Hands-On Lab

Lab Manual

ILL-061 Creating and Using Generics in the .NET Framework 2.0

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

Contents

Lab 1: Generics Lab

Exercise 1 –Generic Stack

In this exercise, you will develop a generic stack, and experience the benefits of generics.

Task 1 – Develop a Generic Stack

Open the *Generics.sln* solution in the Generics lab folder. The solution is a simple console application that uses a stack. Open the *ObjectStack.cs* file. It contains an object-based stack:

```
public class Stack
{
   readonly int m_Size;
   int m_StackPointer = 0;
   object[] m_Items;
   public Stack():this(100)
   {}
   public Stack(int size)
   {
      m_Size = size;
      m_Items = new object[m_Size];
   }
   public void Push(object item)
   {
      if(m_StackPointer >= m_Size)
         throw new StackOverflowException();
      m_Items[m_StackPointer] = item;
      m_StackPointer++;
   }
   public object Pop()
   {
      m_StackPointer--;
      if(m_StackPointer >= 0)
      {
         return m_Items[m_StackPointer];
      }
      else
      {
         m_StackPointer = 0;
         throw new InvalidOperationException("Cannot pop an empty stack");
      }
   }
}
```

The `Main()` method uses the object-based stack:

```
static void Main(string[] args)
{
   Stack stack = new Stack();
   stack.Push(1);
   stack.Push(2);
   int number = (int)stack.Pop();
   Debug.Assert(number == 2);
   Console.WriteLine(number);
   Console.ReadLine();
}
```

Modify the object-based stack to a generic stack. First, save the file under the name *GenericStack.cs*. Next, change the object-based stack to a generic stack, by adding `<T>` to the class definition, and by replacing the use of an `object` with a type parameter `T`:

```
public class Stack<T>
{
   readonly int m_Size;
   int m_StackPointer = 0;
   T[] m_Items;
   public Stack():this(100)
   {}
   public Stack(int size)
   {
      m_Size = size;
      m_Items = new T[m_Size];
   }
   public void Push(T item)
   {
      if(m_StackPointer >= m_Size)
         throw new StackOverflowException();
      m_Items[m_StackPointer] = item;
      m_StackPointer++;
   }
   public T Pop()
   {
      m_StackPointer--;
      if(m_StackPointer >= 0)
      {
         return m_Items[m_StackPointer];
      }
      else
      {
         m_StackPointer = 0;
         throw new InvalidOperationException("Cannot pop an empty stack");
      }
   }
}
```

Modify the `Main()` method to use the generic stack:

```
static void Main(string[] args)
{
   Stack<int> stack = new Stack<int>();
   stack.Push(1);
   stack.Push(2);
   int number = stack.Pop();
   Debug.Assert(number == 2);
}
```

Build and test to make sure all is well. Try using `Stack<T>` with integers and strings, to experience first-hand the productivity benefits of generics.

Task 2 – Testing the Generic Stack

Next, you will test the performance advantage of the generic stack you built in the previous step. Open the solution *GenericsPerfs.sln*. The solution is a micro-benchmark application, which you will use to execute a stack in a tight loop. It will lets you experiment with value and reference types on an Object-based stack and a generic stack, as well as changing the number of loop iterations to see the effect generics have on performance.

Copy the files *ObjectStack.cs* and *GenericStack.cs* to the GenericsPerfs solution folder by adding the files to the project: right-click on the GenericsPerfs solution in Microsoft® Visual Studio® 2005, and select Add|Add Existing Item… to add *ObjectStack.cs* and *GenericStack.cs* to the solution. Build the solution to make sure they were added properly. The performance tester uses a delegate to invoke different tests, using the delegate `TestMethod`, defined as:

```
delegate void TestMethod();
```

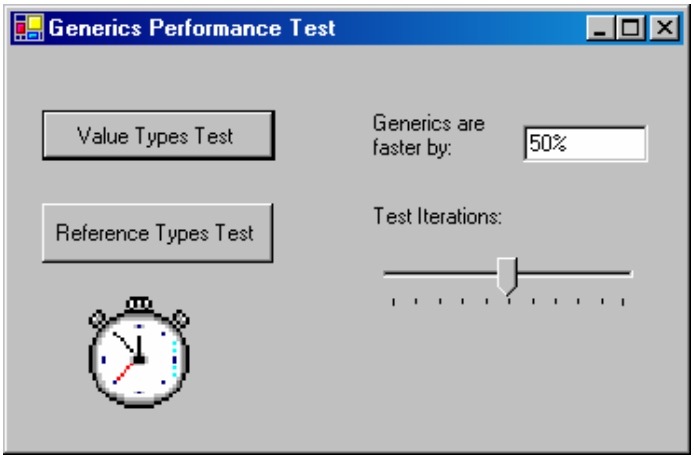
The test routine `GetTestTime()` uses the `Stopwatch` class to measure the test time. It invokes the delegate, and returns the test duration in milliseconds:

```
long GetTestTime(TestMethod testMethod)
{
   Stopwatch stopper = new Stopwatch();

   stopper.Start();
   testMethod();
   stopper.Stop();

   return stopper.ElapsedMilliseconds;
}
```

The test client is the Microsoft Windows® Forms class `TestClient`. It has two buttons, allowing you to test value and reference types, as well as controlling the number of test iterations:



The method `OnValueTest()` handles the click event for the Value Type Test button. It calls `GetTestTime()`, invoking it once with the object-based stack (via the `TestValueBoxed()` method) and once and once with the generic stack, (via the `TestValueGeneric()` method):

```
void OnValueTest(object sender,EventArgs e)
{
   float boxedTime = GetTestTime(TestValueBoxed);
   float genericTime = GetTestTime(TestValueGeneric);

   float perf = 100 * (1-(genericTime / boxedTime));

   m_TextResultBox.Text = Math.Round(perf) + "%";
}
```

`OnValueTest()` then calculates the difference in performance and updates the display.
In a similar manner, the `OnReferenceTest()` method handles the `Click` event for the Reference Type Test button, using reference types:

```
void OnReferenceTest(object sender,EventArgs e)
{
   float boxedTime = GetTestTime(TestReference);
   float genericTime = GetTestTime(TestReferenceGeneric);

   float perf = 100 * (1-(genericTime / boxedTime));

   m_TextResultBox.Text = Math.Round(perf) + "%";
}
```

Next, you need to add the code for the actual four test methods. Add the following code to the test methods:

```
void TestValueBoxed()
{
   Stack stack = new Stack();

   long temp = 0;
   long iteration = Count * m_IterationBar.Value;

   for(long i = 0;i < iteration;i ++)
   {
      stack.Push(i);
      temp = (long)stack.Pop();
   }
}
```

```
void TestValueGeneric()
{
   Stack<long> stack = new Stack<long>();

   long temp = 0;
   long iteration = Count * m_IterationBar.Value;

   for(long i = 0;i < iteration;i ++)
   {
      stack.Push(i);
      temp = stack.Pop();
   }
}
void TestReference()
{
   Stack stack = new Stack();

   string temp = String.Empty;
   long iteration = Count * m_IterationBar.Value;

   for(long i = 0;i < iteration;i ++)
   {
      stack.Push("AAAAA");
      temp = (string)stack.Pop();
   }
}
void TestReferenceGeneric()
{
   Stack<string> stack = new Stack<string>();

   string temp = String.Empty;
   long iteration = Count * m_IterationBar.Value;

   for(long i = 0;i < iteration;i ++)
   {
      stack.Push("AAAAA");
      temp = stack.Pop();
   }
}
```

Build and run the performance test client. Experiment with different number of test iteration to see the effect of generics both on value types (about 50% performance improvement) and on reference types (10% performance improvement).

Resources:

An Introduction to C# Generics
By Juval Lowy, MSDN® November 2003, Updated January 2005
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