

Using C#'s Generic List

by Craig Murphy

It is a fairly common programming scenario to find ourselves with a list of identical objects. In the past, without adequate support from programming languages, we found ourselves writing a lot of searching and sorting code, and that may have put you off using lists in favour of arrays. All that has changed with C# (particularly 2.0) - its implementation of a list makes handling such lists remarkably easy. Over the course of a very simple example, I will demonstrate the power that lists have to offer, power that will hopefully lead to more elegant code.

Preamble...

I often find myself working with a handful of records from a database table or other persistent store. Rarely do I find myself working with thousands of records or entire tables. However, once I've got the handful of records that I need, I often need to sort them or filter them based on another conditions. I could make another trip to the database table and use different SQL to sort the records or filter them. However, that may have a performance impact - I've already got the records I need "in memory", so why can't I do something with those instead. That's where .NET 2.0's System.Collections.Generic.List comes to the rescue.

Getting Started

I won't be going into any detail about how C# or .NET 2.0 deals with generics - plenty has been written about that elsewhere (as a starter, there's a link to Juval Lowey's article in the Links section at the end of this article). I will be presenting a very simple, and hopefully very useable, example...one that I find myself using in variety of scenarios. You'll need .NET 2.0 installed and a compiler that supports the System.Collections.Generic namespace. Whilst I used Visual Studio 2005 Professional, I tested the example using Visual C# Express Edition too. Speaking of the Express Editions, Microsoft have announced that the Express Editions are now free, forever (for more information, please refer to the link at the end of this article).

Let's build up a simple example. Consider Listing 1 which presents the class Person:

```
public class Person
{
    public int age;
    public string name;

    public Person(int age, string name)
    {
        this.age = age;
        this.name = name;
    }
}
```

Listing 1

Prior to .NET 2.0, we might have used an ArrayList to manage collections of Person objects. However, whilst that worked, the ArrayList mechanism would not prevent us from accidentally adding an object other than Person and that can create problems for us downstream. Also, the ArrayList has some special requirements if you need to sort the items in the collection and filtering is a manual exercise.

Adding Items To The List

So, having decided that we are going to use the new generic list support provided in .NET 2.0, we can create a list of Person objects and add six people like so:

```
List<Person>people = new List<Person>();
people.Add(new Person(50, "Fred"));
people.Add(new Person(30, "John"));
people.Add(new Person(26, "Andrew"));
people.Add(new Person(24, "Xavier"));
people.Add(new Person(5, "Mark"));
people.Add(new Person(6, "Cameron"));
```

Listing 2

In reality, you may have populated the list of people by querying a database. Your code is then free to work on the collection of Person objects without knowing where they came from or how they are persisted. This approach [using lists] makes writing elegant collection processing code much easier.

Sorting The List

Now that the people list is populated, there will come a time when we need to sort it. The elegance that I keep referring to should become evident once you have read through Listing 3.

Listing 3 achieves three things:

1. It outputs the unsorted, vanilla people list as we populated it using Listing 2.
2. It sorts the people list by name, then outputs the sorted list.
3. It sorts the people list by age, then outputs the sorted list.

```
Console.WriteLine("Unsorted list");

people.ForEach(delegate(Person p)
    { Console.WriteLine(String.Format("{0} {1}", p.age, p.name)); });

// Sort by name
Console.WriteLine("Sorted list, by name");

people.Sort(delegate(Person p1, Person p2)
    { return p1.name.CompareTo(p2.name); });

people.ForEach(delegate(Person p)
    { Console.WriteLine(String.Format("{0} {1}", p.age, p.name)); });

// Sort by age
Console.WriteLine("Sorted list, by age");

people.Sort(delegate(Person p1, Person p2)
    { return p1.age.CompareTo(p2.age); });

people.ForEach(delegate(Person p)
    { Console.WriteLine(String.Format("{0} {1}", p.age, p.name)); });
```

Listing 3

Here's the output that we should expect to see:

Unsorted list	Sorted list, by name	Sorted list, by age
50 Fred	26 Andrew	5 Mark
30 John	6 Cameron	6 Cameron
26 Andrew	50 Fred	24 Xavier
24 Xavier	30 John	26 Andrew
5 Mark	5 Mark	30 John
6 Cameron	24 Xavier	50 Fred

Filtering

Given the list people, it is not unreasonable to assume that we may wish to filter the list based on a person's age. Fortunately, .NET 2.0's generic list offers us two methods, Find and FindAll. Like the sort method, these methods allow us to write some very elegant code that essentially queries the list people.

Listing 4 demonstrates how we might output the unsorted list, using the FindAll method to create a new list of young persons

```
Console.WriteLine("Unsorted list");

people.ForEach(delegate(Person p)
    { Console.WriteLine(String.Format("{0} {1}", p.age, p.name)); });
```

```
// Find the young
List<Person> young = people.FindAll(delegate(Person p) { return p.age < 25; });

Console.WriteLine("Age is less than 253");

young.ForEach(delegate(Person p)
    { Console.WriteLine(String.Format("{0} {1}", p.age, p.name)); });
```

Listing 4

Here's the output that we should expect to see:

Unsorted list	Age is less than 25
50 Fred	24 Xavier
30 John	5 Mark
26 Andrew	6 Cameron
24 Xavier	
5 Mark	
6 Cameron	

Lastly, the Find method allows us quickly to locate an individual Person object. Listing 5 demonstrates the use of the Find method.

```
Person mySon = people.Find(delegate(Person p) { return p.name == "Cameron"; });
```

Listing 5

One of the key benefits of the Find and FindAll methods is the fact that they return references to the original objects. This means that after using Find and FindAll, the results we have are considered "live", i.e. any changes that we make to the filtered object(s) are actually performed on the original object. This has the advantage that any other methods that use the original object, perhaps from the unsorted and unfiltered people list, will enjoy the benefits of working with an "update once" mechanism.

Conclusions

Lists are powerful and result in fewer, and more elegant, lines of code. Hopefully this short example has demonstrated their ease and you will find yourself using them in your day-to-day development activities.

Links

IDesign's Juval Lowey on Generics:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs05/html/csharp_generics.asp

Visual C# Express Edition – Free Forever:

<http://blogs.msdn.com/danielfe/archive/2006/04/19/579109.aspx>

Download Visual C# Express:

<http://msdn.microsoft.com/vstudio/express/visualcsharp/>

And the benefits of registering your copy of Visual C# Express:

<http://msdn.microsoft.com/vstudio/regbenefits/>



Craig is an author, developer, speaker, blogger, Certified ScrumMaster and Microsoft Most Valuable Professional (Connected Systems). He specialises in all things XML, particularly web services and XSLT. Craig is evangelical about .NET, C#, Test-Driven Development, Extreme Programming, agile methods and Scrum. He can be reached via e-mail at: ddg@craigmurphy.com, or via his web site: <http://www.craigmurphy.com> (where you can also find the source code and PowerPoint files for all of Craig's articles, reviews and presentations).