Microsoft Tech·Ed 2005

Hands-On Lab

Lab Manual

ILL-054 Creating and Using Iterators in the .NET Framework

Please do not remove this manual from the lab
The lab manual will be available from CommNet

Contents

Lab 1: Iterators Lab

Exercise 1 – Using C# 2.0 Iterators

In this lab you will add iterator support to a linked list, using the C# 2.0 iterators feature. Open the *Iterators.sln* solution in the Iterators lab folder. The solution contains a class library called LinkedList, and a test client in the form of a console application called ListClient.
The class `LinkedList` in the file *LinkedList.cs* is a simple linked list, which allows users to add pairs of keys and values. The pairs are stored in the generic type `Node<K,T>`, defined as:

```
//K is the key, T is the data item
internal class Node<K,T>
{
   public K Key;
   public T Item;
   public Node<K,T> NextNode;

   public Node()
   {
      Key      = default(K);
      Item     = default(T);
      NextNode = null;
   }
   public Node(K key,T item,Node<K,T> nextNode)
   {
      Key      = key;
      Item     = item;
      NextNode = nextNode;
   }
}
```

The linked list is defined as:

```
public class LinkedList<K,T>
{
   Node<K,T> m_Head;

   public LinkedList()
   {
      m_Head = new Node<K,T>();
   }
   public void AddHead(K key,T item)
   {
      Node<K,T> newNode = new Node<K,T>(key,item,m_Head.NextNode);
      m_Head.NextNode = newNode;
   }
}
```

The list has a reference to the first node in the list called `m_Head` (the head of the list). `m_Head` itself does not store any data. Its sole purpose is to always point to the first node where the data is stored (such a head is sometimes called a sentinel or an anchor). The list adds new pairs of keys and values by creating a new node and having the head reference `m_Head` point to the new node. The new node's reference to the next node is set to be the reference `m_Head` used to point to.

First, you will manually add iterator support to the list. Add the following manual implementation of `IEnumerable<T>` to the `LinkedList`:

```
public class LinkedList<K,T> : IEnumerable<T>
{
   IEnumerator<T> IEnumerable<T>.GetEnumerator()
   {
      return new ListEnumerator(this);
   }
   IEnumerator IEnumerable.GetEnumerator()
   {
      //Delegate the implementation to IEnumerator<T>
      IEnumerable<T> enumerable = this;
      return enumerable.GetEnumerator();
   }
   //Nested class definition
   class ListEnumerator : IEnumerator<T>
   {
      LinkedList<K,T> m_List;
      Node<K,T> m_Current;

      public ListEnumerator(LinkedList<K,T> list)
      {
         m_List = list;
         IEnumerator enumerator = this;
         enumerator.Reset();
      }

      void IDisposable.Dispose()
      {}

      void IEnumerator.Reset()
      {
         m_Current = m_List.m_Head;
      }

      bool IEnumerator.MoveNext()
      {
         m_Current = m_Current.NextNode;
         if(m_Current != null)
         {
            return true;
         }
         else
         {
            return false;
         }
      }
```

```
         T IEnumerator<T>.Current
         {
            get
            {
               if(m_Current == null)
               {
                  throw new InvalidOperationException();
               }
               return m_Current.Item;
            }
         }
         object IEnumerator.Current
         {
            get
            {
               IEnumerator<T> enumerator = this;
               return enumerator.Current;
            }
         }
      }
      //Rest of LinkedList
   }
```

The file *ListClient.cs* contains the class `ListClient`, with the following code to initialize the list and then iterate over it (using a `foreach` statement):

```
   class ListClient
   {
         static void Main(string[] args)
         {
         LinkedList<int,string> list = new LinkedList<int,string>();
         list.AddHead(1,"AAA");
         list.AddHead(2,"BBB");
         list.AddHead(3,"CCC");
         foreach(string item in list)
         {
            Console.WriteLine(item);
         }
         Console.ReadLine();//Wait for user to signal
      }
   }
```

Build and run the client, ensuring you get the following output:

```
   CCC
   BBB
   AAA
```

Now, add iterator support to the list using C# 2.0 iterators:
Remove the following code in bold **font** from `LinkedList` (the body of `IEnumerator<T>.GetEnumerator()` and the all the nested class definition):

```
public class LinkedList<K,T> : IEnumerable<T>
{
   IEnumerator<T> IEnumerable<T>.GetEnumerator()
   {
      return new ListEnumerator<T>(this);
   }
   IEnumerator IEnumerable.GetEnumerator()
   {
      //Delegate the implementation to IEnumerator<T>
      IEnumerable<T> enumerable = this;
      return enumerable.GetEnumerator();
   }
   //Nested class definition
   class ListEnumerator : IEnumerator<T>
   {
      LinkedList<K,T> m_List;
      Node<K,T> m_Current;

      public ListEnumerator(LinkedList<K,T> list)
      {...}
      void IDisposable.Dispose()
      {}
      void IEnumerator.Reset()
      {...}
      bool IEnumerator.MoveNext()
      {...}
      T IEnumerator<T>.Current
      {...}
      object IEnumerator.Current
      {...}
   }
   //Rest of LinkedList
}
```

Add the following **bold** font code to the `LinkedList` class:

```
public class LinkedList<K,T> : IEnumerable<T>
{
   IEnumerator<T> IEnumerable<T>.GetEnumerator()
   {
      Node<K,T> current = m_Head.NextNode;
      while(current != null)
      {
         yield return current.Item;
         current = current.NextNode;
      }
   }
   //Rest of LinkedList
}
```

Build and test the solution: make sure the client traces the same output.
The iterator as implemented returns only the data items themselves and the information about the associated keys is not present. You can modify the iterator to provide that support, by having it return a node object. First, make the `Node<K,T>` a public class:

```
public class Node<K,T>
{...}
```

Next, modify the `LinkedList` definition to implement an enumerable that yields node objects, by deriving from `IEnumerator<`**`Node<K,T>`**`>`.
Change `IEnumerable<`**`Node<K,T>`**`>.GetEnumerator()` to return `IEnumerator<`**`Node<K,T>`**`>`, and iterate over whole nodes, not the current item.

Change `IEnumerable.GetEnumerator()` to delegate to the implementation of `IEnumerable<`**`Node<K,T>`**`>` as well.

```
public class LinkedList<K,T> : IEnumerable<Node<K,T>>
{
   IEnumerator<Node<K,T>> IEnumerable<Node<K,T>>.GetEnumerator()
   {
      Node<K,T> current = m_Head.NextNode;
      while(current != null)
      {
         yield return current~~.Item~~;
         current = current.NextNode;
      }
   }
   IEnumerator IEnumerable.GetEnumerator()
   {
      //Delegate the implementation to IEnumerator<Node<K,T>>
      IEnumerable<Node<K,T>> enumerable = this;
      return enumerable.GetEnumerator();
   }
   //Rest of the LinkedList
}
```

Modify the client to trace both data items and their keys:

```
class ListClient
{
      static void Main(string[] args)
      {
      LinkedList<int,string> list = new LinkedList<int,string>();
      list.AddHead(1,"AAA");
      list.AddHead(2,"BBB");
      list.AddHead(3,"CCC");

      foreach(Node<int,string> node in list)
      {
         Console.WriteLine("Key is {0}, Item is {1}",node.Key,node.Item);
      }
      //Wait for user to exit
      Console.ReadLine();
   }
}
```

Build and run the client, ensuring you get the following output:

```
Key is 3, Item is CCC
Key is 2, Item is BBB
Key is 1, Item is AAA
```

Resources:

Create Elegant Code with Anonymous Methods, Iterators, and Partial Classes
By Juval Lowy, MSDN® Magazine, May 2004
Generics FAQ
By Juval Lowy, MSDN June 2005

Programming .NET Components 2nd Edition

By Juval Lowy, O'Reilly 2005

The IDesign Advanced .NET Master Class

Authored by Juval Lowy, this world acclaimed intense class covers everything, from .NET essentials to the application frameworks and system programming.
More at www.idesign.net