**Lecture 8 – Data Structures, Collections, Generic**

Introduction to C# collections library

Any application you create with the .NET platform will need to contend with the issue of maintaining and manipulating a set of data points in memory. These data points can come from any variety of locations including a relational database, a local text file, an XML document, a web service call, or perhaps via user provided input.

When the .NET platform was first released, programmers frequently used the classes of the

System.Collections namespace to store and interact with bits of data used within an application. In .NET 2.0, the C# programming language was enhanced to support a feature termed *generics*; and with this change, a brand new namespace was introduced in the base class libraries:

System.Collections.Generic.

This chapter will provide you will an overview of the various collection (generic and nongeneric) namespaces and types found within the .NET base class libraries. As you will see, generic containers are often favored over their nongeneric counterparts because they typically provide greater type safety and performance benefits. After you’ve seen learned how to create and manipulate the generic items found in the framework, the remainder of this chapter will examine how to build your own generic methods and generic types. As you do this, you will learn about the role of *constraints (and the corresponding C#* where *keyword)*, which allow you to build extremely type-safe classes.

The System.Collections Namespace

When the .NET platform was first released, programmers frequently used the nongeneric collection classes found within the System.Collections namespace, which contains a set of classes used to manage and organize large amounts of in-memory data. Table 9-1 documents some of the more commonly used collection classes of this namespace, and the core interfaces they implement.

**Note** Any .NET application built with .NET 2.0 or higher should ignore the classes in System.Collections in favor of the corresponding classes in System.Collections.Generic. However, it is important to know the basics of the nongeneric collection classes, as you might have some legacy software to maintain.

The Problems of Nongeneric Collections

While it is true that many successful .NET applications have been built over the years using these nongeneric collection classes (and interfaces), history has shown that use of these types can result in a number of issues.

The first issue is that using the System.Collections and System.Collections.Specialized classes

can result in some poorly performing code, especially when you are manipulating numerical data (e.g., value types). As you’ll see momentarily, the CLR must perform a number of memory transfer operations when you store structures in any nongeneric collection class prototyped to operate on System.Objects, which can hurt runtime execution speed.

The second issue is that most of the nongeneric collection classes are not type safe because (again) they were developed to operate on System.Objects, and they could therefore contain anything at all. If a .NET developer needed to create a highly type-safe collection (e.g., a container that can hold objects implementing only a certain interface), the only real choice was to create a brand new collection class by hand. Doing so was not too labor intensive, but it was a tad bit on the tedious side.

Before you look at how to use generics in your programs, you’ll find it helpful to examine the issues of nongeneric collection classes a bit closer; this will help you better understand the problems generics intend to solve in the first place. If you want to follow along, create a new Console Application named IssuesWithNonGenericCollections. Next, make sure you import the System.Collections namespace to the top of your C# code file: using System.Collections;

The Issue of Performance

As you might recall from Chapter 4, the .NET platform supports two broad categories of data: value types and reference types. Given that .NET defines two major categories of types, you might occasionally need to represent a variable of one category as a variable of the other category. To do so, C# provides a simple mechanism, termed *boxing*, to store the data in a value type within a reference variable. Assume that you

have created a local variable of type int in a method called SimpleBoxUnboxOperation(). If, during the course of your application, you were to represent this value type as a reference type, you would *box* the value, as follows:

static void SimpleBoxUnboxOperation()

{

// Make a ValueType (int) variable.

int myInt = 25;

// Box the int into an object reference.

object boxedInt = myInt;

}

Boxing can be formally defined as the process of explicitly assigning a value type to a System.Object variable. When you box a value, the CLR allocates a new object on the heap and copies the value type’s value (25, in this case) into that instance. What is returned to you is a reference to the newly allocated heap-based object.

The opposite operation is also permitted through *unboxing*. Unboxing is the process of converting the value held in the object reference back into a corresponding value type on the stack. Syntactically speaking, an unboxing operation looks like a normal casting operation. However, the semantics are quite different. The CLR begins by verifying that the receiving data type is equivalent to the boxed type; and if so, it copies the value back into a local stack-based variable. For example, the following unboxing operations work successfully, given that the underlying type of the boxedInt is indeed an int:

static void SimpleBoxUnboxOperation()

{

// Make a ValueType (int) variable.

int myInt = 25;

// Box the int into an object reference.

object boxedInt = myInt;

// Unbox the reference back into a corresponding int.

int unboxedInt = (int)boxedInt;

}

When the C# compiler encounters boxing/unboxing syntax, it emits CIL code that contains the

box/unbox op codes. If you were to examine your compiled assembly using ildasm.exe, you would find the following:

.method private hidebysig static void SimpleBoxUnboxOperation() cil managed

{

// Code size 19 (0x13)

.maxstack 1

.locals init ([0] int32 myInt, [1] object boxedInt, [2] int32 unboxedInt)

IL\_0000: nop

IL\_0001: ldc.i4.s 25

IL\_0003: stloc.0

IL\_0004: ldloc.0

IL\_0005: box [mscorlib]System.Int32

IL\_000a: stloc.1

IL\_000b: ldloc.1

IL\_000c: unbox.any [mscorlib]System.Int32

IL\_0011: stloc.2

IL\_0012: ret

} // end of method Program::SimpleBoxUnboxOperation

Remember that unlike when performing a typical cast, you *must* unbox into an appropriate data

type. If you attempt to unbox a piece of data into the incorrect data type, an InvalidCastException exception will be thrown. To be perfectly safe, you should wrap each unboxing operation in try/catch logic; however, this would be quite labor intensive to do for every unboxing operation. Consider the following code update, which will throw an error because you’re attempting to unbox the boxed int into a long:

static void SimpleBoxUnboxOperation()

{

// Make a ValueType (int) variable.

int myInt = 25;

// Box the int into an object reference.

object boxedInt = myInt;

// Unbox in the wrong data type to trigger

// runtime exception.

try

{

long unboxedInt = (long)boxedInt;

}

catch (InvalidCastException ex)

{

Console.WriteLine(ex.Message);

}

}

At first glance, boxing/unboxing might seem like a rather uneventful language feature that is more academic than practical. After all, you will seldom need to store a local value type in a local object variable, as seen here. However, it turns out that the boxing/unboxing process is quite helpful because it allows you to assume everything can be treated as a System.Object, while the CLR takes care of the memory-related details on your behalf.

Let’s look at a practical use of these techniques. Assume you have created a nongeneric System.Collections.ArrayList to hold onto a batch of numeric (stack-allocated) data. If you were to examine the members of ArrayList, you would find they are prototyped to operate on System.Object data. Now consider the Add(), Insert(), and Remove() methods, as well as the class indexer:

public class ArrayList : object,

IList, ICollection, IEnumerable, ICloneable

{

...

public virtual int Add(object value);

public virtual void Insert(int index, object value);

public virtual void Remove(object obj);

public virtual object this[int index] {get; set; }

}

ArrayList has been built to operate on objects, which represent data allocated on the heap, so it

might seem strange that the following code compiles and executes without throwing an error:

static void WorkWithArrayList()

{

// Value types are automatically boxed when

// passed to a method requesting an object.

ArrayList myInts = new ArrayList();

myInts.Add(10);

myInts.Add(20);

myInts.Add(35);

}

Although you pass in numerical data directly into methods requiring an object, the runtime

automatically boxes the stack-based data on your behalf. Later, if you want to retrieve an item from the ArrayList using the type indexer, you must unbox the heap-allocated object into a stack-allocated integer using a casting operation. Remember that the indexer of the ArrayList is returning System.Objects, not System.Int32s:

static void WorkWithArrayList()

{

// Value types are automatically boxed when

// passed to a member requesting an object.

ArrayList myInts = new ArrayList();

myInts.Add(10);

myInts.Add(20);

myInts.Add(35);

// Unboxing occurs when a object is converted back to

// stack-based data.

int i = (int)myInts[0];

// Now it is reboxed, as WriteLine() requires object types!

Console.WriteLine("Value of your int: {0}", i);

}

Again, note that the stack-allocated System.Int32 is boxed prior to the call to ArrayList.Add(), so it can be passed in the required System.Object. Also note that the System.Object is unboxed back into a System.Int32 once it is retrieved from the ArrayList via the casting operation, only to be boxed *again* when it is passed to the Console.WriteLine() method, as this method is operating on System.Object variables.

Boxing and unboxing are convenient from a programmer’s viewpoint, but this simplified approach to stack/heap memory transfer comes with the baggage of performance issues (in both speed of execution and code size) and a lack of type safety. To understand the performance issues, ponder the steps that must occur to box and unbox a simple integer.

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.

Although this particular WorkWithArrayList() method won’t cause a major bottleneck in terms of performance, you could certainly feel the impact if an ArrayList contained thousands of integers that your program manipulates on a somewhat regular basis. In an ideal world, you could manipulate stackbased data in a container without any performance issues. Ideally, it would be nice if you did not have to have to bother plucking data from this container using try/catch scopes (this is exactly what generics let you achieve).

The Issue of Type Safety

We touched on the issue of type safety when we looked at unboxing operations. Recall that you must unbox your data into the same data type it was declared as before boxing. However, there is another aspect of type safety you must keep in mind in a generic-free world: the fact that a majority of the classes of System.Collections can typically hold anything whatsoever because their members are prototyped to operate on System.Objects. For example, this method builds an ArrayList of random bits of unrelated data:

static void ArrayListOfRandomObjects()

{

// The ArrayList can hold anything at all.

ArrayList allMyObjects = new ArrayList();

allMyObjects.Add(true);

allMyObjects.Add(new OperatingSystem(PlatformID.MacOSX, new Version(10, 0)));

allMyObjects.Add(66);

allMyObjects.Add(3.14);

}

In some cases, you will require an extremely flexible container that can hold literally anything (as seen here). However, most of the time you desire a *type-safe* container that can operate only on a particular type of data point. For example, you might need a container that can hold only database connections, bitmaps, or IPointy-compatible objects.

Prior to generics, the only way you could address this issue of type safety was to create a custom

(strongly typed) collection class manually. Assume you wish to create a custom collection that can contain only objects of type Person:

public class Person

{

public int Age {get; set;}

public string FirstName {get; set;}

public string LastName {get; set;}

public Person(){}

public Person(string firstName, string lastName, int age)

{

Age = age;

FirstName = firstName;

LastName = lastName;

}

public override string ToString()

{

return string.Format("Name: {0} {1}, Age: {2}",

FirstName, LastName, Age);

}

}

To build a *collection which can hold only* Person *objects*, you could define a

System.Collections.ArrayList member variable within a class named PersonCollection and configure all members to operate on strongly typed Person objects, rather than on System.Object types. Here is a simple example (a production-level custom collection could support many additional members and might extend an abstract base class from the System.Collections or System.Collections.Specialized namespace):

public class PersonCollection : IEnumerable

{

private ArrayList arPeople = new ArrayList();

// Cast for caller.

public Person GetPerson(int pos)

{ return (Person)arPeople[pos]; }

// Insert only Person objects.

public void AddPerson(Person p)

{ arPeople.Add(p); }

public void ClearPeople()

{ arPeople.Clear(); }

public int Count

{ get { return arPeople.Count; } }

// Foreach enumeration support.

IEnumerator IEnumerable.GetEnumerator()

{ return arPeople.GetEnumerator(); }

}

Notice that the PersonCollection class implements the IEnumerable interface, which allows a

foreach-like iteration over each contained item. Also notice that your GetPerson() and AddPerson()

methods have been prototyped to operate only on Person objects, not bitmaps, strings, database

connections, or other items. With these types defined, you are now assured of type safety, given that the C# compiler will be able to determine any attempt to insert an incompatible data type:

static void UsePersonCollection()

{

Console.WriteLine("\*\*\*\*\* Custom Person Collection \*\*\*\*\*\n");

PersonCollection myPeople = new PersonCollection();

myPeople.AddPerson(new Person("Homer", "Simpson", 40));

myPeople.AddPerson(new Person("Marge", "Simpson", 38));

myPeople.AddPerson(new Person("Lisa", "Simpson", 9));

myPeople.AddPerson(new Person("Bart", "Simpson", 7));

myPeople.AddPerson(new Person("Maggie", "Simpson", 2));

// This would be a compile-time error!

// myPeople.AddPerson(new Car());

foreach (Person p in myPeople)

Console.WriteLine(p);

}

While custom collections do ensure type safety, this approach leaves you in a position where you

must create an (almost identical) custom collection for each unique data type you want to contain. Thus, if you need a custom collection that can operate only on classes deriving from the Car base class, you need to build a highly similar collection class:

{

private ArrayList arCars = new ArrayList();

// Cast for caller.

public Car GetCar(int pos)

{ return (Car) arCars[pos]; }

// Insert only Car objects.

public void AddCar(Car c)

{ arCars.Add(c); }

public void ClearCars()

{ arCars.Clear(); }

public int Count

{ get { return arCars.Count; } }

// Foreach enumeration support.

IEnumerator IEnumerable.GetEnumerator()

{ return arCars.GetEnumerator(); }

}

However, a custom collection class does nothing to solve the issue of boxing/unboxing penalties.

Even if you were to create a custom collection named IntCollection that you designed to operate only on System.Int32 items, you would have to allocate some type of object to hold the data (e.g., System.Array and ArrayList):

public class IntCollection : IEnumerable

{

private ArrayList arInts = new ArrayList();

// Get an int (performs unboxing!).

public int GetInt(int pos)

{ return (int)arInts[pos]; }

// Insert an int (performs boxing)!

public void AddInt(int i)

{ arInts.Add(i); }

public void ClearInts()

{ arInts.Clear(); }

public int Count

{ get { return arInts.Count; } }

IEnumerator IEnumerable.GetEnumerator()

{ return arInts.GetEnumerator(); }

}

Regardless of which type you might choose to hold the integers, you cannot escape the boxing

dilemma using nongeneric containers.

A First Look at Generic Collections

When you use generic collection classes, you rectify all of the previous issues, including

boxing/unboxing penalties and a lack of type safety. Also, the need to build a custom (generic) collection class becomes quite rare. Rather than having to build unique classes that can contain people, cars, and integers, you can use a generic collection class and specify the type of type.

Consider the following method, which uses the generic List<T> class (in the

System.Collections.Generic namespace) to contain various types of data in a strongly typed manner (don’t fret the details of generic syntax at this time):

static void UseGenericList()

{

Console.WriteLine("\*\*\*\*\* Fun with Generics \*\*\*\*\*\n");

// This List<> can hold only Person objects.

List<Person> morePeople = new List<Person>();

morePeople.Add(new Person ("Frank", "Black", 50));

Console.WriteLine(morePeople[0]);

// This List<> can hold only integers.

List<int> moreInts = new List<int>();

moreInts.Add(10);

moreInts.Add(2);

int sum = moreInts[0] + moreInts[1];

// Compile-time error! Can't add Person object

// to a list of ints!

// moreInts.Add(new Person());

}

The first List<T> object can contain only Person objects. Therefore, you do not need to perform a cast when plucking the items from the container, which makes this approach more type safe. The second List<T> can contain only integers, all of which are allocated on the stack; in other words, there is no hidden boxing or unboxing as you found with the nongeneric ArrayList. Here is a short list of the benefits generic containers provide over their nongeneric counterparts.

• Generics provide better performance because they do not result in boxing or

unboxing penalties when storing value types.

• Generics are type safe because they can contain only the type of type you specify.

• Generics greatly reduce the need to build custom collection types because you

specify the “type of type” when creating the generic container.

The Role of Generic Type Parameters

You can find generic classes, interfaces, structures, and delegates throughout the .NET base class

libraries, and these might be part of any .NET namespace. Also be very aware that generics have far more uses then simply defining a collection class. To be sure, you will see many different generics used in the remainder of this book for various reasons.

Specifying Type Parameters for Generic Interfaces

It is common to implement generic interfaces when you build classes or structures that need to support various framework behaviors (e.g., cloning, sorting, and enumeration). In Chapter 8, you learned about a number of nongeneric interfaces, such as IComparable, IEnumerable, IEnumerator, and IComparer. Recall that the nongeneric IComparable interface was defined like this:

public interface IComparable

{

int CompareTo(object obj);

}

You also implemented this interface on your Car class to enable sorting in a standard

array. However, the code required several runtime checks and casting operations because the parameter

was a general System.Object:

public class Car : IComparable

{

...

// IComparable implementation.

int IComparable.CompareTo(object obj)

{

Car temp = obj as Car;

if (temp != null)

{

if (this.CarID > temp.CarID)

return 1;

if (this.CarID < temp.CarID)

return -1;

else

return 0;

}

else

throw new ArgumentException("Parameter is not a Car!");

}

}

Now assume you use the generic counterpart of this interface:

public interface IComparable<T>

{

int CompareTo(T obj);

}

In this case, your implementation code will be cleaned up considerably:

public class Car : IComparable<Car>

{

...

// IComparable<T> implementation.

int IComparable<Car>.CompareTo(Car obj)

{

if (this.CarID > obj.CarID)

return 1;

if (this.CarID < obj.CarID)

return -1;

else

return 0;

}

}

Here, you do not need to check whether the incoming parameter is a Car because it can *only* be a

Car! If someone were to pass in an incompatible data type, you would get a compile-time error. Now that you have a better handle on how to interact with generic items, as well as the role of type parameters (a.k.a. placeholders), you’re ready to examine the classes and interfaces of the

System.Collections.Generic namespace.

The System.Collections.Generic Namespace When you are building a .NET application and need a way to manage in-memory data, the classes of System.Collections.Generic will most likely fit the bill. At the opening of this chapter, I briefly mentioned some of the core nongeneric interfaces implemented by the nongeneric collection classes. Not too surprisingly, the System.Collections.Generic namespace defines generic replacements for many of them.

In fact, you can find a number of the generic interfaces that extend their nongeneric counterparts.

This might seem odd; however, by doing so, implementing classes will also support the legacy

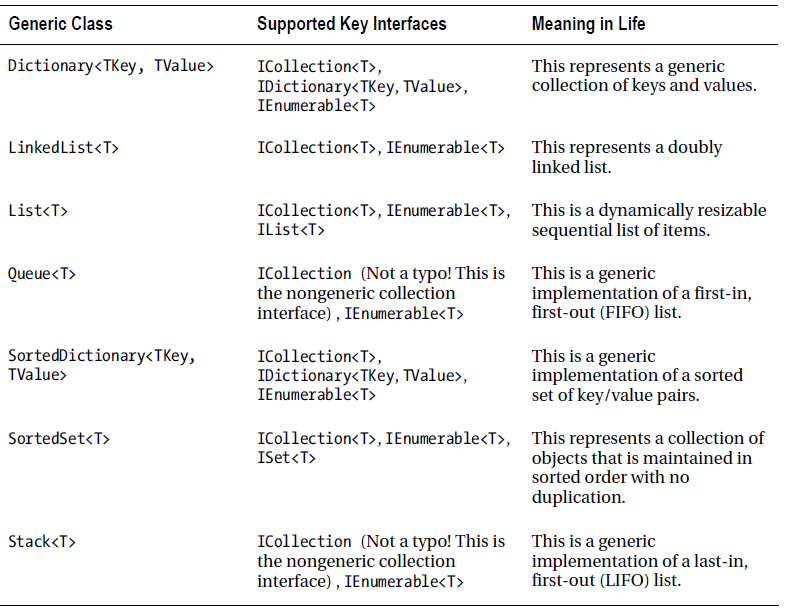
functionally found in their nongeneric siblings. For example, IEnumerable<T> extends IEnumerable

*Key Interfaces Supported by Classes of System.Collections.Generic*

*ICollection<T>, IComparer <T>, IDictionary <TKey, TValue>, IEnumerable <T>, IEnumerator<T>, IList<T>, ISet<T>*

The System.Collections.Generic namespace also defines several classes that implement many of

these key interfaces.

**

The System.Collections.Generic namespace also defines many auxiliary classes and structures that work in conjunction with a specific container. For example, the LinkedListNode<T> type represents a node within a generic LinkedList<T>, the KeyNotFoundException exception is raised when attempting to grab an item from a container using a nonexistent key, and so forth.

It is also worth pointing out that mscorlib.dll and System.dll are not the only assemblies that add new types to the System.Collections.Generic namespace. For example, System.Core.dll adds the HashSet<T> class to the mix. Be sure to consult the .NET Framework documentation for full details of the System.Collections.Generic namespace.

In any case, your next task is to learn how to use some of these generic collection classes. Before you do however, allow me to illustrate a C# language feature (first introduced in .NET 3.5) that simplifies the way you populate generic (and nongeneric) collection containers with data.

Understanding Collection Initialization Syntax

In Chapter 4, you learned about *object initialization syntax*, which allows you to set properties on a new variable at the time of construction. Closely related to this is *collection initialization syntax*. This C# language feature makes it possible to populate many containers (such as ArrayList or List<T>) with items by using syntax similar to what you use to populate a basic array.

 **Note** You can apply collection initialization syntax only to classes that support an Add() method, which is

formalized by the ICollection<T>/ICollection interfaces.

Consider the following examples:

// Init a standard array.

int[] myArrayOfInts = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Init a generic List<> of ints.

List<int> myGenericList = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Init an ArrayList with numerical data.

ArrayList myList = new ArrayList { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

If your container is managing a collection of classes or a structure, you can blend object

initialization syntax with collection initialization syntax to yield some functional code. You might recall the Point class from Chapter 5, which defined two properties named X and Y. If you wanted to build a generic List<T> of Point objects, you could write the following:

List<Point> myListOfPoints = new List<Point>

{

new Point { X = 2, Y = 2 },

new Point { X = 3, Y = 3 },

new Point(PointColor.BloodRed){ X = 4, Y = 4 }

};

foreach (var pt in myListOfPoints)

{

Console.WriteLine(pt);

}

Again, the benefit of this syntax is that you save yourself numerous keystrokes. While the nested

curly brackets can become difficult to read if you don’t mind your formatting, imagine the amount of code that would be required to fill the following List<T> of Rectangles if you did not have collection initialization syntax (you might recall from Chapter 4 that you created a Rectangle class that contained two properties encapsulating Point objects):

List<Rectangle> myListOfRects = new List<Rectangle>

{

new Rectangle {TopLeft = new Point { X = 10, Y = 10 },

BottomRight = new Point { X = 200, Y = 200}},

new Rectangle {TopLeft = new Point { X = 2, Y = 2 },

BottomRight = new Point { X = 100, Y = 100}},

new Rectangle {TopLeft = new Point { X = 5, Y = 5 },

BottomRight = new Point { X = 90, Y = 75}}

};

Questions

1. On which factors does program algorithm depends?
2. How to select optimal colleciton type?
3. Why it is recommended to use generic collections?
4. When to use generic classes?
5. When to use partial classes?