CIS 365 Lab07: Generic Justification

Adam Lewis

<2015-05-18 Mon>

Contents

1 Objectives

1. Introduce the concepts of the C# Generics (template data types)
2. Consider how we can write code to be as general as possible
3. Begin a discussion about interfaces

2 Background

The concept of *generic programming* defines a programming style where we implement algorithms in terms of type parameters where we defer the exact type used until the *point-of-use* rather than *point-of-definition*. There are many algorithms where the exact type of the variables in the algorithm isn't significant; that is, we can write the algorithm in a generic fashion that applies to all types and make the decision about the type when we use the algorithm.

In this lab we will implement the Bubble Sort algorithm and consider how we can use that algorithm with any data type.

3 Instructions

Start a new C# Console application in C#. Name your app as *BubbleSortTestApp* and name your class as being *BubbleSortTestClass*.

3.1 Setup the data structures needed by the simulation

Let's define a set of local arrays in the **Main()** function that we will use as input into our sorting functions:

```
int [] anArrayOfIntegers = { 9, 7, 4, 10, 48, 1, 3, 5, 7 };
double [] anArrayOfDoubles = { 9.0, 7.0, 3.5, 8.9, 8.3, 3.14, 42.0 };
string[] anArrayOfStrings = { "Tom", "Dick", "Harry", "James", "Sam" };
```

3.2 Our first run at sorting

Add the following code to your **Main()** function:

```
BubbleSortTestClass bSortTestObj = new BubbleSortTestClass();
bSortTestObj.BubbleSort(ref anArrayOfIntegers);
bSortTestObj.PrintArray(anArrayOfIntegers);
```

Note that what we are doing here is creating a new reference to our test class and then working with that instance (as opposed to implementing everything as static class methods as we have done in past labs.

Now we have to implement the code to sort and print the arrays. Add the following functions to your class:

```
// BubbleSort(ref int[])
// Assumes: Existence of an array.
// Result: Nothing returned, but the array is sorted as a side effect
// NOTE: Overloaded method.
public void BubbleSort(ref int[] arr)
{
    int i;
    int j;
    for (i = (arr.Length - 1); i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (arr[j - 1] > arr[j])
            {
                int temp = arr[j - 1];
                arr[j - 1] = arr[j];
                arr[j] = temp;
            }
        }
    }
}

// PrintArray(int [])
// Assumes: An array has been initialized
// Result: Outputs the array
// NOTE: Overloaded method
public void PrintArray(int [] arr) {
    foreach (int elem in arr) {
        Console.Write("{0} ", elem);
    }
    Console.WriteLine();
}
```

The function **BubbleSort()** implements the typical multi-part selection that you see in the Bubble Sort algorithm. Save, build, and execute your program

3.3 But what do I do for the array of doubles

Now we have a problem: I want to be able to sort and print a different data type. One approach would be for us to overload the **BubbleSort()** and **PrintArray()** functions. Add the following to your class:

```
// BubuleSort(ref double[])
 // Assumes: Existence of an array.
 // Result: Nothing returned, but the array is sorted as a side effect
 // NOTE: Overloaded method.
 public void BubbleSort(ref double[] arr)
 {
     int i;
```

```
    int j;
    for (i = ( arr.Length - 1 ); i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (arr[j - 1] > arr[j])
            {
                double temp = arr[j - 1];
                arr[j - 1] = arr[j];
                arr[j] = temp;
            }
        }
    }
}

// PrintArray(double [])
// Assumes: An array has been initialized
// Result: Outputs the array
// NOTE: Overloaded method
public void PrintArray(double [] arr) {
    foreach (double elem in arr) {
        Console.Write("{0} ", elem);
    }
    Console.WriteLine();
}
```

Note that the only difference between the the methods are the data type of the parameters and the variable **temp**. This works, as you can see if you add the following to your **Main()** function:

```
bSortTestObj.BubbleSort(ref anArrayOfDoubles);
bSortTestObj.PrintArray(anArrayOfDoubles);
```

3.4 Is there a way to avoid all of this copying?

The C# concept of "Generics" give us a way to avoid all of this duplication (What might be some of the problems caused by duplicating the code?). We can "parameterize" this code by putting in a placeholder for the data types. The compiler is smart enough to understand that it needs to replace these parameters with the correct datatypes based upon how the functions get called.

Let's start by writing a generic version of the **PrintArray** function. We will, for now, assume we will still be working with arrays of the generic data type.

```
// PrintArrayGeneric<T>(T [])
// Assumes: An array has been initialized
// Result: Outputs the array
// NOTE: Overloaded method
public void PrintArrayGeneric<T>(T [] arr) {
    foreach (T elem in arr) {
        Console.Write("{0} ", elem);
    }
    Console.WriteLine();
}
```

Note that we have to note the generic data type as part of the function name. From that point forward, we replace the references to **int** or **double** with our generic type parameter **T**.

The sorting function is complicated by the use of generic data types. In particular, we have to use a more general means of comparing objects: the **CompareTo()** method. Many of the standard classes implement this method, which will compare two objects and return back a negative value if an object is "less than" another object, 0 if they are "equal", or a positive value if they are "greater than" another object.

There is a small amount of black magic involved here as we have to use a C# feature that we will talk about in Module 5 to make certain that the generic data type provides a **CompareTo()** method: the **where** clause on a function definition. More about that in a future lab.

```
// BubbleSortGeneric<T)
// Assumes: The type T supports the IComparable interface (implements CompareTo())
// Returns: Nothing, but the array arr is sorted as a side effect
public void BubbleSortGeneric<T>(ref T [] arr ) where T : IComparable
{
  int i;
  int j;
  for (i = (arr.Length - 1); i >= 0; i--)
  {
    int compareToResult = 0;
    for (j = 1; j <= i; j++)
    {
     compareToResult =  arr[j-1].CompareTo(arr[j]);
     if (compareToResult > 0)
     {
       T temp = arr[j - 1];
       arr[j - 1] = arr[j];
       arr[j] = temp;
     }
    }
  }
}
```

Note that the code is exactly the same as before except for the use of the generic data types and **compareTo()** method.

Add the following to the **main()** method to test our work:

```
bSortTestObj.BubbleSortGeneric<int>(ref anArrayOfIntegers);
bSortTestObj.PrintArrayGeneric<int>(anArrayOfIntegers);

bSortTestObj.BubbleSortGeneric<double>(ref anArrayOfDoubles);
bSortTestObj.PrintArrayGeneric<double>(anArrayOfDoubles);

bSortTestObj.BubbleSortGeneric<string>(ref anArrayOfStrings);
bSortTestObj.PrintArrayGeneric<string>(anArrayOfStrings);
```

Note how we can now use our sort function with any data type that implements the **IComparable** interface; i.e., implements the **CompareTo()** method.

4 Submission instructions

Combine a copy of your source code and a screen-shot of your program into a PDF file (you can use Microsoft Word to do this) and attach it to the assignment on Blackboard. Emacs 24.4.1 (Org mode 8.2.10)