Pro C# 2005 and the .NET 2.0 Platform

Andrew Troelsen

Apress®

ISBN: 1-59059-419-3

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Technical Reviewer: Gavin Smyth
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Associate Publisher and Project Manager: Grace Wong
Copy Edit Manager: Nicole LeClerc
Copy Editors: Nicole LeClerc, Ami Knox
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor and Artist: Kinetic Publishing Services, LLC
Proofreader: Nancy Sixsmith
Indexers: Kevin Broccoli and Dan Mabbutt
Interior Designer: Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 10

Understanding Generics

With the release of .NET 2.0, the C# programming language has been enhanced to support a new feature of the CTS termed *generics*. Simply put, generics provide a way for programmers to define "placeholders" (formally termed *type parameters*) for method arguments and type definitions, which are specified at the time of invoking the generic method or creating the generic type.

To illustrate this new language feature, this chapter begins with an examination of the System.Collections.Generic namespace. Once you've seen generic support within the base class libraries, in the remainder of this chapter you'll examine how you can build your own generic members, classes, structures, interfaces, and delegates.

Revisiting the Boxing, Unboxing, and System.Object Relationship

To understand the benefits provided by generics, it is helpful to understand the "issues" programmers had without them. As you recall from Chapter 3, the .NET platform supports automatic conversion between stack-allocated and heap-allocated memory through *boxing* and *unboxing*. At first glance, this may seem like a rather uneventful language feature that is more academic than practical. In reality, the (un)boxing process is very helpful in that it allows us to assume everything can be treated as a System.Object, while the CLR takes care of the memory-related details on our behalf.

To review the boxing process, assume you have created a System.Collections.ArrayList to hold numeric (stack-allocated) data. Recall that the members of ArrayList are all prototyped to receive and return System.Object types. However, rather than forcing programmers to manually wrap the stack-based integer in a related object wrapper, the runtime will automatically do so via a boxing operation:

```
static void Main(string[] args)
{
     // Value types are automatically boxed when
     // passed to a member requesting an object.
     ArrayList myInts = new ArrayList();
     myInts.Add(10);
     Console.ReadLine();
}
```

If you wish to retrieve this value from the ArrayList object using the type indexer, you must unbox the heap-allocated object into a stack-allocated integer using a casting operation:

```
static void Main(string[] args)
{
...
     // Value is now unboxed...then reboxed!
```

```
        Console.WriteLine("Value of your int: {0}",
            (int)myInts[0]);
        Console.ReadLine();
    }
```

When the C# compiler transforms a boxing operation into terms of CIL code, you find the `box` opcode is used internally. Likewise, the unboxing operation is transformed into a CIL `unbox` operation. Here is the relevant CIL code for the previous `Main()` method (which can be viewed using `ildasm.exe`):

```
.method private hidebysig static void  Main(string[] args) cil managed
{
...
     box  [mscorlib]System.Int32
     callvirt  instance int32 [mscorlib]System.Collections.ArrayList::Add(object)
     pop
     ldstr  "Value of your int: {0}"
     ldloc.0
     ldc.i4.0
     callvirt  instance object [mscorlib]
        System.Collections.ArrayList::get_Item(int32)
     unbox  [mscorlib]System.Int32
     ldind.i4
     box  [mscorlib]System.Int32
     call  void [mscorlib]System.Console::WriteLine(string, object)
...
}
```

Note that the stack-allocated `System.Int32` is boxed prior to the call to `ArrayList.Add()` in order to pass in the required `System.Object`. Also note that the `System.Object` is unboxed back into a `System.Int32` once retrieved from the `ArrayList` using the type indexer (which maps to the hidden `get_Item()` method), only to be *boxed again* when it is passed to the `Console.WriteLine()` method.

The Problem with (Un)Boxing Operations

Although boxing and unboxing are very convenient from a programmer's point of view, this simplified approach to stack/heap memory transfer comes with the baggage of performance issues 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 the current `Main()` 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 are manipulated by your program on a somewhat regular basis.

Now consider the lack of type safety regarding unboxing operations. As you know, to unbox a value using the syntax of C#, you make use of the casting operator. However, the success or failure of a cast is not known until *runtime*. Therefore, if you attempt to unbox a value into the wrong data type, you receive an `InvalidCastException`:

```
static void Main(string[] args)
{
...
     // Ack!  Runtime exception!
     Console.WriteLine("Value of your int: {0}",
          (short)myInts[0]);
     Console.ReadLine();
}
```

In an ideal world, the C# compiler would be able to resolve illegal unboxing operations at compile time, rather than at runtime. On a related note, in a *really* ideal world, we could store sets of value types in a container that did not require boxing in the first place. .NET 2.0 generics are the solution to each of these issues. However, before we dive into the details of generics, let's see how programmers attempted to contend with these issues under .NET 1.x using strongly typed collections.

Type Safety and Strongly Typed Collections

In the world of .NET prior to version 2.0, programmers attempted to address type safety by building custom strongly typed collections. To illustrate, assume you wish to create a custom collection that can only contain objects of type `Person`:

```
public class Person
{
     // Made public for simplicity.
     public int currAge;
     public string fName, lName;

     public Person(){}
     public Person(string firstName, string lastName, int age)
     {
         currAge = age;
         fName = firstName;
         lName = lastName;
     }

     public override string ToString()
     {
         return string.Format("{0}, {1} is {2} years old",
             lName, fName, currAge);
     }
}
```

To build a person collection, you could define a `System.Collections.ArrayList` member variable within a class named `PeopleCollection` and configure all members to operate on strongly typed `Person` objects, rather than on generic `System.Objects`:

```
public class PeopleCollection : IEnumerable
{
    private ArrayList arPeople = new ArrayList();
    public PeopleCollection(){}

    // Cast for caller.
    public Person GetPerson(int pos)
    { return (Person)arPeople[pos]; }

    // Only insert Person types.
    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(); }
}
```

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 type:

```
static void Main(string[] args)
{
     Console.WriteLine("***** Custom Person Collection *****\n");
     PeopleCollection myPeople = new PeopleCollection();
     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);
     Console.ReadLine();
}
```

While custom collections do ensure type safety, this approach leaves you in a position where you must create a (almost identical) custom collection for each type you wish to contain. Thus, if you need a custom collection that will be able to operate only on classes deriving form the `Car` base class, you need to build a very similar type:

```
public class CarCollection : IEnumerable
{
    private ArrayList arCars = new ArrayList();
    public CarCollection(){}

    // Cast for caller.
    public Car GetCar(int pos)
    { return (Car) arCars[pos]; }

    // Only insert Car types.
    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(); }
}
```

As you may know from firsthand experience, the process of creating multiple strongly typed collections to account for various types is not only labor intensive, but also a nightmare to maintain. Generic collections allow us to delay the specification of the contained type until the time of creation. Don't fret about the syntactic details just yet, however. Consider the following code, which makes use of a generic class named `System.Collections.Generic.List<>` to create two type-safe container objects:

```
static void Main(string[] args)
{
     // Use the generic List type to hold only people.
     List<Person> morePeople = new List<Person>();
     morePeople.Add(new Person());

     // Use the generic List type to hold only cars.
     List<Car> moreCars = new List<Car>();

     // Compile-time error!
     moreCars.Add(new Person());
}
```

Boxing Issues and Strongly Typed Collections

Strongly typed collections are found throughout the .NET base class libraries and are very useful programming constructs. However, these custom containers do little to solve the issue of boxing penalties. Even if you were to create a custom collection named `IntCollection` that was constructed to operate only on `System.Int32` data types, you must allocate some type of object to hold the data (`System.Array`, `System.Collections.ArrayList`, etc.):

```
public class IntCollection : IEnumerable
{
    private ArrayList arInts = new ArrayList();
    public IntCollection() { }

    // Unbox for caller.
    public int GetInt(int pos)
    { return (int)arInts[pos]; }

    // Boxing operation!
    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 may choose to hold the integers (`System.Array`, `System.Collections.ArrayList`, etc.), you cannot escape the boxing dilemma using .NET 1.1. As you might guess, generics come to the rescue again. The following code leverages the `System.Collections.Generic.List<>` type to create a container of integers that does *not* incur any boxing or unboxing penalties when inserting or obtaining the value type:

```
 static void Main(string[] args)
{
     // No boxing!
     List<int> myInts = new List<int>();
     myInts.Add(5);

     // No unboxing!
     int i = myInts[0];
}
```

Just to prove the point, the previous `Main()` method results in the following CIL code (note the lack of any `box` or `unbox` opcodes):

```
.method private hidebysig static void Main(string[] args) cil managed
{
     .entrypoint
     .maxstack  2
     .locals init ([0] class [mscorlib]System.Collections.Generic.'List`1'<int32>
          myInts, [1] int32 i)
     newobj instance void class
        [mscorlib]System.Collections.Generic.'List`1'<int32>::.ctor()
     stloc.0
     ldloc.0
     ldc.i4.5
     callvirt instance void class [mscorlib]
          System.Collections.Generic.'List`1'<int32>::Add(!0)
     nop
     ldloc.0
     ldc.i4.0
     callvirt instance !0 class [mscorlib]
          System.Collections.Generic.'List`1'<int32>::get_Item(int32)
     stloc.1
     ret
}
```

So now that you have a better feel for the role generics can play under .NET 2.0, you're ready to dig into the details. To begin, allow me to formally introduce the `System.Collections.Generic` namespace.

■ **Source Code** The CustomNonGenericCollection project is located under the Chapter 10 directory.

The System.Collections.Generic Namespace

Generic types are found sprinkled throughout the .NET 2.0 base class libraries; however, the `System.Collections.Generic` namespace is chock full of them (as its name implies). Like its nongeneric counterpart (`System.Collections`), the `System.Collections.Generic` namespace contains numerous class and interface types that allow you to contain subitems in a variety of containers. Not surprisingly, the generic interfaces mimic the corresponding nongeneric types in the `System.Collections` namespace:

- `ICollection<T>`
- `IComparer<T>`
- `IDictionary<K, V>`
- `IEnumerable<T>`
- `IEnumerator<T>`
- `IList<T>`

■ **Note** By convention, generic types specify their placeholders using uppercase letters. Although any letter (or word) will do, typically `T` is used to represent types, `K` is used for keys, and `V` is used for values.

The `System.Collections.Generic` namespace also defines a number of classes that implement many of these key interfaces. Table 10-1 describes the core class types of this namespace, the interfaces they implement, and any corresponding type in the `System.Collections` namespace.

Table 10-1. *Classes of* `System.Collections.Generic`

Generic Class	Nongeneric Counterpart in `System.Collections`	Meaning in Life
`Collection<T>`	`CollectionBase`	The basis for a generic collection
`Comparer<T>`	`Comparer`	Compares two generic objects for equality
`Dictionary<K, V>`	`Hashtable`	A generic collection of name/value pairs
`List<T>`	`ArrayList`	A dynamically resizable list of items
`Queue<T>`	`Queue`	A generic implementation of a first-in, first-out (FIFO) list
`SortedDictionary<K, V>`	`SortedList`	A generic implementation of a sorted set of name/value pairs
`Stack<T>`	`Stack`	A generic implementation of a last-in, first-out (LIFO) list
`LinkedList<T>`	N/A	A generic implementation of a doubly linked list
`ReadOnlyCollection<T>`	`ReadOnlyCollectionBase`	A generic implementation of a set of read-only items

The `System.Collections.Generic` namespace also defines a number of “helper” 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.

As you can see from Table 10-1, many of the generic collection classes have a nongeneric counterpart in the `System.Collections` namespace (some of which are identically named). Given that Chapter 7 illustrated how to work with these nongeneric types, I will not provide a detailed examination of each generic counterpart. Rather, I’ll make use of `List<T>` to illustrate the process of working with generics. If you require details regarding other members of the `System.Collections.Generic` namespace, consult the .NET Framework 2.0 documentation.

Examining the List<T> Type

Like nongeneric classes, generic classes are heap-allocated objects, and therefore must be `new`-ed with any required constructor arguments. In addition, you are required to specify the type(s) to be substituted for the type parameter(s) defined by the generic type. For example, `System.Collections.Generic.List<T>` requires you to specify a single value that describes the type of item the `List<T>` will operate upon. Therefore, if you wish to create three `List<>` objects to contain integers and `SportsCar` and `Person` objects, you would write the following:

```
static void Main(string[] args)
{
    // Create a List containing integers.
    List<int> myInts = new List<int>();

    // Create a List containing SportsCar objects.
    List<SportsCar> myCars = new List<SportsCar>();

    // Create a List containing Person objects.
    List<Person> myPeople = new List<Person>();
}
```

At this point, you might wonder what exactly becomes of the specified placeholder value. If you were to make use of the Visual Studio 2005 Code Definition View window (see Chapter 2), you will find that the placeholder `T` is used throughout the definition of the `List<T>` type. Here is a partial listing (note the items in **bold**):

```
// A partial listing of the List<T> type.
namespace System.Collections.Generic
{
    public class List<T> :
        IList<T>, ICollection<T>, IEnumerable<T>,
        IList, ICollection, IEnumerable
    {
...
        public void Add(T item);
        public IList<T> AsReadOnly();
        public int BinarySearch(T item);
        public bool Contains(T item);
        public void CopyTo(T[] array);
        public int FindIndex(System.Predicate<T> match);
        public T FindLast(System.Predicate<T> match);
        public bool Remove(T item);
        public int RemoveAll(System.Predicate<T> match);
        public T[] ToArray();
        public bool TrueForAll(System.Predicate<T> match);
        public T this[int index] { get; set; }
..
    }
}
```

When you create a `List<T>` specifying `SportsCar` types, it is as if the `List<T>` type was really defined as so:

```
namespace System.Collections.Generic
{
    public class List<SportsCar> :
        IList<SportsCar>, ICollection<SportsCar>, IEnumerable<SportsCar>,
        IList, ICollection, IEnumerable
    {
...
        public void Add(SportsCar item);
        public IList<SportsCar> AsReadOnly();
        public int BinarySearch(SportsCar item);
        public bool Contains(SportsCar item);
        public void CopyTo(SportsCar[] array);
        public int FindIndex(System.Predicate<SportsCar> match);
        public SportsCar FindLast(System.Predicate<SportsCar> match);
        public bool Remove(SportsCar item);
```

```
        public int RemoveAll(System.Predicate<SportsCar> match);
        public SportsCar [] ToArray();
        public bool TrueForAll(System.Predicate<SportsCar> match);
        public SportsCar this[int index] { get; set; }
..
    }
}
```

Of course, when you create a generic List<T>, the compiler does not literally create a brand-new implementation of the List<T> type. Rather, it will address only the members of the generic type you actually invoke. To solidify this point, assume you exercise a List<T> of SportsCar objects as so:

```
static void Main(string[] args)
{
    // Exercise a List containing SportsCars
    List<SportsCar> myCars = new List<SportsCar>();
    myCars.Add(new SportsCar());
    Console.WriteLine("Your List contains {0} item(s).", myCars.Count);
}
```

If you examine the generated CIL code using ildasm.exe, you will find the following substitutions:

```
.method private hidebysig static void Main(string[] args) cil managed
{
     .entrypoint
     .maxstack  2
     .locals init ([0] class [mscorlib]System.Collections.Generic.'List`1'
          <class SportsCar> myCars)
     newobj instance void class [mscorlib]System.Collections.Generic.'List`1'
          <class SportsCar>::.ctor()
     stloc.0
     ldloc.0
     newobj instance void CollectionGenerics.SportsCar::.ctor()
     callvirt instance void class [mscorlib]System.Collections.Generic.'List`1'
          <class SportsCar>::Add(!0)
     nop
     ldstr "Your List contains {0} item(s)."
     ldloc.0
     callvirt  instance int32 class [mscorlib]System.Collections.Generic.'List`1'
          <class SportsCar>::get_Count()
     box [mscorlib]System.Int32
     call void [mscorlib]System.Console::WriteLine(string, object)
     nop
     ret
}
```

Now that you've looked at the process of working with generic types provided by the base class libraries, in the remainder of this chapter you'll examine how to create your own generic methods, types, and collections.

Creating Generic Methods

To learn how to incorporate generics into your own projects, you'll begin with a simple example of a generic swap routine. The goal of this example is to build a swap method that can operate on any possible data type (value-based or reference-based) using a single type parameter. Due to the nature of swapping algorithms, the incoming parameters will be sent by reference (via the C# ref keyword). Here is the full implementation:

```
// This method will swap any two items.
// as specified by the type parameter <T>.
static void Swap<T>(ref T a, ref T b)
{
     Console.WriteLine("You sent the Swap() method a {0}",
          typeof(T));
     T temp;
     temp = a;
     a = b;
     b = temp;
}
```

Notice how a generic method is defined by specifying the type parameter after the method name but before the parameter list. Here, you're stating that the `Swap()` method can operate on any two parameters of type `<T>`. Just to spice things up a bit, you're printing out the type name of the supplied placeholder to the console using the C# `typeof()` operator. Now ponder the following `Main()` method that swaps integer and string types:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Generics *****\n");
    // Swap 2 ints.
    int a = 10, b = 90;
    Console.WriteLine("Before swap: {0}, {1}", a, b);
    Swap<int>(ref a, ref b);
    Console.WriteLine("After swap: {0}, {1}", a, b);
    Console.WriteLine();

    // Swap 2 strings.
    string s1 = "Hello", s2 = "There";
    Console.WriteLine("Before swap: {0} {1}!", s1, s2);
    Swap<string>(ref s1, ref s2);
    Console.WriteLine("After swap: {0} {1}!", s1, s2);
    Console.ReadLine();
}
```

Omission of Type Parameters

When you invoke generic methods such as `Swap<T>`, you can optionally omit the type parameter if (and only if) the generic method requires arguments, as the compiler can infer the type parameter based on the member parameters. For example, you could swap two `System.Boolean` types as so:

```
// Compiler will infer System.Boolean.
bool b1 = true, b2 = false;
Console.WriteLine("Before swap: {0}, {1}", b1, b2);
Swap(ref b1, ref b2);
Console.WriteLine("After swap: {0}, {1}", b1, b2);
```

However, if you had another generic method named `DisplayBaseClass<T>` that did not take any incoming parameters, as follows:

```
static void DisplayBaseClass<T>()
{
     Console.WriteLine("Base class of {0} is: {1}.",
          typeof(T), typeof(T).BaseType);
}
```

you are required to supply the type parameter upon invocation:

```
static void Main(string[] args)
{
...
     // Must supply type parameter if
     // the method does not take params.
     DisplayBaseClass<int>();
     DisplayBaseClass<string>();

     // Compiler error! No params?  Must supply placeholder!
     // DisplayBaseClass();
...
}
```

Figure 10-1 shows the current output of this application.

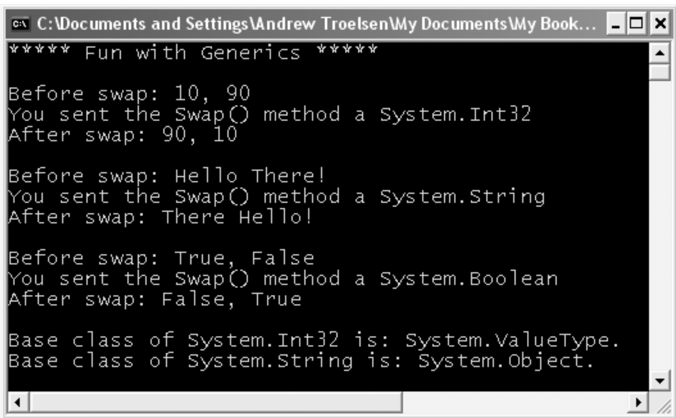

Figure 10-1. *Generic methods in action*

Currently, the generic `Swap<T>` and `DisplayBaseClass<T>` methods have been defined within the application object (i.e., the type defining the `Main()` method). If you would rather define these members in a new class type (`MyHelperClass`), you are free to do so:

```
public class MyHelperClass
{
     public static void Swap<T>(ref T a, ref T b)
     {
          Console.WriteLine("You sent the Swap() method a {0}",
               typeof(T));
          T temp;
          temp = a;
          a = b;
          b = temp;
     }

     public static void DisplayBaseClass<T>()
     {
          Console.WriteLine("Base class of {0} is: {1}.",
               typeof(T), typeof(T).BaseType);
     }
}
```

Notice that the MyHelperClass type is not in itself generic; rather, it defines two generic methods. In any case, now that the Swap<T> and DisplayBaseClass<T> methods have been scoped within a new class type, you will need to specify the type's name when invoking either member, for example:

```
MyHelperClass.Swap<int>(ref a, ref b);
```

Finally, generic methods do not need to be static. If Swap<T> and DisplayBaseClass<T> were instance level, you would simply make an instance of MyHelperClass and invoke them off the object variable:

```
MyHelperClass c = new MyHelperClass();
c.Swap<int>(ref a, ref b);
```

Creating Generic Structures (or Classes)

Now that you understand how to define and invoke generic methods, let's turn our attention to the construction of a generic structure (the process of building a generic class is identical). Assume you have built a flexible Point structure that supports a single type parameter representing the underlying storage for the (x, y) coordinates. The caller would then be able to create Point<T> types as so:

```
// Point using ints.
Point<int> p = new Point<int>(10, 10);

// Point using double.
Point<double> p2 = new Point<double>(5.4, 3.3);
```

Here is the complete definition of Point<T>, with analysis to follow:

```
// A generic Point structure.
public struct Point<T>
{
    // Generic state date.
    private T xPos;
    private T yPos;

    // Generic constructor.
    public Point(T xVal, T yVal)
    {
        xPos = xVal;
        yPos = yVal;
    }

    // Generic properties.
    public T X
    {
        get { return xPos; }
        set { xPos = value; }
    }

    public T Y
    {
        get { return yPos; }
        set { yPos = value; }
    }

    public override string ToString()
```

```
    {
        return string.Format("[{0}, {1}]", xPos, yPos);
    }

    // Reset fields to the default value of the
    // type parameter.
    public void ResetPoint()
    {
        xPos = default(T);
        yPos = default(T);
    }
}
```

The default Keyword in Generic Code

As you can see, `Point<T>` leverages its type parameter in the definition of the field data, constructor arguments, and property definitions. Notice that in addition to overriding `ToString()`, `Point<T>` defines a method named `ResetPoint()` that makes use of some new syntax:

```
// The 'default' keyword is overloaded in C# 2005.
// when used with generics, it represents the default
// value of a type parameter.
public void ResetPoint()
{
     xPos = default(T);
     yPos = default(T);
}
```

Under C# 2005, the `default` keyword has been given a dual identity. In addition to its use within a `switch` construct, it can be used to set a type parameter to its default value. This is clearly helpful given that a generic type does not know the actual placeholders up front and therefore cannot safely assume what the default value will be. The defaults for a type parameter are as follows:

- Numeric values have a default value of 0.
- Reference types have a default value of null.
- Fields of a structure are set to 0 (for value types) or null (for reference types).

For `Point<T>`, you could simply set `xPos` and `yPos` to 0 directly, given that it is safe to assume the caller will supply only numerical data. However, by using the `default(T)` syntax, you increase the overall flexibility of the generic type. In any case, you can now exercise the methods of `Point<T>` as so:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Generics *****\n");

    // Point using ints.
    Point<int> p = new Point<int>(10, 10);
    Console.WriteLine("p.ToString()={0}", p.ToString());
    p.ResetPoint();
    Console.WriteLine("p.ToString()={0}", p.ToString());
    Console.WriteLine();

    // Point using double.
    Point<double> p2 = new Point<double>(5.4, 3.3);
    Console.WriteLine("p2.ToString()={0}", p2.ToString());
    p2.ResetPoint();
```

```
        Console.WriteLine("p2.ToString()={0}", p2.ToString());
        Console.WriteLine();

        // Swap 2 Points.
        Point<int> pointA = new Point<int>(50, 40);
        Point<int> pointB = new Point<int>(543, 1);
        Console.WriteLine("Before swap: {0}, {1}", pointA, pointB);
        Swap<Point<int>>(ref pointA, ref pointB);
        Console.WriteLine("After swap: {0}, {1}", pointA, pointB);
        Console.ReadLine();
    }
```

Figure 10-2 shows the output.

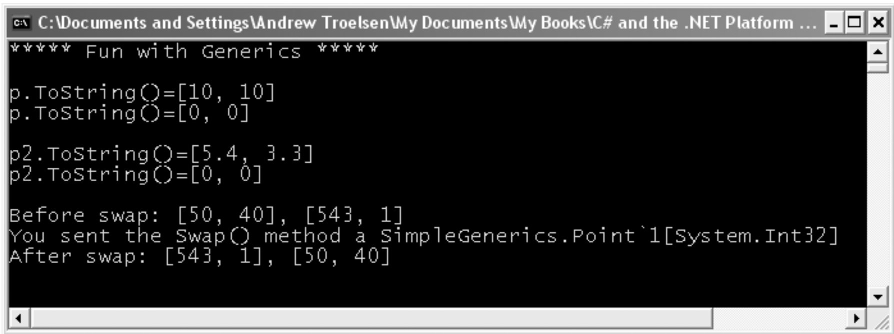

Figure 10-2. *Using the generic* `Point` *type*

■ **Source Code** The SimpleGenerics project is located under the Chapter 10 subdirectory.

Creating a Custom Generic Collection

As you have seen, the `System.Collections.Generic` namespace provides numerous types that allow you to create type-safe and efficient containers. Given the set of available choices, the chances are quite good that you will not need to build custom collection types when programming with .NET 2.0. Nevertheless, to illustrate how you could build a stylized generic container, the next task is to build a generic collection class named `CarCollection<T>`.

Like the nongeneric `CarCollection` created earlier in this chapter, this iteration will leverage an existing collection type to hold the subitems (a `List<>` in this case). As well, you will support `foreach` iteration by implementing the generic `IEnumerable<>` interface. Do note that `IEnumerable<>` extends the nongeneric `IEnumerable` interface; therefore, the compiler expects you to implement *two* versions of the `GetEnumerator()` method. Here is the update:

```
public class CarCollection<T> : IEnumerable<T>
{
    private List<T> arCars = new List<T>();

    public T GetCar(int pos)
    { return arCars[pos]; }
```

```
    public void AddCar(T c)
    { arCars.Add(c); }

    public void ClearCars()
    { arCars.Clear(); }

    public int Count
    { get { return arCars.Count; } }

    // IEnumerable<T> extends IEnumerable, therefore
    // we need to implement both versions of GetEnumerator().
    IEnumerator<T> IEnumerable<T>.GetEnumerator()
    { return arCars.GetEnumerator(); }
    IEnumerator IEnumerable.GetEnumerator()
    { return arCars.GetEnumerator(); }
}
```

You could make use of this updated `CarCollection<T>` as so:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Custom Generic Collection *****\n");

    // Make a collection of Cars.
    CarCollection<Car> myCars = new CarCollection<Car>();
    myCars.AddCar(new Car("Rusty", 20));
    myCars.AddCar(new Car("Zippy", 90));

    foreach (Car c in myCars)
    {
        Console.WriteLine("PetName: {0}, Speed: {1}",
        c.PetName, c.Speed);
    }
    Console.ReadLine();
}
```

Here you are creating a `CarCollection<T>` type that contains only `Car` types. Again, you could achieve a similar end result if you make use of the `List<T>` type directly. The major benefit at this point is the fact that you are free to add unique methods to the `CarCollection` that delegate the request to the internal `List<T>`.

Constraining Type Parameters Using where

Currently, the `CarCollection<T>` class does not buy you much beyond uniquely named public methods. Furthermore, an object user could create an instance of `CarCollection<T>` and specify a completely unrelated type parameter:

```
// This is syntactically correct, but confusing at best...
CarCollection<int> myInts = new CarCollection<int>();
myInts.AddCar(5);
myInts.AddCar(11);
```

To illustrate another form of generic abuse, assume that you have now created two new classes (`SportsCar` and `MiniVan`) that derive from the `Car` type:

```
public class SportsCar : Car
{
     public SportsCar(string p, int s)
          : base(p, s){}
     // Assume additional SportsCar methods.
}

public class MiniVan : Car
{
     public MiniVan(string p, int s)
          : base(p, s){}
     // Assume additional MiniVan methods.
}
```

Given the laws of inheritance, it is permissible to add a `MiniVan` or `SportsCar` type directly into a `CarCollection<T>` created with a type parameter of `Car`:

```
// CarCollection<Car> can hold any type deriving from Car.
CarCollection<Car> myCars = new CarCollection<Car>();
myInts.AddCar(new MiniVan("Family Truckster", 55));
myInts.AddCar(new SportsCar("Crusher", 40));
```

Although this is syntactically correct, what if you wished to update `CarCollection<T>` with a new public method named `PrintPetName()`? This seems simple enough—just access the correct item in the `List<T>` and invoke the `PetName` property:

```
// Error! System.Object does not have a
// property named PetName.
public void PrintPetName(int pos)
{
     Console.WriteLine(arCars[pos].PetName);
}
```

However, this will not compile, given that the true identity of `T` is not yet known, and you cannot say for certain if the item in the `List<T>` type has a `PetName` property. When a type parameter is not constrained in any way (as is the case here), the generic type is said to be *unbound*. By design, unbound type parameters are assumed to have only the members of `System.Object` (which clearly does not provide a `PetName` property).

You may try to trick the compiler by casting the item returned from the `List<T>`'s indexer method into a strongly typed `Car`, and invoking `PetName` from the returned object:

```
// Error!
// Cannot convert type 'T' to 'Car'
public void PrintPetName(int pos)
{
     Console.WriteLine(((Car)arCars[pos]).PetName);
}
```

This again does not compile, given that the compiler does not yet know the value of the type parameter `<T>` and cannot guarantee the cast would be legal.

To address such issues, .NET generics may be defined with optional constraints using the `where` keyword. As of .NET 2.0, generics may be constrained in the ways listed in Table 10-2.

Table 10-2. *Possible Constraints for Generic Type Parameters*

Generic Constraint	Meaning in Life
`where T : struct`	The type parameter `<T>` must have `System.ValueType` in its chain of inheritance.
`where T : class`	The type parameter `<T>` must *not* have `System.ValueType` in its chain of inheritance (e.g., `<T>` must be a reference type).
`where T : new()`	The type parameter `<T>` must have a default constructor. This is very helpful if your generic type must create an instance of the type parameter, as you cannot assume the format of custom constructors. Note that this constraint must be listed last on a multiconstrained type.
`where T :` *`NameOfBaseClass`*	The type parameter `<T>` must be derived from the class specified by *`NameOfBaseClass`*.
`where T :` *`NameOfInterface`*	The type parameter `<T>` must implement the interface specified by *`NameOfInterface`*.

When constraints are applied using the `where` keyword, the constraint list is placed after the generic type's base class and interface list. By way of a few concrete examples, ponder the following constraints of a generic class named `MyGenericClass`:

```
// Contained items must have a default ctor.
public class MyGenericClass<T> where T : new()
{...}

// Contained items must be a class implementing IDrawable
// and support a default ctor.
public class MyGenericClass<T> where T : class, IDrawable, new()
{...}

// MyGenericClass derives from MyBase and implements ISomeInterface,
// while the contained items must be structures.
public class MyGenericClass<T> : MyBase, ISomeInterface where T : struct
{...}
```

On a related note, if you are building a generic type that specifies multiple type parameters, you can specify a unique set of constraints for each:

```
// <K> must have a default ctor, while <T> must
// implement the generic IComparable interface.
public class MyGenericClass<K, T> where K : new()
    where T : IComparable<T>
{...}
```

If you wish to update `CarCollection<T>` to ensure that only `Car`-derived types can be placed within it, you could write the following:

```
public class CarCollection<T> : IEnumerable<T> where T : Car
{
...
    public void PrintPetName(int pos)
    {
        // Because all subitems must be in the Car family,
        // we can now directly call the PetName property.
        Console.WriteLine(arCars[pos].PetName);
    }
}
```

Notice that once you constrain CarCollection<T> such that it can contain only Car-derived types, the implementation of PrintPetName() is straightforward, given that the compiler now assumes <T> is a Car-derived type. Furthermore, if the specified type parameter is not Car-compatible, you are issued a compiler error:

```
// Compiler error!
CarCollection<int> myInts = new CarCollection<int>();
```

Do be aware that generic methods can also leverage the where keyword. For example, if you wish to ensure that only System.ValueType-derived types are passed into the Swap() method created previously in this chapter, update the code accordingly:

```
// This method will swap any Value types.
static void Swap<T>(ref T a, ref T b) where T : struct
{
...
}
```

Understand that if you were to constrain the Swap() method in this manner, you would no longer be able to swap string types (as they are reference types).

The Lack of Operator Constraints

When you are creating generic methods, it may come as a surprise to you that it is a *compiler error* to apply any C# operators (+, -, *, ==, etc.) on the type parameters. As an example, I am sure you could imagine the usefulness of a class that can Add(), Subtract(), Multiply(), and Divide() generic types:

```
// Compiler error!  Cannot apply
// operators to type parameters!
public class BasicMath<T>
{
     public T Add(T arg1, T arg2)
     { return arg1 + arg2; }
     public T Subtract(T arg1, T arg2)
     { return arg1 - arg2; }
     public T Multiply(T arg1, T arg2)
     { return arg1 * arg2; }
     public T Divide(T arg1, T arg2)
     { return arg1 / arg2; }
}
```

Sadly, the preceding BasicMath<T> class will not compile. While this may seem like a major restriction, you need to remember that generics *are* generic. Of course, the System.Int32 type can work just fine with the binary operators of C#. However, for the sake of argument, if <T> were a custom class or structure type, the compiler cannot assume it has overloaded the +, -, *, and / operators. Ideally, C# would allow a generic type to be constrained by supported operators, for example:

```
// Illustrative code only!
// This is not legal code under C# 2.0.
public class BasicMath<T> where T : operator +, operator -,
     operator *, operator /
{
     public T Add(T arg1, T arg2)
     { return arg1 + arg2; }
     public T Subtract(T arg1, T arg2)
     { return arg1 - arg2; }
     public T Multiply(T arg1, T arg2)
```

```
    { return arg1 * arg2; }
    public T Divide(T arg1, T arg2)
    { return arg1 / arg2; }
}
```

Alas, operator constraints are not supported under C# 2005.

Source Code The CustomGenericCollection project is located under the Chapter 10 subdirectory.

Creating Generic Base Classes

Before we examine generic interfaces, it is worth pointing out that generic classes can be the base class to other classes, and can therefore define any number of virtual or abstract methods. However, the derived types must abide by a few rules to ensure that the nature of the generic abstraction flows through. First of all, if a nongeneric class extends a generic class, the derived class must specify a type parameter:

```
// Assume you have created a custom
// generic list class.
public class MyList<T>
{
    private List<T> listOfData = new List<T>();
}

// Concrete types must specify the type
// parameter when deriving from a
// generic base class.
public class MyStringList : MyList<string>
{}
```

Furthermore, if the generic base class defines generic virtual or abstract methods, the derived type must override the generic methods using the specified type parameter:

```
// A generic class with a virtual method.
public class MyList<T>
{
    private List<T> listOfData = new List<T>();
    public virtual void PrintList(T data) { }
}

public class MyStringList : MyList<string>
{
    // Must substitute the type parameter used in the
    // parent class in derived methods.
    public override void PrintList(string data) { }
}
```

If the derived type is generic as well, the child class can (optionally) reuse the type placeholder in its definition. Be aware, however, that any constraints placed on the base class must be honored by the derived type, for example:

```
// Note that we now have a default constructor constraint.
public class MyList<T> where T : new()
{
    private List<T> listOfData = new List<T>();
```

```
    public virtual void PrintList(T data) { }
}

// Derived type must honor constraints.
public class MyReadOnlyList<T> : MyList<T> where T : new()
{
    public override void PrintList(T data) { }
}
```

Now, unless you plan to build your own generics library, the chances that you will need to build generic class hierarchies are slim to none. Nevertheless, C# does support generic inheritance.

Creating Generic Interfaces

As you saw earlier in the chapter during the examination of the `System.Collections.Generic` namespace, generic interfaces are also permissible (e.g., `IEnumerable<T>`). You are, of course, free to define your own generic interfaces (with or without constraints). Assume you wish to define an interface that can perform binary operations on a generic type parameter:

```
public interface IBinaryOperations<T>
{
    T Add(T arg1, T arg2);
    T Subtract(T arg1, T arg2);
    T Multiply(T arg1, T arg2);
    T Divide(T arg1, T arg2);
}
```

Of course, interfaces are more or less useless until they are implemented by a class or structure. When you implement a generic interface, the supporting type specifies the placeholder type:

```
public class BasicMath : IBinaryOperations<int>
{
    public int Add(int arg1, int arg2)
    { return arg1 + arg2; }

    public int Subtract(int arg1, int arg2)
    { return arg1 - arg2; }

    public int Multiply(int arg1, int arg2)
    { return arg1 * arg2; }

    public int Divide(int arg1, int arg2)
    { return arg1 / arg2; }
}
```

At this point, you make use of `BasicMath` as you would expect:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Generic Interfaces *****\n");
    BasicMath m = new BasicMath();
    Console.WriteLine("1 + 1 = {0}", m.Add(1, 1));
    Console.ReadLine();
}
```

If you would rather create a `BasicMath` class that operates on floating-point numbers, you could specify the type parameter as so:

```
public class BasicMath : IBinaryOperations<double>
{
     public double Add(double arg1, double arg2)
     { return arg1 + arg2; }
...
}
```

■ **Source Code** The GenericInterface project is located under the Chapter 10 subdirectory.

Creating Generic Delegates

Last but not least, .NET 2.0 does allow you to define generic delegate types. For example, assume you wish to define a delegate that can call any method returning `void` and receiving a single argument. If the argument in question may differ, you could model this using a type parameter. To illustrate, ponder the following code (notice the delegate targets are being registered using both "traditional" delegate syntax and method group conversion):

```
namespace GenericDelegate
{
     // This generic delegate can call any method
     // returning void and taking a single parameter.
     public delegate void MyGenericDelegate<T>(T arg);

     class Program
     {
          static void Main(string[] args)
          {
               Console.WriteLine("***** Generic Delegates *****\n");

               // Register target with 'traditional' delegate syntax.
               MyGenericDelegate<string> strTarget =
                    new MyGenericDelegate<string>(StringTarget);
               strTarget("Some string data");

               // Register target using method group conversion.
               MyGenericDelegate<int> intTarget = IntTarget;
               intTarget(9);
               Console.ReadLine();
          }

          static void StringTarget(string arg)
          {
               Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
          }

          static void IntTarget(int arg)
          {
               Console.WriteLine("++arg is: {0}", ++arg);
          }
     }
}
```

Notice that `MyGenericDelegate<T>` defines a single type parameter that represents the argument to pass to the delegate target. When creating an instance of this type, you are required to specify the value of the type parameter as well as the name of the method the delegate will invoke. Thus, if you specified a string type, you send a string value to the target method:

```
// Create an instance of MyGenericDelegate<T>
// with string as the type parameter.
MyGenericDelegate<string> strTarget =
     new MyGenericDelegate<string>(StringTarget);
strTarget("Some string data");
```

Given the format of the `strTarget` object, the `StringTarget()` method must now take a single string as a parameter:

```
static void StringTarget(string arg)
{
     Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
}
```

Simulating Generic Delegates Under .NET 1.1

As you can see, generic delegates offer a more flexible way to specify the method to be invoked. Under .NET 1.1, you could achieve a similar end result using a generic `System.Object`:

```
public delegate void MyDelegate(object arg);
```

Although this allows you to send any type of data to a delegate target, you do so without type safety and with possible boxing penalties. For instance, assume you have created two instances of `MyDelegate`, both of which point to the same method, `MyTarget`. Note the boxing/unboxing penalties as well as the inherent lack of type safety:

```
class Program
{
     static void Main(string[] args)
     {
...
          // Register target with 'traditional' delegate syntax.
          MyDelegate d = new MyDelegate(MyTarget);
          d("More string data");

          // Register target using method group conversion.
          MyDelegate d2 = MyTarget;
          d2(9);  // Boxing penalty.
...
     }

     // Due to a lack of type safety, we must
     // determine the underlying type before casting.
     static void MyTarget(object arg)
     {
          if(arg is int)
          {
               int i = (int)arg;  // Unboxing penalty.
               Console.WriteLine("++arg is: {0}", ++i);
          }
          if(arg is string)
```

```
        {
            string s = (string)arg;
            Console.WriteLine("arg in uppercase is: {0}", s.ToUpper());
        }
    }
}
```

When you send out a value type to the target site, the value is (of course) boxed and unboxed once received by the target method. As well, given that the incoming parameter could be anything at all, you must dynamically check the underlying type before casting. Using generic delegates, you can still obtain the desired flexibility without the "issues."

A Brief Word Regarding Nested Delegates

I'll wrap up this chapter by covering one final aspect regarding generic delegates. As you know, delegates may be nested within a class type to denote a tight association between the two reference types. If the nesting type is a generic, the nested delegate may leverage any type parameters in its definition:

```
// Nested generic delegates may access
// the type parameters of the nesting generic type.
public class MyList<T>
{
    private List<T> listOfData = new List<T>();
    public delegate void ListDelegate(T arg);
}
```

■ **Source Code** The GenericDelegate project is located under the Chapter 10 directory.

Summary

Generics can arguably be viewed as the major enhancement provided by C# 2005. As you have seen, a generic item allows you to specify "placeholders" (i.e., type parameters) that are specified at the time of creation (or invocation, in the case of generic methods). Essentially, generics provide a solution to the boxing and type-safety issues that plagued .NET 1.1 development.

While you will most often simply make use of the generic types provided in the .NET base class libraries, you are also able to create your own generic types. When you do so, you have the option of specifying any number of constraints to increase the level of type safety and ensure that you are performing operations on types of a "known quantity."