegate, or array type
where T : notnull	The type argument must be a non-nullable type. The argument can be a non-nullable reference type or a non-nullable value type
where T : unmanaged	The type argument must be a non-nullable unmanaged type. The unmanaged constraint implies the struct constraint and can't be combined with either the struct or new() constraints
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 : *<base class name>?*	The type argument must be or derive from the specified base class. T may be either a nullable or non-nullable type derived from the specified base class
where T : *<interface name>*	The type argument must be or implement the specified interface
where T : *<interface name>?*	The type argument must be or implement the specified interface. T may be a nullable reference type, a non-nullable reference type, or a value type. T may not be a nullable value type
where T : U	The type argument supplied for T must be or derive from the argument supplied for U. In a nullable context, if U is a non-nullable reference type, T must be non-nullable reference type. If U is a nullable reference type, T may be either nullable or non-nullable

Constraints on Type Parameters

- **Multiple constraints can be applied to the same type parameter, and the constraints themselves can be generic types, as follows:**

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>, new()
{
    // ...
}
```

- **Constraining multiple parameters**

```
class Base {
    //...
}
class Test<T, U> where U : struct where T : Base, new()
{
    //...
}
```

Generic Interfaces

- **It is often useful to define interfaces either for generic collection classes, or for the generic classes that represent items in the collection**
- **The preference for generic classes is to use generic interfaces, such as IComparable<T> rather than IComparable, in order to avoid boxing and unboxing operations on value types**
- **The .NET class library defines several generic interfaces for use with the collection classes in the System.Collections.Generic namespace**
- **When an interface is specified as a constraint on a type parameter, only types that implement the interface can be used**

Generic Interfaces

```csharp
// Declare an interface with constraint:struct(Value type)
interface IBasic<T> where T:struct
{
    T Add(T a, T b);
}
// Implement interface IBasic with int type
class MyFirstClass : IBasic<int>
{
    public int Add(int a, int b) => a + b;
}
// Implement interface IBasic with double type
class MySecondClass : IBasic<double>
{
    public double Add(double a, double b) => a + b;
}
```

```csharp
class Program
{
    static void Main(string[] args)
    {
        MyFirstClass firstClass = new MyFirstClass();
        dynamic r = firstClass.Add(10, 20);
        Console.WriteLine(r);
        MySecondClass secondClass = new MySecondClass();
        r = secondClass.Add(10.5, 20.5);
        Console.WriteLine(r);
        Console.ReadLine();
    }
}
```

D:\Demo\FU\Basic.NET\Slot_06\Demo_Generic_Interfaces\

```
30
31
```

Default Values in Generics

- **With the default keyword, null is assigned to reference types and 0 is assigned to value types**

```
class MyClass<T>
{
    public T Value1 { get; set; } = 0; //error
    public T Value2 { get; set; } = default(T);
    //...
}
```

Collections in C#

Collection Interfaces and Types

- **Most collection classes are in the System.Collections and System.Collections.Generic namespaces**
- **Generic collection classes are located in the System.Collections.Generic namespace**
- **Collection classes that are specialized for a specific type are located in the System.Collections.Specialized namespace**
- **Thread-safe collection classes are in the System.Collections.Concurrent namespace**
- **The following table describes the most important interfaces implemented by collections and lists.**

Key Interfaces Supported by Classes of System.Collections.Generic

Interface	Description
ICollection<T>	Defines general characteristics (e.g., size, enumeration, and thread safety) for all generic collection types
IComparer<T>	Defines a way to compare to objects
IDictionary<TKey,TValue>	Allows a generic collection object to represent its contents using key-value pairs
IEnumerable/IAsyncEnumerable	Returns the IEnumerator interface for a given object
IEnumerator	Enables foreach-style iteration over a generic collection
IList	Provides behavior to add, remove, and index items in a sequential list of objects
ISet	Provides the base interface for the abstraction of sets

Classes of System.Collections.Generic

Interface	Supported Key Interfaces	**Description**
Dictionary<TKey,TValue>	ICollection<T>,IDictionary<TKey,TValue>, IEnumerable<T>	This represents a generic collection of keys and values
LinkedList<T>	ICollection<T>, IEnumerable<T>	This represents a doubly linked list
List<T>	ICollection<T>,IEnumerable<T>, IList<T>	This is a dynamically resizable sequential list of items
Queue	ICollection, IEnumerable<T>	This is a generic implementation of a first-in, first-out list
SortedDictionary<TKey,TValue>	ICollection<T>,IDictionary<TKey,TValue>, IEnumerable<T>	This is a generic implementation of a sorted set of key-value pairs
SortedSet<T>	ICollection<T>,IEnumerable<T>, ISet<T>	This represents a collection of objects that is maintained in sorted order with no duplication
Stack<T>	ICollection , IEnumerable<T>	This is a generic implementation of a last-in, first-out list

Demo Generics Collections

- List<T> Class
- SortedSet<T> Class
- IEnumerable<T> Interface

Working with the List<T> Class

- The List<T> is a collection of strongly typed objects that can be accessed by index and having methods for sorting, searching, and modifying list
- List<T> equivalent of the ArrayList, which implements IList<T>
- List<T> can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic
- Elements can be added using the Add(), AddRange() methods or collection-initializer syntax
- Elements can be accessed by passing an index. Indexes start from zero
- List<T> performs faster and less error-prone than the ArrayList

Working with the List<T> Class

```
public class Person {
    public int Age { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public override string ToString() =>
        $"Name: {FirstName} {LastName}, Age: {Age}";
}
```

```
class Program {
    static void Main(string[] args){
        List<Person> people = new List<Person>(){
            new Person {FirstName= "David", LastName="Simpson", Age=50},
            new Person {FirstName= "Marge", LastName="Simpson", Age=45},
            new Person {FirstName= "Lisa", LastName="Simpson", Age=19},
            new Person {FirstName= "Jack", LastName="Simpson", Age=16}
        };
        // Print out # of items in List.
        Console.WriteLine("Items in list: {0}", people.Count);
        // Enumerate over list.
        foreach (Person p in people){
            Console.WriteLine(p);
        }
        Console.ReadLine();
    }
}
```

D:\Demo\FU\Basic.NET\Slot_06\DemoList\bin\Det

```
Items in list: 4
Name: David Simpson, Age: 50
Name: Marge Simpson, Age: 45
Name: Lisa Simpson, Age: 19
Name: Jack Simpson, Age: 16
```

Working with the SortedSet<T> Class

- **SortedSet is a collection of objects in sorted order. It is of the generic type collection and defined under System.Collections.Generic namespace**
- **It also provides many mathematical set operations, such as intersection, union, and difference**
- **It is a dynamic collection means the size of the SortedSet is automatically increased when the new elements are added**
- **In SortedSet, the elements must be unique and the order of the element is ascending**
- **It is generally used SortedSet class if we have to store unique elements and maintain ascending order**
- **In SortedSet, the we can only store the same type of elements**

Working with the SortedSet<T> Class

```
class Program{
    static void Main(string[] args){
        //using collection initializer to initialize SortedSet
        SortedSet<int> mySet = new SortedSet<int>(){8,7,9,1,3};
        // Add the elements in SortedSet using Add method
        mySet.Add(5);
        mySet.Add(4);
        mySet.Add(6);
        mySet.Add(2);
        Console.WriteLine("Elements of mySet:\n");
        // Accessing elements of SortedSet using foreach loop
        foreach (var val in mySet){
            Console.Write($"{val,3}");
        }
        Console.ReadLine();
    }
}
```

D:\Demo\FU\Basic.NET\Slot_06\DemoSortSet\bin\D

```
Elements of mySet:

  1  2  3  4  5  6  7  8  9
```

The LinkList<T> Class

- LinkedList<T> Class is a generic type that allows fast inserting and removing of elements. It implements a classic linked list
- Each object is separately allocated. In the LinkedList, certain operations do not require the whole collection to be copied
- We can remove nodes and reinsert them, either in the same list or in another list, which results in no additional objects allocated on the heap.
- Each node in a LinkedList<T> object is of the type LinkedListNode<T>
- The LinkedList class does not support chaining, splitting, cycles, or other features that can leave the list in an inconsistent state
- The LinkedList is doubly linked, therefore, each node points forward to the Next node and backward to the Previous node

The Dictionary<TKey, TValue> Class

- The Dictionary<TKey, TValue> is a generic collection that stores key-value pairs in no particular order
- Dictionary<TKey, TValue> stores key-value pairs
- Keys must be unique and cannot be null
- Values can be null or duplicate
- Values can be accessed by passing associated key in the indexer(e.g. myDictionary[key])
- Elements are stored as KeyValuePair<TKey, TValue> objects

The IEnumerable<T> Interface

- IEnumerable in C# is an interface that defines one method GetEnumerator which returns an IEnumerator interface. This allows readonly access to a collection then a collection that implements IEnumerable can be used with a **for-each statement**
- Implement the **IEnumerable<T>** Interface :

```
public class Person{
    public int Age { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Person(){ }
    public override string ToString() => $"Name: {FirstName} {LastName}, Age: {Age}";
}
```

The IEnumerable<T> Interface

```
public class MyCollection<T>:IEnumerable where T: class, new() {
    private List<T> myList = new List<T>();
    public void AddItem(params T[] item) => myList.AddRange(item);
    IEnumerator IEnumerable.GetEnumerator() => myList.GetEnumerator();
}
class Program {
    static void Main(string[] args){
        MyCollection<Person> collection = new MyCollection<Person>();
        var p1 = new Person { FirstName = "David", LastName = "Simpson", Age = 50 };
        var p2 = new Person { FirstName = "Marge", LastName = "Simpson", Age = 45 };
        var p3 = new Person { FirstName = "Lisa", LastName = "Simpson", Age = 19 };
        var p4 = new Person { FirstName = "Jack", LastName = "Simpson", Age = 16 };
        collection.AddItem(p1, p2,p3,p4);
        foreach (var p in collection){
            Console.WriteLine(p);
        }
    }
}
```

Microsoft Visual Studio Debug Console

```
Name: David Simpson, Age: 50
Name: Marge Simpson, Age: 45
Name: Lisa Simpson, Age: 19
Name: Jack Simpson, Age: 16
```

List<T> – Data Structure

- **List<T>** holds a list of elements (like array, but extendable)
- Provides operations to **add** / **insert** / **remove** / **find** elements:
  - **Add(element)** – adds an element to the List<T>
  - **Count** – number of elements in the List<T>
  - **Remove(element)** – removes an element (returns true / false)
  - **RemoveAt(index)** – removes element at index
  - **Insert(index, element)** – inserts an element to given position
  - **Contains(element)** – determines whether an element is in the list
  - **Sort()** – sorts the array/list in ascending order

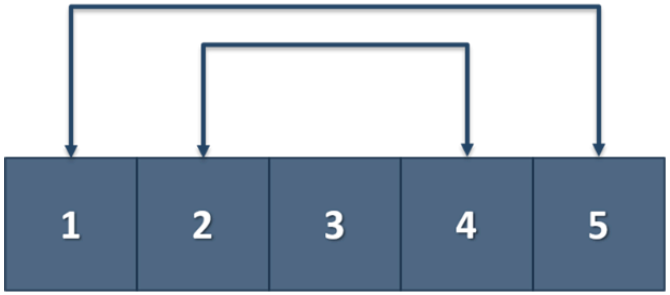
Problem: Sum Adjacent Equal Numbers

- Write a program to sum all adjacent equal numbers in a list of decimal numbers, starting from left to right.
- Examples:

3 3 6 1	➡	12 1
8 2 2 4 8 16	➡	16 8 16
5 4 2 1 1 4	➡	5 8 4

Problem: Gauss' Trick

- Write a program that sum all numbers in a list in the following order:
  - first + last, first + 1 + last - 1, first + 2 + last - 2, ... first + n, last – n
- Examples:



1 2 3 4 5 → 6 6 3

1 2 3 4 → 5 5

Problem: Merging Lists

- You receive two lists with numbers. Print a result list which contains the numbers from both of the lists.
  - If the length of the two lists are not equal, just add the remaining elements at the end of the list:
  - list1[0], list2[0], list1[1], list2[1], …

```
1 2 3 4 5
6 7 8
```

→

```
1 6 2 7 3 8 4 5
```

LAB1

❑Exercise 3: List

```
Student st1 = new Student(1, "IT", 3.8f, "Nguyễn Văn Lợi");
Student st2 = new Student(2, "BU", 9.2f, "Nguyễn Văn Nghia");
List<Student> ds = new List<Student>();
ds.Add(st1);
ds.Add(st2);
foreach (Student st in ds)
{
    st.Show();
}
```

LAB1

❑Exercise 3: ArrayList

```
ArrayList MyArray = new ArrayList();
Student st1 = new Student(1, "IT", 9.8f, "Nguyễn Văn Lợi");
Student st2 = new Student(2, "BU", 9.2f, "Nguyễn Văn Nghia");

List<Student> ds = new List<Student>();
MyArray.Add(st1);
MyArray.Add(st2);
MyArray.Add(new Student(3, "LY", 7.5f, "An Binh"));
```

Icomparable/ Icomparer

```
class Student:
{
        public int Ma { get; set; }
        public string Ten { get; set; }
        public float DiemTB { get; set; }
}
```

```
List<Student> LStudent = new List<Student>();
LStudent.Add(new Student(1, "Binh", 8.6f));
LStudent.Add(new Student(3, "An", 8.1f));
LStudent.Add(new Student(2, "Nam", 9.9f));
```

Icomparable/ Icomparer

How to sort List Lstudent?

- **Using Icomparable: 3 steps**

  Step 1: class Student : IComparable<Student>

  Step 2: Implement method CompareTo()

```
//Order by Descending
public int CompareTo(Student other)
      {
         if (DiemTB > other.DiemTB) return -1;
         else if (DiemTB == other.DiemTB) return 0;
         else return 1;
      }
```

  Step 3: Invoke method Sort():

  LStudent.Sort();

IComparable/ IComparer

How to sort List Lstudent?

- **Using Icomparer: 2 steps**

  Step 1: Create new Class SortTen Inheritance from Icomparer.

  Step 2: Implement CompareTo method

  Step 3: Lstudent.Sort(new SortTen());

Icomparable/ Icomparer

How to sort List Lstudent?
Using Icomparer: 2 steps
Step 1: Create new Class SortTen

```
{class SortTen : IComparer

    public int Compare(object x, object y)
    {
        //ép kiểu về đối tượng so sánh
        SinhVien sv1 = x as SinhVien;
        SinhVien sv2 = y as SinhVien;

        if (sv1 == null || sv2 == null)
        {
            throw new NotImplementedException();
        }
        else
        //Order by Ascending
        //return string.Compare(sv1.Ten, sv2.Ten);
        //Order by Descending
        {
            if (sv1.Ten.CompareTo(sv2.Ten) == 1) return -1;
            else if (sv1.Ten.CompareTo(sv2.Ten) == -1) return 1;
            else return 0;
        }
    }
}
```

Thank You !