

Visual C# 2005

A Developer's Notebook™

Jesse Liberty

- Generics
- Iterators
- Anonymous Methods
- Data controls
- Web Personalization
- Better Windows Forms

CHAPTER 1

C# 2.0

In this chapter, you will learn about and use the new features in C# 2.0, including generics, iterators, anonymous methods, partial types, static classes, nullable types, and limiting access to properties, as well as delegate covariance and contravariance.

Probably the most exciting and most anticipated new feature in C# 2.0 is generics, which provide you with quick and easy type-safe collections. So, let's start there.

Create a Type-Safe List Using a Generic Collection

Type safety is the key to creating code that's easy to maintain. A type-safe language (and framework) finds bugs at compile time (reliably) rather than at runtime (usually after you've shipped the product!). The key weakness in C# 1.x was the absence of *generics*, which enable you to declare a general collection (for example, a stack or a list) that can accept members of any type yet will be type-safe at compile time.

In Version 1.x of the framework, nearly all the collections were declared to hold instances of `System.Object`, and because *everything* derives from `System.Object`, these collections could hold any type at all; that is, they were not type-safe.

Suppose, for example, you were creating a list of `Employee` objects in C# 1.x. To do so, you would use an `ArrayList`, which holds objects of the `System.Object` type. Adding new `Employees` to the collection was not a problem because `Employees` were derived from `System.Object`, but when you tried to retrieve an `Employee` from the `ArrayList`, all you would get back was an `Object` reference, which you would then have to cast:

```
Employee theEmployee = (Employee) myArrayList[1];
```

In this chapter:

- *Create a Type-Safe List Using a Generic Collection*
- *Create Your Own Generic Collection*
- *Implement the Collection Interfaces*
- *Enumerate Using Generic Iterators*
- *Implement GetEnumerator with Complex Data Structures*
- *Simplify Your Code with Anonymous Methods*
- *Hide Designer Code with Partial Types*
- *Create Static Classes*
- *Express Null Values with Nullable Types*

An even bigger problem, however, was that there was nothing to stop you from adding a string or some other type to the `ArrayList`. As long as you never needed to access the string, you would never note the errant type. Suppose, however, that you passed that `ArrayList` to a method that expected an `ArrayList` of `Employee` objects. When that method attempted to cast the `String` object to the `Employee` type at runtime, an exception would be thrown.

A final problem with .NET 1.x collections arose when you added value types to the collection. Value types had to be boxed on their way into the collection and explicitly unboxed on their way out.

.NET 2.0 eliminates all these problems with a new library of collections, which you will find in the `System.Collections.Generic` namespace. A *generic collection* is simply a collection that allows you to specify its member types when you declare it. Once declared, the compiler will allow only objects of that type to be added to your list. You define generic collections using special syntax; the syntax uses angle brackets to indicate variables that must be defined when an instance of the collection is declared.

There is no need to cast when you retrieve objects from a generic collection, and your code is safer, easier to maintain, and simpler to use than it is with untyped collections such as `ArrayList`.

With generic collections your code is type-safe, easier to maintain, and simpler to use.

How do I do that?

To get a feel for the new generic types in .NET 2.0, let's use the type-safe `List` class to create a list of employees. To execute this lab, open Visual Studio 2005, create a new C# Console application, and name it `CreateATypeSafeList`. Replace the code Visual Studio 2005 creates for you with the code in Example 1-1.

TIP

You must use the `System.Collections.Generic` namespace to use the generic types. By default Visual Studio 2005 adds this namespace to all projects.

Example 1-1. Creating a type-safe list

```
using System;
using System.Collections.Generic;

namespace CreateATypeSafeList
{
```

Example 1-1. Creating a type-safe list (continued)

```
// a class to store in the List
public class Employee
{
    private int empID;

    // constructor
    public Employee(int empID)
    {
        this.empID = empID;
    }

    // override the ToString method to
    // display this employee's id
    public override string ToString()
    {
        return empID.ToString();
    }
} // end class

// Test driver class
public class Program
{
    // entry point
    static void Main()
    {
        // Declare the type safe list (of Employee objects)
        List<Employee> empList = new List<Employee>();

        // Declare a second type safe list (of integers)
        List<int> intList = new List<int>();

        // populate the Lists
        for (int i = 0; i < 5; i++)
        {
            empList.Add(new Employee(i + 100));
            intList.Add(i * 5);
            // empList.Add(i * 5); // see "What About" section below
        }

        // print the integer list
        foreach (int i in intList)
        {
            Console.Write("{0} ", i.ToString());
        }

        Console.WriteLine("\n");

        // print the Employee List
        foreach (Employee employee in empList)
        {
            Console.Write("{0} ", employee.ToString());
        }
    }
}
```

Example 1-1. Creating a type-safe list (continued)

```
        Console.WriteLine("\n");  
    }  
}  
}
```

Output:

```
0 5 10 15 20  
100 101 102 103 104
```

TIP

All the source code for the labs in this chapter is available on my web site, <http://www.LibertyAssociates.com>. Click Books, and then scroll down to C# 2.0 Programmer's Notebook and click Source to save the source code to your computer.

Once unzipped, the source code is in chapter folders, and each lab folder is named with the namespace shown in the listing. For instance, for Example 1-1, the source is stored in *Chapter 1\CreateATypeSafeList*.

While you are at my site, you can also read the FAQ list and errata sheet and join a private support discussion forum.

What just happened?

This listing creates two classes: an `Employee` class to be held in the collection and the `Program` class created by Visual Studio 2005. It also uses the `List` class provided by the .NET Framework Class Library.

The `Employee` class contains a single private field (`empID`), a constructor, and an override of `ToString` to return the `empID` field as a string.

First you create an instance of `List` that will hold `Employee` objects. The type of `empList` is "List of Employee Objects" and is declared thus:

```
List<Employee> empList
```

When you see the definition `List<T>`, the `T` is a placeholder for the actual type you'll place in that list.

As always, `empList` is just a reference to the object you create on the heap using the `new` keyword. The `new` keyword expects you to invoke a constructor, which you do as follows:

```
new List<Employee>()
```

This creates an instance of “List of Employee Objects” on the heap, and the entire statement, put together, assigns a reference to that new object to `empList`:

```
List<Employee> empList = new List<Employee>();
```

TIP

This is just like writing:

```
Dog milo = new Dog();
```

in which you create an instance of `Dog` on the heap and assign it to the reference to `Dog`, `milo`.

In the next statement, you create a second `List`, this time of type “List of Integers”:

```
List<int> intList = new List<int>();
```

Now you are free to add integers to the list of integers, and `Employee` objects to the list of `Employee` objects. Once the lists are populated, you can iterate through each of them, using a `foreach` loop to display their contents in the console window:

```
foreach (Employee employee in empList)
{
    Console.WriteLine("{0} ", employee.ToString());
}
```

What about...

...if you try to add an integer to the list of `Employees`?

Try it. Start by uncommenting the following line in Example 1-1 and recompiling the program:

```
empList.Add(i * 5);
```

You’ll get a pair of compile errors:

```
Error 1      The best overloaded method match for 'System.Collections.
Generic.List<ListCollection.Employee>.Add(ListCollection.Employee)' has some
invalid arguments
Error 2      Argument '1': cannot convert from 'int' to
'ListCollection.Employee'
```

The information provided in these two compile errors enable you to determine that it is not legal to add an `int` to a collection of `Employee` objects because no implicit conversion or subtype relationship exists from one to the other.

You can store derived types in a type-safe collection. Thus, a collection of Employees will hold a Manager object, if Manager derives from Employee.

The good news is that this is a compile error, rather than a runtime error, which can sneak out the door and happen only when your client runs the application!

...what about other generic collections; are any available?

Other generic collections are available as well. For instance, the Stack and Queue collections, as well as the ICollection interface, are available in type-safe versions in .NET 2.0.

You use these just as you would List<T>. For example, to make a stack of Employee objects, you replace T in the Stack definition (Stack<T>) with the Employee type:

```
Stack<Employee> employeeStack = new Stack<Employee>();
```

Where can I learn more?

You can learn about all the .NET 2.0 generic classes in the MSDN topic titled "Commonly Used Collection Types," and you can read an article on the subject on O'Reilly's ONDotnet.com site at <http://www.ondotnet.com/pub/a/dotnet/2004/05/17/liberty.html>.

TIP

A document on my web site lists the links I mention in each lab so that you can copy and paste them into your browser. To get it, go to <http://www.LibertyAssociates.com>, click Books, scroll down to *Visual C# 2005: A Developer's Notebook*, and click Links.doc.

The next lab will show you how to create your own type-safe collections to supplement those provided by the Framework.

Create Your Own Generic Collection

.NET 2.0 provides a number of generic collection classes for lists, stacks, queues, dictionaries, etc. Typically, these are more than sufficient for your programming needs. But from time to time you might decide to create your own generic collection classes, such as when you want to provide those collections with problem-specific knowledge or capabilities that are simply not available in existing collections (for example, creating an optimized linked list, or adding generic collection semantics to another class you've created). It is a goal of the language and the Framework to empower you to create your own generic collection types.

From time to time you will decide to create your own generic collection classes.

How do I do that?

The easiest way to create a generic collection class is to create a specific collection (for example, one that holds integers) and then replace the type (for example, `int`) with the generic type (for example, `T`).

Thus:

```
private int data;
```

becomes:

```
private T data; // T is a generic Type Parameter
```

The generic type parameter (in this case, `T`) is defined by you when you create your collection class by placing the type parameter inside angle brackets (`< >`):

```
public class Node<T>
```

TIP

Many programmers use `T` for “type,” but Microsoft recommends you use longer, more descriptive type names (for example, `Node<DocumentType>`).

Now you have defined a new type, “Node of `T`,” which at runtime will become “Node of `int`” or node of any other type the compiler recognizes.

Example 1-2 creates a linked list of nodes of `T`, and then uses two instances of that generic list, each holding a different type of object.

Example 1-2. Creating your own generic collection

```
using System;

namespace GenericLinkedList
{
    public class Pilgrim
    {
        private string name;
        public Pilgrim(string name)
        {
            this.name = name;
        }
        public override string ToString()
        {
            return this.name;
        }
    }
    public class Node<T>
    {
        // member fields
    }
}
```

Example 1-2. Creating your own generic collection (continued)

```
private T data;
private Node<T> next = null;

// constructor
public Node(T data)
{
    this.data = data;
}

// properties
public T Data { get { return this.data; } }

public Node<T> Next
{
    get { return this.next; }
}

// methods
public void Append(Node<T> newNode)
{
    if (this.next == null)
    {
        this.next = newNode;
    }
    else
    {
        next.Append(newNode);
    }
}

public override string ToString()
{
    string output = data.ToString();

    if (next != null)
    {
        output += ", " + next.ToString();
    }

    return output;
}
} // end class

public class LinkedList<T>
{
    // member fields
    private Node<T> headNode = null;

    // properties

    // indexer
    public T this[int index]
```

Example 1-2. Creating your own generic collection (continued)

```
{
    get
    {
        int ctr = 0;
        Node<T> node = headNode;

        while (node != null && ctr <= index)
        {
            if (ctr == index)
            {
                return node.Data;
            }
            else
            {
                node = node.Next;
            }

            ++ctr;
        } // end while
        throw new ArgumentOutOfRangeException();
    } // end get
} // end indexer

// constructor
public LinkedList()
{
}

// methods
public void Add(T data)
{
    if (headNode == null)
    {
        headNode = new Node<T>(data);
    }
    else
    {
        headNode.Append(new Node<T>(data));
    }
}

public override string ToString()
{
    if (this.headNode != null)
    {
        return this.headNode.ToString();
    }
    else
    {
        return string.Empty;
    }
}
```

Example 1-2. Creating your own generic collection (continued)

```
}  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        LinkedList<int> myLinkedList = new LinkedList<int>();  
        for (int i = 0; i < 10; i++)  
        {  
            myLinkedList.Add(i);  
        }  
  
        Console.WriteLine("Integers: " + myLinkedList);  
        LinkedList<Pilgrim> pilgrims = new LinkedList<Pilgrim>();  
        pilgrims.Add(new Pilgrim("The Knight"));  
        pilgrims.Add(new Pilgrim("The Miller"));  
        pilgrims.Add(new Pilgrim("The Reeve"));  
        pilgrims.Add(new Pilgrim("The Cook"));  
  
        Console.WriteLine("Pilgrims: " + pilgrims);  
        Console.WriteLine("The fourth integer is " + myLinkedList[3]);  
        Pilgrim d = pilgrims[1];  
        Console.WriteLine("The second pilgrim is " + d);  
    }  
}
```

Output:

```
Integers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9  
Pilgrims: The Knight, The Miller, The Reeve, The Cook  
The fourth integer is 3  
The second pilgrim is The Miller
```

What just happened?

You just created a *generic* linked list; one that is type-safe for any type of object you hold in the collection. In fact, one way to create a linked list such as this is to start by creating a type-specific linked list. This simple example works by defining a generic linked list whose head node is initialized to null:

```
public class LinkedList<T>  
{  
    private Node<T> headNode = null;  
    ...  
}
```

When you add data to the linked list, a new node is created and if there is no head node, that new node becomes the head; otherwise, append is called on the head node.

Each node checks to see if its next field is null (and thus the current node is the end of the list). If so, the current node appends the new node; otherwise, it passes the new node to the next member in the list.

Notice that `LinkedList` is intentionally declared with the same generic type parameter as `Node`. Because they both use the same letter (`T`), the compiler knows that the type used to substitute for `T` in `LinkedList` will be the same type used to substitute for `T` in `Node`. This makes sense: a linked list of integers will hold nodes of integers.

Creating collections with generics is far easier than you might imagine. The simplest way to approach the problem is to build a type-specific collection, and then replace the type with the generic `<T>`.

What about...

...using generics with other code structures? Can I do that?

Sure; you also can use generics with structs, interfaces, delegates, and even methods.

Where can I learn more?

For more about creating your own class with generics, see the MSDN Help file, "Topic: Generics," as well as the article mentioned previously on O'Reilly's ONDotnet.com site at <http://www.ondotnet.com/pub/a/dotnet/2004/05/17/liberty.html>. Also, an open "Community Project to Develop the Best Public License Collection Classes for .NET" is available on the Wintellect site at <http://www.wintellect.com/powercollections/>.

Implement the Collection Interfaces

In addition to its generic collection classes, .NET 2.0 also provides a set of generic interfaces that enable you to create type-safe collections that have all the functionality of the earlier, nongeneric .NET 1.x collection types. You'll find these interfaces in the `System.Collections.Generic` namespace. The namespace also includes a number of related generic interfaces, such as `IComparable<T>`, which you can use to compare two objects of type `T` regardless of whether they are part of a collection.

You can create a sorted linked list by having each datatype stored in the list implement the `IComparable<T>` interface and by having your `Node` object be responsible for inserting each new `Node` at the correct (sorted) position in the linked list.

How do I do that?

Integer already implements `IComparable`; you can easily modify `Pilgrim` to do so as well. Modify the definition of the `Pilgrim` class to indicate that it implements the `IComparable<T>` interface:

```
public class Pilgrim : IComparable<Pilgrim>
```

Be sure to implement the `CompareTo` and the `Equals` methods that the interface requires. The objects these methods receive will be of type `Pilgrim` because this is a type-safe interface, not a “standard” interface that would pass in objects:

```
public int CompareTo(Pilgrim rhs)
public bool Equals(Pilgrim rhs)
```

You can constrain the datatypes your generic type accepts by using constraints.

All you need to do now is change the logic of adding a node. This time, instead of adding to the end of the list, you’ll insert the new node into the list where it belongs based on the implementation of the `CompareTo` method.

For this to work, you must ensure that the datatype held in the node implements `IComparable`. You accomplish this with a constraint using the keyword `where`:

```
public class Node<T> : IComparable<Node<T>> where T:IComparable<T>
```

This line of code declares a class `Node` of `T` that implements `IComparable` (of `Node` of `T`) and that is constrained to hold datatypes that implement `IComparable`. If you try to have your `Node` class hold an object that does not implement `IComparable`, you will receive an error message when you attempt to compile it.

You must be careful to return the new head of the list if the new node is “less than” the current head of the list, as shown in Example 1-3 (Changes from the previous example are highlighted.)

Example 1-3. Implementing generic interfaces

```
using System;
using System.Collections.Generic;

namespace ImplementingGenericInterfaces
{
    public class Pilgrim : IComparable<Pilgrim>
    {
        private string name;
        public Pilgrim(string name)
        {
            this.name = name;
        }
        public override string ToString()
```

Example 1-3. Implementing generic interfaces (continued)

```
{
    return this.name;
}

// implement the interface
public int CompareTo(Pilgrim rhs)
{
    return this.name.CompareTo(rhs.name);
}
public bool Equals(Pilgrim rhs)
{
    return this.name == rhs.name;
}
}

// node must implement IComparable of Node of T
// constrain Nodes to only take items that implement IComparable
// by using the where keyword.
public class Node<T> : IComparable<Node<T>> where T:IComparable<T>
{
    // member fields
    private T data;
    private Node<T> next = null;
    private Node<T> prev = null;

    // constructor
    public Node(T data)
    {
        this.data = data;
    }

    // properties
    public T Data { get { return this.data; } }

    public Node<T> Next
    {
        get { return this.next; }
    }

    public int CompareTo(Node<T> rhs)
    {
        // this works because of the constraint
        return data.CompareTo(rhs.data);
    }

    public bool Equals(Node<T> rhs)
    {
        return this.data.Equals(rhs.data);
    }
}
```

Example 1-3. Implementing generic interfaces (continued)

```
// methods
public Node<T> Add(Node<T> newNode)
{
    if (this.CompareTo(newNode) > 0) // goes before me
    {
        newNode.next = this; // new node points to me

        // if I have a previous, set it to point to
        // the new node as its next
        if (this.prev != null)
        {
            this.prev.next = newNode;
            newNode.prev = this.prev;
        }
        // set prev in current node to point to new node
        this.prev = newNode;
        // return the newNode in case it is the new head
        return newNode;
    }
    else // goes after me
    {
        // if I have a next, pass the new node along for comparison
        if (this.next != null)
        {
            this.next.Add(newNode);
        }
        // I don't have a next so set the new node
        // to be my next and set its prev to point to me.
        else
        {
            this.next = newNode;
            newNode.prev = this;
        }
        return this;
    }
}

public override string ToString()
{
    string output = data.ToString();

    if (next != null)
    {
        output += ", " + next.ToString();
    }

    return output;
}
// end class
```

Example 1-3. Implementing generic interfaces (continued)

```
public class SortedLinkedList<T> where T : IComparable<T>
{
    // member fields
    private Node<T> headNode = null;

    // properties

    // indexer
    public T this[int index]
    {
        get
        {
            int ctr = 0;
            Node<T> node = headNode;

            while (node != null && ctr <= index)
            {
                if (ctr == index)
                {
                    return node.Data;
                }
                else
                {
                    node = node.Next;
                }

                ++ctr;
            } // end while
            throw new ArgumentOutOfRangeException();
        } // end get
    } // end indexer

    // constructor
    public SortedLinkedList()
    {
    }

    // methods
    public void Add(T data)
    {
        if (headNode == null)
        {
            headNode = new Node<T>(data);
        }
        else
        {
            headNode = headNode.Add(new Node<T>(data));
        }
    }
}
```

Example 1-3. Implementing generic interfaces (continued)

```
public override string ToString()
{
    if (this.headNode != null)
    {
        return this.headNode.ToString();
    }
    else
    {
        return string.Empty;
    }
}
}

class Program
{
    // entry point
    static void Main(string[] args)
    {
        SortedLinkedList<int> mySortedLinkedList = new SortedLinkedList<int>();
        Random rand = new Random();
        Console.Write("Adding: ");
        for (int i = 0; i < 10; i++)
        {
            int nextInt = rand.Next(10);
            Console.Write("{0} ", nextInt);
            mySortedLinkedList.Add(nextInt);
        }

        SortedLinkedList<Pilgrim> pilgrims = new SortedLinkedList<Pilgrim>();
        pilgrims.Add(new Pilgrim("The Knight"));
        pilgrims.Add(new Pilgrim("The Miller"));
        pilgrims.Add(new Pilgrim("The Reeve"));
        pilgrims.Add(new Pilgrim("The Cook"));
        pilgrims.Add(new Pilgrim("The Man of Law"));

        Console.WriteLine("\nRetrieving collections...");

        DisplayList<int>("Integers", mySortedLinkedList);
        DisplayList<Pilgrim>("Pilgrims", pilgrims);
        //Console.WriteLine("Integers: " + mySortedLinkedList);
        //Console.WriteLine("Pilgrims: " + pilgrims);

        Console.WriteLine("The fourth integer is " + mySortedLinkedList[3]);
        Pilgrim d = pilgrims[2];
        Console.WriteLine("The third pilgrim is " + d);
        //      foreach (Pilgrim p in pilgrims)
        //      {
        //          Console.WriteLine("The pilgrim's name is " + p.ToString());
        //      }
    } // end main
}
```

Example 1-3. Implementing generic interfaces (continued)

```
        private static void DisplayList<T>(string intro, SortedLinkedList<T>
theList)
        where T : IComparable<T>
        {
            Console.WriteLine(intro + ": " + theList);
        }
    } // end class
} // end namespace
```

Output:

```
Adding: 2 8 2 5 1 7 2 8 5 5
Retrieving collections...
Integers: 1, 2, 2, 5, 7, 8, 8
Pilgrims: The Cook, The Knight, The Man of Law, The Miller, The Reeve
The fourth integer is 5
The third pilgrim is The Man of Law
```

What just happened?

The Pilgrim class changed just enough to implement the generic IComparable interface. The linked list didn't change at all, but the Node class did undergo some changes to support the sorted list.

First, the Node class was marked to implement IComparable and was constrained to hold only objects that themselves implement IComparable:

```
public class Node<T> : IComparable<Node<T>> where T:IComparable<T>
```

Second, Node added a reference to the previous node, in addition to the next node (making this a doubly linked list):

```
private Node<T> next = null;
private Node<T> prev = null;
```

The Node class must implement CompareTo and Equals. These are simple to implement because the constraint ensures that the data you are comparing also implements IComparable:

```
public int CompareTo(Node<T> rhs)
{
    // this works because of the constraint
    data.CompareTo(rhs.data);
}
```

What about...

...the IComparable requirement? Why did Pilgrim and Node require IComparable, but the linked list did not?

To understand this, it's important to note that both Pilgrims and Nodes must be compared; linked lists are not compared. Because the linked list is sorted by sorting its nodes, there is no need to compare two linked lists to see which one is "greater" than the other.

...what about passing generic types to a method; can I do that?

Yes, you can pass a generic type to a method, but only if the method is generic. In Example 1-3, you display the contents of the list of integers and the list of pilgrims with the following code:

```
Console.WriteLine("Integers: " + myLinkedList);  
Console.WriteLine("Pilgrims: " + pilgrims);
```

You are free to create a method to take these lists and display them (or manipulate them):

```
private void DisplayList<T>(string intro, LinkedList<T> theList) where T :  
    IComparable<T>  
{  
    Console.WriteLine(intro + ": " + theList);  
}
```

When you call the method, you supply the type:

```
DisplayList<int>("Integers", myLinkedList);  
DisplayList<Pilgrim>("Pilgrims", pilgrims);
```

TIP

The compiler is capable of type inference, so you can rewrite the preceding two lines as follows:

```
DisplayList("Integers", myLinkedList);  
DisplayList("Pilgrims", pilgrims);
```

Where can I learn more?

The MSDN Library covers the `Generic` namespace extensively. Search on `Systems.Collections.Generic`. Also, see my article on generics on O'Reilly's ONDotnet.com site at <http://www.ondotnet.com/pub/a/dotnet/2004/05/17/liberty.html>.

Enumerate Using Generic Iterators

In the previous examples you could not iterate over your list of Pilgrims using a foreach loop. As such, if you try to use the following code in Example 1-3:

```
foreach ( Pilgrim p in pilgrims )
{
    Console.WriteLine("The pilgrim's name is " + p.ToString());
}
```

you will receive the following error:

```
Error      1      foreach statement cannot operate on variables of type
'ImplementingGenericInterfaces.LinkedList <ImplementingGenericInterfaces.
Pilgrim>' because 'ImplementingGenericInterfaces.LinkedList
<ImplementingGenericInterfaces.Pilgrim>' does not contain a public
definition for 'GetEnumerator'
```

Adding iterators allows a client to iterate over your class using foreach.

In earlier versions of C#, implementing GetEnumerator was somewhat complicated and always tedious, but in C# 2.0 it is greatly simplified.

How do I do that?

To simplify the process of creating iterators, we'll begin by simplifying both the Pilgrim class and the Linked List class. The Linked List class will forgo all use of nodes and will store its contents in a fixed-size array (as the simplest type-safe container imaginable). Thus, it is a Linked List in name only! This will allow us to focus on the implementation of the IEnumerator interface, as shown in Example 1-4.

Example 1-4. Implementing IEnumerator, simplified

```
#region Using directives

using System;
using System.Collections.Generic;
using System.Text;

#endregion

namespace SimplifiedEnumerator
{
    // simplified Pilgrim
    public class Pilgrim
    {
        private string name;
        public Pilgrim(string name)
        {
            this.name = name;
        }
    }
}
```

Example 1-4. Implementing IEnumerator, simplified (continued)

```
    }
    public override string ToString()
    {
        return this.name;
    }
}

// simplified Linked List
class NotReallyALinkedList<T> : IEnumerable<T>
{
    // the entire linked list is stored in this
    // fixed size array
    T[] myArray;

    // constructor takes an array and stores the members
    public NotReallyALinkedList(T[] members)
    {
        myArray = members;
    }

    // implement the method for IEnumerable
    IEnumerator<T> IEnumerable<T>.GetEnumerator()
    {
        foreach (T t in this.myArray)
        {
            yield return t;
        }
    }
}

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
{
    throw new NotImplementedException();
}

}

class Program
{
    static void Main(string[] args)
    {
        // hardcode a string array of Pilgrim objects
        Pilgrim[] pilgrims = new Pilgrim[5];
        pilgrims[0] = new Pilgrim("The Knight");
        pilgrims[1] = new Pilgrim("The Miller");
        pilgrims[2] = new Pilgrim("The Reeve");
        pilgrims[3] = new Pilgrim("The Cook");
        pilgrims[4] = new Pilgrim("The Man Of Law");

        // create the linked list, pass in the array
        NotReallyALinkedList<Pilgrim> pilgrimCollection =
            new NotReallyALinkedList<Pilgrim>(pilgrims);
    }
}
```

Example 1-4. Implementing IEnumerator, simplified (continued)

```
// iterate through the linked list
foreach (Pilgrim p in pilgrimCollection)
{
    Console.WriteLine(p);
}
}
```

Output:

```
The Knight
The Miller
The Reeve
The Cook
The Man Of Law
```

What just happened?

In this example, the linked list is greatly simplified to keep its members in an array (in fact, it is not really a linked list at all). Because you've made your pseudo-LinkedList enumerable, however, now you can enumerate the Pilgrims in the pilgrims collection using a foreach loop.

When you write:

```
foreach (Pilgrim p in pilgrimCollection)
```

the C# compiler invokes the GetEnumerator method of the class. Internally, it looks more or less like this:

```
Enumerator e = pilgrimCollection.GetEnumerator();
while (e.MoveNext())
{
    Pilgrim p = e.Current;
}
```

Whenever you call foreach, the compiler internally translates it to a call to GetEnumerator.

As noted earlier, in C# 2.0 you do not have to worry about implementing MoveNext() or the current property. You need only use the new C# keyword yield.

TIP

You use yield only in iterator blocks. It either provides a value to the enumerator object or it signals the end of the iteration:

```
yield return expression;
yield break;
```

If you step into the foreach loop with the debugger, you'll find that each time through the foreach loop, the GetEnumerator method of the linked list is called, and each time through the next member in the array, it is yielded back to the calling foreach loop.

What about...

...implementing the GetEnumerator method on a more complex data structure, such as our original LinkedList?

That is shown in the next lab.

Where can I learn more?

For more on this subject, see the extensive article in MSDN titled "Iterators (C#)."

Implement GetEnumerator with Complex Data Structures

To add an iterator to your original LinkedList class, you'll implement IEnumerable<T> on both LinkedList and the Node class:

```
public class LinkedList<T> : IEnumerable<T>
    public class Node<T> : IComparable<Node<T>>, IEnumerable<Node<T>>
```

How do I do that?

As noted in the previous lab, the IEnumerable interface requires that you implement only one method, GetEnumerator, as shown in Example 1-5. (Changes from Example 1-3 are highlighted.)

Example 1-5. Enumerating through your linked list

```
using System;
using System.Collections.Generic;

namespace GenericEnumeration
{
    public class Pilgrim : IComparable<Pilgrim>
    {
        private string name;
        public Pilgrim(string name)
        {
            this.name = name;
        }
        public override string ToString()
```

Example 1-5. Enumerating through your linked list (continued)

```
{
    return this.name;
}

// implement the interface
public int CompareTo(Pilgrim rhs)
{
    return this.name.CompareTo(rhs.name);
}
public bool Equals(Pilgrim rhs)
{
    return this.name == rhs.name;
}
}

// node must implement IComparable of Node of T
// node now implements IEnumerable allowing its use in a foreach loop
public class Node<T> : IComparable<Node<T>>, IEnumerable<Node<T>> where T :
IComparable<T>
{
    // member fields
    private T data;
    private Node<T> next = null;
    private Node<T> prev = null;

    // constructor
    public Node(T data)
    {
        this.data = data;
    }

    // properties
    public T Data { get { return this.data; } }

    public Node<T> Next
    {
        get { return this.next; }
    }

    public int CompareTo(Node<T> rhs)
    {
        return data.CompareTo(rhs.data);
    }
    public bool Equals(Node<T> rhs)
    {
        return this.data.Equals(rhs.data);
    }
    // methods
    public Node<T> Add(Node<T> newNode)
    {
        if (this.CompareTo(newNode) > 0) // goes before me
```

Example 1-5. Enumerating through your linked list (continued)

```
{
    newNode.next = this; // new node points to me

    // if I have a previous, set it to point to
    // the new node as its next
    if (this.prev != null)
    {
        this.prev.next = newNode;
        newNode.prev = this.prev;
    }

    // set prev in current node to point to new node
    this.prev = newNode;

    // return the newNode in case it is the new head
    return newNode;
}
else // goes after me
{
    // if I have a next, pass the new node along for comparison
    if (this.next != null)
    {
        this.next.Add(newNode);
    }

    // I don't have a next so set the new node
    // to be my next and set its prev to point to me.
    else
    {
        this.next = newNode;
        newNode.prev = this;
    }

    return this;
}
}

public override string ToString()
{
    string output = data.ToString();

    if (next != null)
    {
        output += ", " + next.ToString();
    }

    return output;
}

// Method required by IEnumerable
IEnumerator<Node<T>> IEnumerable<Node<T>>.GetEnumerator()
```

Example 1-5. Enumerating through your linked list (continued)

```
{
    Node<T> nextNode = this;

    // iterate through all the nodes in the list
    // yielding each in turn
    do
    {
        Node<T> returnNode = nextNode;
        nextNode = nextNode.next;
        yield return returnNode;
    } while (nextNode != null);
}
System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
{
    throw new NotImplementedException();
}
} // end class

// implements IEnumerable so that you can use a LinkedList
// in a foreach loop
public class LinkedList<T> : IEnumerable<T> where T : IComparable<T>
{
    // member fields
    private Node<T> headNode = null;

    // properties
    // indexer
    public T this[int index]
    {
        get
        {
            int ctr = 0;
            Node<T> node = headNode;

            while (node != null && ctr <= index)
            {
                if (ctr == index)
                {
                    return node.Data;
                }
                else
                {
                    node = node.Next;
                }

                ++ctr;
            } // end while
            throw new ArgumentOutOfRangeException();
        } // end get
    } // end indexer
}
```

Example 1-5. Enumerating through your linked list (continued)

```
// constructor
public LinkedList()
{
}

// methods
public void Add(T data)
{
    if (headNode == null)
    {
        headNode = new Node<T>(data);
    }
    else
    {
        headNode = headNode.Add(new Node<T>(data));
    }
}
public override string ToString()
{
    if (this.headNode != null)
    {
        return this.headNode.ToString();
    }
    else
    {
        return string.Empty;
    }
}

// Implement IEnumerable required method
// iterate through the node (which is enumerable)
// and yield up the data from each node returned
IEnumerator<T> IEnumerable<T>.GetEnumerator()
{
    foreach (Node<T> node in this.headNode)
    {
        yield return node.Data;
    }
}
System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
{
    throw new NotImplementedException();
}
}

class Program
{
    private static void DisplayList<T>(string intro, LinkedList<T> theList)
    where T : IComparable<T>

```

Example 1-5. Enumerating through your linked list (continued)

```
{
    Console.WriteLine(intro + ": " + theList);
}

// entry point
static void Main(string[] args)
{
    LinkedList<Pilgrim> pilgrims = new LinkedList<Pilgrim>();
    pilgrims.Add(new Pilgrim("The Knight"));
    pilgrims.Add(new Pilgrim("The Miller"));
    pilgrims.Add(new Pilgrim("The Reeve"));
    pilgrims.Add(new Pilgrim("The Cook"));
    pilgrims.Add(new Pilgrim("The Man of Law"));

    DisplayList<Pilgrim>("Pilgrims", pilgrims);

    Console.WriteLine("Iterate through pilgrims...");

    // Now that the linked list is enumerable, we can put
    // it into a foreach loop
    foreach (Pilgrim p in pilgrims)
    {
        Console.WriteLine("The pilgrim's name is " + p.ToString());
    }
}
}
```

Output:

```
Pilgrims: The Cook, The Knight, The Man of Law, The Miller, The Reeve
Iterate through pilgrims...
The pilgrim's name is The Cook
The pilgrim's name is The Knight
The pilgrim's name is The Man of Law
The pilgrim's name is The Miller
The pilgrim's name is The Reeve
```

What just happened?

The linked list implements its enumerator to call `foreach` on the head node (which you can do because `Node` also implements `IEnumerable`). Then you yield the data object you get back from the node:

```
IEnumerator<T> IEnumerable<T>.GetEnumerator()
{
    foreach (Node<T> node in this.headNode)
    {
        yield return node.Data;
    }
}
```

When you use the yield statement, the C# compiler automatically generates a nested implementation of IEnumerator for you. It keeps its own state; you simply tell it which value to yield.

This gives Node the responsibility of iterating through the node list, which is accomplished, once again, using the yield statement in its own GetEnumerator method.

```
IEnumerator<Node<T>> IEnumerable<Node<T>>.GetEnumerator()  
{  
    Node<T> nextNode = this;  
    do  
    {  
        Node<T> returnNode = nextNode;  
        nextNode = nextNode.next;  
        yield return returnNode;  
    } while (nextNode != null);  
}
```

You initialize nextNode to the current node, and then you begin your do..while loop. This is guaranteed to run at least once. returnNode is set to nextNode, and then, once that is stashed away, nextNode is set to its next node (that is, the next node in the list). Then you yield returnNode. Each time through you are returning the next node in the list until nextNode is null, at which time you stop.

What about...

...the fact that in LinkedList you asked for each Node<T> in headNode? Is headNode a collection?

Actually, headNode is the top node in a linked list. Because Node implements IEnumerable, the node is acting like a collection. This isn't as arbitrary as it sounds because a node acts as a collection in the sense that it can give you the next node in the list. You could redesign the linked list to make the nodes "dumber" and the list itself "smarter," in which case it would be the list's job to iterate over each node in turn.

Where can I learn more?

You can learn more about the IEnumerable<T> interface in the MSDN Help files, "Topic: IEnumerable<T>."

Simplify Your Code with Anonymous Methods

Anonymous methods allow you to define method blocks inline. In general, you can use anonymous methods anywhere you can use a delegate. This can greatly simplify registering event handlers.

How do I do that?

To see how you can use an anonymous method, follow these steps:

1. Open a new Windows application in Visual Studio .NET 2005 and call it AnonymousMethods.
2. Drag two controls onto the default form: a label and a button. Don't bother renaming them.
3. Double-click the button. You will be taken to the code page, where you will enter the following code:

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = "Goodbye";
}
```

4. Run and test the application. Clicking the button changes the label text to Goodbye.

Anonymous methods allow you to pass a block of code as a parameter.

Great. No problem. But there is a bit of overhead here. You must register the delegate (Visual Studio 2005 did this for you), and you must write an entire method to handle the button click. Anonymous methods help simplify these tasks.

To see how this works, click the Show All Files button, as shown in Figure 1-1.

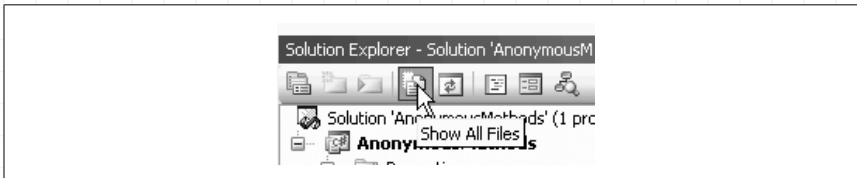


Figure 1-1. Show All Files button

Open *Form1.Designer.cs* and navigate to the delegate for button1.Click:

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

You can't replace this code without confusing the designer, but we will eliminate this line by returning to the form and clicking the lightning bolt in the Properties window, to go to the event handlers. Remove the event handler for the Click event.

If you return to *Form1.Designer.cs* you'll find that the button1.Click event handler is not registered!

Next, open *Form1.cs* and add the following line to the constructor, after the call to `InitializeComponent()`:

```
this.button1.Click += delegate { label1.Text = "Goodbye"; };
```

Now you are ready to delete (or comment out) the event handler method:

```
// private void button1_Click(object sender, EventArgs e)
// {
//     label1.Text = "Goodbye";
// }
```

Run the application. It should work exactly as it did originally.

Instead of registering the delegate which then invokes the method, the code for the delegate is placed inline in an *anonymous method*: that is, an inline, unnamed block of code.

What about...

...using anonymous methods in my own code?

No problem. Not only can you use anonymous methods when you initialize delegates, but also you can pass a block of code *anywhere* you might otherwise use a delegate.

...what happens if I reference local variables in my anonymous block?

Good question. This can cause quite a bit of confusion and is a natural trap to fall into, especially if you don't fully understand the consequences. C# allows local variables to be captured in the scope of the anonymous code block, and then they are accessed when the code block is executed. This can create some odd side effects, such as keeping objects around after they might otherwise have been collected.

...what about removing the handler for an event that I added with an anonymous delegate; how do I do that?

If you add an event handler with an anonymous delegate, you cannot remove it; therefore, I strongly recommend that you use anonymous delegates only for event handlers you expect to keep permanently attached.

You *can* use anonymous delegates for other requirements, such as implementing a `List.Find` method that takes, for example, a delegate describing the search criteria.

Where can I learn more?

On the MSDN web site, you'll find a good article touching on anonymous methods. Written by Juval Lowy, the article is titled "Create Elegant Code with Anonymous Methods, Iterators, and Partial Classes." Also, visit O'Reilly's ONDotnet.com site at <http://www.ondotnet.com/pub/a/dotnet/2004/04/05/csharpwhidbeypt1.html>.

Hide Designer Code with Partial Types

In previous versions of C# the entire definition for a class had to be in a single file. Now, using the partial keyword, you can split your class across more than one file. This provides two significant advantages:

- You can have different team members working on different parts of the class.
- Visual Studio 2005 can separate the designer-generated code from your own user code.

Using the partial keyword, you can split your class across more than one file.

How do I do that?

The easiest way to see partial types at work is to examine the previous example (AnonymousMethods). Examine the declaration of the class in *Form1.cs*:

```
partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        this.button1.Click += delegate { label1.Text = "Goodbye"; };
    }

    // private void button1_Click(object sender, EventArgs e)
    // {
    //     label1.Text = "Goodbye";
    // }
}
```

The partial keyword indicates that the code in this file does not necessarily represent the complete definition of this class. In fact, you saw earlier that the Visual Studio 2005 designer generated a second file, *Form1.Designer.cs*, which contains the rest of the definition:

```
namespace AnonymousMethods
{
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
    }
}
```

```

protected override void Dispose(bool disposing)
{
    if (disposing && (components != null))
    {
        components.Dispose();
    }
    base.Dispose(disposing);
}

#region Windows Form Designer generated code
/// Designer-generated initialization code
...
#endregion

private System.Windows.Forms.Label label1;
private System.Windows.Forms.Button button1;
}
}

```

Together, these two files completely define the Form1 class, but you are spared dealing with the designer-generated code unless you need to work with it. This makes for simpler and cleaner development.

There is some “fine print” you need to be aware of in regard to using partial classes:

- All partial type definitions must be modified with the `partial` keyword and must belong to the same namespace and the same module and assembly.
- The partial modifier can appear only before the class, interface, and struct keywords.
- Access modifiers (`public`, `private`, etc.) must match all the partial types of the same class.

What about...

...using partial classes in my own projects?

Microsoft suggests that partial classes can allow developers to work on different aspects of a class independently. It is still too early to see what best practices will emerge; I’m inclined to think that any class that is big enough to be divided in this way is big enough to be split into two (or more) classes. For now, the primary use of partial classes is to hide the cruft created by the designer.

TIP

Robert MacNeil reports in the PBS documentary “Do You Speak American?” that *cruft* is a neologism invented by surfers. However, the Online Computing Dictionary (<http://www.instantweb.com/D/dictionary/index.html>) reports that “This term is one of the oldest in the jargon and no one is sure of its etymology.” In any case, the Online Computing Dictionary defines *cruft* as “an unpleasant substance...excess; superfluous junk” and “the results of shoddy construction.”

Where can I learn more?

Developer.com provides a good article on partial types. Visit <http://www.developer.com/net/net/article.php/2232061> for more information.

Create Static Classes

In addition to declaring methods as being static, now you also can declare classes as being static.

The purpose of a static class is to provide a set of static utility methods scoped to the name of the class, much as you see done with the Convert class in the Framework Class Library.

In C# 2.0 you can declare an entire class as being static to signal that you've scoped a set of static utility methods to that class.

How do I do that?

To create a static class, just add the `static` keyword before the class name and make sure your static class meets the criteria described earlier for static members. Also, note that static classes have the following restrictions:

- They can contain only static members.
- It is not legal to instantiate a static class.
- All static classes are sealed (you cannot derive them from a static class).

In addition to these restrictions, a static class cannot contain a constructor. Example 1-6 shows the proper use of a static class.

Example 1-6. Using static classes

```
#region Using directives  
  
using System;  
  
#endregion
```

Example 1-6. Using static classes (continued)

```
namespace StaticClass
{
    public static class CupConversions
    {
        public static int CupToOz(int cups)
        {
            return cups * 8; // 8 ounces in a cup
        }
        public static double CupToPint(double cups)
        {
            return cups * 0.5; // 1 cup = 1/2 pint
        }

        public static double CupToMil(double cups)
        {
            return cups * 237; // 237 mil to 1 cup
        }

        public static double CupToPeck(double cups)
        {
            return cups / 32; // 8 quarts = 1 peck
        }

        public static double CupToBushel(double cups)
        {
            return cups / 128; // 4 pecks = 1 bushel
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("You might like to know that " +
                "1 cup liquid measure is equal to: ");
            Console.WriteLine(CupConversions.CupToOz(1) + " ounces");
            Console.WriteLine(CupConversions.CupToPint(1) + " pints");
            Console.WriteLine(CupConversions.CupToMil(1) + " milliliters");
            Console.WriteLine(CupConversions.CupToPeck(1) + " pecks");
            Console.WriteLine(CupConversions.CupToBushel(1) + " bushels");
        }
    }
}
```

Output:

```
You might like to know that 1 cup liquid measure is equal to:
8 ounces
0.5 pints
237 milliliters
0.03125 pecks
0.0078125 bushels
```

The Program class's main method makes calls on the static methods of the CupConversions class. Because CupConversions exists only to provide several helper methods, and no instance of CupConversions is ever needed, it is safe and clean to make CupConversions a static class.

What about...

...fields and properties? Can my static class have such members?

Yes, they can, but all the members (methods, fields, and properties) must be static.

Where can I learn more?

Eric Gunnerson has written an excellent article on static classes. You can find it in MSDN at <http://blogs.msdn.com/ericgu/archive/2004/04/13/112274.aspx>.

Express Null Values with Nullable Types

With new nullable types, you can assign value types a null value. This can be tremendously powerful, especially when working with databases where the value returned might be null; without nullable types you would have no way to express that an integer value is null, or that a Boolean is neither true nor false.

With nullable types, a value type such as bool or int can have the value null.

How do I do that?

You can declare a nullable type as follows:

```
System.Nullable<T> variable
```

Or, if you are within the scope of a generic type or method, you can write:

```
T? variable
```

Thus, you can create two Nullable integer variables with these lines of code:

```
System.Nullable<int> myNullableInt;  
int? myOtherNullableInt;
```

You can check whether a nullable variable is null in two ways as well. You can check like this:

```
if (myNullableInt.HasValue)
```

or like this:

```
if (myNullableInt != null)
```

Each will return true if the myNullableInt variable is not null, and false if it is, as illustrated in Example 1-7.

Example 1-7. Nullable types

```
using System;

namespace NullableTypes
{
    public class Dog
    {
        private int age;
        public Dog(int age)
        {
            this.age = age;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            int? myNullableInt = 25;
            double? myNullableDouble = 3.14159;
            bool? myNullableBool = null; // neither yes nor no

            // string? myNullableString = "Hello"; // not permitted
            // Dog? myNullableDog = new Dog(3); // not permitted

            if (myNullableInt.HasValue)
            {
                Console.WriteLine("myNullableInt is " + myNullableInt.Value);
            }
            else
            {
                Console.WriteLine("myNullableInt is undefined!");
            }

            if (myNullableDouble != null)
            {
                Console.WriteLine("myNullableDouble: " + myNullableDouble);
            }
            else
            {
                Console.WriteLine("myNullableDouble is undefined!");
            }

            if ( myNullableBool != null )
            {
                Console.WriteLine("myNullableBool: " + myNullableBool);
            }
        }
    }
}
```

Example 1-7. Nullable types (continued)

```
    }
    else
    {
        Console.WriteLine("myNullableBool is undefined!");
    }

    myNullableInt = null;    // assign null to the integer
    // int a = myNullableInt; // won't compile

    int b;
    try
    {
        b = (int)myNullableInt; // will throw an exception if x is null
        Console.WriteLine("b: " + b);
    }
    catch (System.Exception e)
    {
        Console.WriteLine("Exception! " + e.Message);
    }

    int c = myNullableInt ?? -1; // will assign -1 if x is null

    Console.WriteLine("c: {0}", c);

    // careful about your assumptions here
    // If either type is null, all comparisons evaluate false!
    if (myNullableInt >= c)
    {
        Console.WriteLine("myNullableInt is greater than or equal to c");
    }
    else
    {
        Console.WriteLine("Is myNullableInt less than c?");
    }
}
}
```

Output:

```
myNullableInt is 25
myNullableDouble: 3.14159
myNullableBool is undefined!
Exception! Nullable object must have a value.
c: -1
Is myNullableInt less than c?
```

What just happened?

Let's focus on the Main method. Five nullable types are created:

```
int? myNullableInt = 25;
double? myNullableDouble = 3.14159;
bool? myNullableBool = null; // neither yes nor no

// string? myNullableString = "Hello";
// Dog? myNullableDog = new Dog(3);
```

However, structs can be user-defined, and it's OK to use them as nullables.

The first three are perfectly valid, but you cannot create a nullable string or a nullable user-defined type (class), and thus they should be commented out.

We check whether each nullable type is null (or, equivalently, whether the HasValue property is true). If so, we print their value (or equivalently, we access their Value property).

After this the value null is assigned to myNullableInt:

```
myNullableInt = null;
```

The next line would like to declare an integer and initialize it with the value in myNullableInt, but this is not legal; there is no implicit conversion from a nullable int to a normal int. You can solve this in two ways. The first is with a cast:

```
b = (int)myNullableInt;
```

Comparison operators always return false if one value is null!

This will compile, but it will throw an exception at runtime if myNullableInt is null (which is why we've enclosed it in a try/catch block).

The second way to assign a nullable int to an int is to provide a default value to be used in case the nullable int is null:

```
int c = myNullableInt ?? -1;
```

This line reads as follows: "initialize int c with the value in myNullableInt unless myNullableInt is null, in which case initialize c to -1."

It turns out that all the comparison operators (>, <, <=, etc.) return false if either value is null. Thus, a true value can be trusted:

```
if (myNullableInt >= c)
{
    Console.WriteLine("myNullableInt is greater than or equal to c");
}
```

WARNING

Note, however, that == will return true if both arguments are null.

If the statement “myNullableInt is greater than or equal to c” displays, you know that myNullableInt is not null, nor is c, and that myNullableInt is greater than c. However, a false value cannot be trusted in the normal fashion:

```
else
{
    Console.WriteLine("Is myNullableInt less than c?");
}
```

This else clause can be reached if myNullableInt is less than c, but it can also be reached if either myNullableInt or c is null.

What about...

...Boolean null values? How are they compared to correspond to the SQL three-value Boolean type?

C# provides two new operators:

```
bool? operator &(bool? x, bool? y)
bool? operator |(bool? x, bool? y)
```

You can use these operators to create the truth table depicted in Table 1-1.

Table 1-1. Truth table for nullable Boolean operators

| If x is... | And y is... | x and y evaluate to... | x y evaluates to... |
|------------|-------------|------------------------|---------------------|
| True | True | True | True |
| True | False | False | True |
| True | Null | Null | True |
| False | True | False | True |
| False | False | False | False |
| False | Null | False | Null |
| Null | True | Null | True |
| Null | False | False | Null |
| Null | Null | Null | Null |

Where can I learn more?

The Visual C# Developer Center has a good article on nullable types. Visit <http://msdn.microsoft.com/vcsharp/2005/overview/language/nullabletypes/> for more information.

Access Objects in the Global Namespace

As in previous versions of C#, the `namespace` keyword is used to declare a scope. This lets you organize your code and prevents identifier collisions (for example, two different classes with the same name), especially when using third-party components.

The global namespace qualifier allows you to specify an identifier in the (default) global namespace rather than in the local namespace.

Any object that is not defined within a specific namespace is in the global namespace. Objects in the global namespace are available to objects in any other namespace. If a name collision occurs, however, you will need a way to specify that you want the object in the global namespace rather than in the local namespace.

How do I do that?

To access objects in the global namespace, you use the new global namespace qualifier (`global::`), as shown in Example 1-8.

Example 1-8. Using the global namespace

```
using System;

namespace GlobalNameSpace
{
    class Program
    {
        // create a nested System class that will provide
        // a set of utilities for interacting with the
        // underlying system (conflicts with System namespace)
        public class System
        {
        }

        static void Main(string[] args)
        {

            // flag indicates if we're in a console app
            // conflicts with Console in System namespace
            bool Console = true;

            int x = 5;

            // Console.WriteLine(x); // won't compile - conflict with Console
            // System.Console.WriteLine(x); // conflicts with System

            global::System.Console.WriteLine(x); // works great.
            global::System.Console.WriteLine(Console);
        }
    }
}
```

Example 1-8. Using the global namespace (continued)

```
    }  
}
```

Output:

```
5  
True
```

What just happened?

In this somewhat artificial example, you create a nested class that you named `System` and you created a local Boolean variable named `Console`. You have blocked access to the global `System` and `Console` identifiers, so neither of these lines will compile:

```
Console.WriteLine(x);  
System.Console.WriteLine(x);
```

To designate that you want to use the `System` object in the global namespace, you will use the global namespace qualifier:

```
global::System.Console.WriteLine(x);
```

Notice that in the final line, the global namespace qualifier is used to access the `System` and `Console` objects in the global namespace, and the unqualified `Console` identifier is used to access the local Boolean value:

```
global::System.Console.WriteLine(Console);
```

What about...

...other uses for the double-colon (`::`) operator?

The `::` operator is the namespace alias qualifier. It always appears between two identifiers:

```
identifierOne::identifierTwo
```

If `identifierOne` is the global namespace, this operator is used to find `identifierTwo` within the global namespace. But if `identifierOne` is any namespace other than the global namespace, the operator serves to restrict the lookup of `identifierOne`.

Where can I learn more?

The global namespace qualifier is mentioned in the MSDN article “Create Elegant Code with Anonymous Methods, Iterators, and Partial Classes” by Juval Lowy, available at <http://msdn.microsoft.com/msdnmag/issues/04/05/c20/>.

Limit Access Within Properties

Now you can restrict the accessibility level of the get and set accessors within a property.

It is now possible to restrict the accessibility level of the get and set accessors within a property using access modifiers. Usually you would restrict access to the set accessor and make the get accessor public.

How do I do that?

Add the access modifier to either the get or the set accessor within the property, as illustrated in Example 1-9.

Example 1-9. Limiting access to property accessors

```
#region Using directives

using System;
using System.Collections.Generic;
using System.Text;

#endregion

namespace LimitPropertyAccess
{
    public class Employee
    {
        private string name;
        public Employee(string name)
        {
            this.name = name;
        }
        public string Name
        {
            get { return name; }
            protected set { name = value; }
        }
        public virtual void ChangeName(string name)
        {
            // do work here to update many records
            Name = name; // access the private accessor
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Employee joe = new Employee("Joe");
        // other work here
        string whatName = joe.Name; // works
        // joe.Name = "Bob"; // doesn't compile
        joe.ChangeName("Bob"); // works
        Console.WriteLine("joe's name: {0}", joe.Name);
    }
}
```

Example 1-9. Limiting access to property accessors (continued)

```
    }  
  }  
}
```

Output:

```
joe's name: Bob
```

What just happened?

The design of your `Employee` class calls for the string `name` to be private. You anticipate that one day you'll want to move this to a database field, so you resolve that all access to this field will be through a property, `Name`.

Other classes are free to access `Name`, but you do not want them to set the name directly. If they are going to change the name field, they must do so through the `ChangeName` virtual method. You anticipate that derived classes will do different work when an employee changes his name.

Thus you want to provide access to the set accessor to this class's methods and to methods of any class that derives from this class, but not to other classes. You accomplish this by adding the restricting access modifier `protected` to the set accessor:

```
protected set { name = value; }
```

What about...

...restrictions on using access modifiers?

You cannot use these modifiers on interfaces or explicit interface member implementations. You can use them only if both `get` and `set` are included, and you can use them only on one or the other.

Further, the modifier must restrict access, not broaden it. Thus, you cannot make the property `protected` and then use a modifier to make `get` public.

Where can I learn more?

You can learn more about properties and access modifiers by reading the MSDN article on properties at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vclrfPropertiesPG.asp>.

Gain Flexibility with Delegate Covariance and Contravariance

Covariance allows you to encapsulate a method with a return type that is directly or indirectly derived from the delegate's return type.

Now it is legal to provide a delegate method with a return type that is derived (directly or indirectly) from the delegate's defined return type; this is called *covariance*. That is, if a delegate is defined to return a `Mammal`, it is legal to use that delegate to encapsulate a method that returns a `Dog` if `Dog` derives from `Mammal` and a `Retriever` if `Retriever` derives from `Dog` which derives from `Mammal`.

Similarly, now it is legal to provide a delegate method signature in which one or more of the parameters is derived from the type defined by the delegate. This is called *contravariance*. That is, if the delegate is defined to take a method whose parameter is a `Dog` you can use it to encapsulate a method that passes in a `Mammal` as a parameter, if `Dog` derives from `Mammal`.

Contravariance allows you to encapsulate a method with a parameter that is of a type from which the declared parameter is directly or indirectly derived.

How do I do that?

Covariance and contravariance give you more flexibility in the methods you encapsulate in delegates. The use of both covariance and contravariance is illustrated in Example 1-10.

Example 1-10. Using covariance and contravariance

```
#region Using directives
using System;
using System.Collections.Generic;
using System.Text;

#endregion

namespace CoAndContravariance
{
    class Mammal
    {
        public virtual Mammal ReturnsMammal()
        {
            Console.WriteLine("Returning a mammal");
            return this;
        }
    }

    class Dog : Mammal
    {
```

Example 1-10. Using covariance and contravariance (continued)

```
public Dog ReturnsDog()
{
    Console.WriteLine("Returning a dog");
    return this;
}

}

class Program
{
    public delegate Mammal theCovariantDelegate();
    public delegate void theContravariantDelegate(Dog theDog);

    private static void MyMethodThatTakesAMammal(Mammal theMammal)
    {
        Console.WriteLine("in My Method That Takes A Mammal");
    }

    private static void MyMethodThatTakesADog(Dog theDog)
    {
        Console.WriteLine("in My Method That Takes A Dog");
    }

    static void Main(string[] args)
    {
        Mammal m = new Mammal();
        Dog d = new Dog();

        theCovariantDelegate myCovariantDelegate =
            new theCovariantDelegate(m.ReturnsMammal);
        myCovariantDelegate();

        myCovariantDelegate =
            new theCovariantDelegate(d.ReturnsDog);
        myCovariantDelegate();

        theContravariantDelegate myContravariantDelegate =
            new theContravariantDelegate(MyMethodThatTakesADog);
        myContravariantDelegate(d);

        myContravariantDelegate =
            new theContravariantDelegate(MyMethodThatTakesAMammal);
        myContravariantDelegate(d);
    }
}
}
```

Output:

```
Returning a mammal
Returning a dog
in My Method That Takes A Dog
in My Method That Takes A Mammal
```

What just happened?

The Program class in Example 1-10 declares two delegates. The first is for a method that takes no parameters and returns a Mammal:

```
public delegate Mammal theCovariantMethod();
```

In the run method, you declare an instance of Mammal and an instance of Dog:

```
Mammal m = new Mammal();
Dog d = new Dog();
```

You are ready to create your first instance of theCovariantDelegate:

```
theCovariantDelegate myCovariantDelegate =
    new theCovariantDelegate(m.ReturnsMammal);
```

This matches the delegate signature (m.ReturnsMammal() is a method that takes no parameters and returns a Mammal), so you can invoke the method through the delegate:

```
myCovariantDelegate();
```

Now you use covariance to encapsulate a second method within the same delegate:

```
myCovariantDelegate =
    new theCovariantDelegate(d.ReturnsDog);
```

This time, however, you are passing in a method (d.ReturnsDog()) that does not return a Mammal; it returns a Dog that derives from Mammal:

```
public Dog ReturnsDog()
{
    Console.WriteLine("Returning a dog");
    return this;
}
```

That is covariance at work. To see contravariance, you declare a second delegate to encapsulate a method that returns null and takes a Dog as a parameter:

```
public delegate void theContravariantDelegate(Dog theDog);
```

Your first instantiation of this delegate encapsulates a method with the appropriate return value (void) and parameter (Dog):

```
myContravariantDelegate =
    new theContravariantDelegate(MyMethodThatTakesADog);
```

Earlier in the file you declared Dog to derive from Mammal.

You can invoke the method through the delegate. Your second use of this delegate, however, encapsulates a method that does not take a `Dog` as a parameter, but rather, takes a `Mammal` as a parameter:

```
theContravariantDelegate myContravariantDelegate =  
    new theContravariantDelegate(MyMethodThatTakesAMammal);
```

`MyMethodThatTakesAMammal` is defined to take a `Mammal`, not a `Dog`, as a parameter:

```
private void MyMethodThatTakesAMammal(Mammal theMammal)  
{  
    Console.WriteLine("in My Method That Takes A Mammal");  
}
```

Again, this works because a `Dog` is a `Mammal` and contravariance allows you to make this substitution.

Notice that with contravariance, you can pass in an object of the base type as a parameter where an object of the derived type is expected.

What about...

...contravariance? I get why with covariance I can return a `Dog` (a `Dog` is a `Mammal`), but why does contravariance work the other way? Shouldn't it accept a derived type when it expects a base type?

Contravariance is consistent with Postel's Law: "Be liberal in what you accept, and conservative in what you send." As the client, you must make sure that what you send to the method will work with that method, but as the implementer of the method you must be liberal and accept the `Dog` in any form, even as a reference to its base class.

Dr. Jonathan Bruce Postel (1943-1998), contributor to Internet Standards.

...what about reversing the usage of covariance and returning a base type where a derived type is expected? Can I do that?

No, it works in only one direction. You can, however, return a derived type where a base type is expected.

...what about reversing the usage of contravariance and passing in a derived type where a base type is expected?

You can't do that, either. Again, it works in only one direction. You can, however, pass in a base type where a derived type is expected.

Where can I learn more?

Internet newsgroups contain numerous discussions about the advantages and disadvantages of object-oriented languages that support covariance and contravariance. For details about using covariance and contravariance in C# programs, see the MSDN Help file pages for the two topics.