

A case study of delegates and generics in C#

Hans-Wolfgang Loidl
<H.W.Loidl@hw.ac.uk>

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh



Semester 1 — 2018/19

Motivation

- As a case study, we want to implement a sequential *quick-sort* algorithm.
- The idea of quick-sort is:
 - ▶ Pick a *pivot* element in the list
 - ▶ *Partition* the list into elements *greater than* and elements *less than or equal* to the pivot element.
 - ▶ *Recursively* sort the two partitions.
- See this handout on quick-sort with examples¹
- We will develop three versions:
 - ① The first version operates on arrays of integers, with a fixed comparison operation.
 - ② The second version uses *delegates*, so that we can *re-define the comparison operation*.
 - ③ The third version additionally uses *generics*, so that we can *sort values other than integers*.

¹From Goodrich & Tamassia, “Data Structures & Algorithms” in Java

Components of in-place quicksort

We need the following functions

- A top-level function that performs sorting of an array:

```
1 private void QuickSort(int[] array, int from,
    int to, int level)
```

- A function that *partitions* a segment of the array:

```
1 private int Partition(int[] array, int from, int
    to, int pivot)
```

- A swap function that exchanges two array elements (in-place):

```
1 private static void Swap(int[] array, int i, int
    j)
```

This initial implementation fixes the base type (`int`) and the comparison fct (`>`).

Components of in-place quicksort

We need the following functions

- A top-level function that performs sorting of an array:

```
1 private void QuickSort(int[] array, int from,
    int to, int level)
```

- A function that *partitions* a segment of the array:

```
1 private int Partition(int[] array, int from, int
    to, int pivot)
```

- A swap function that exchanges two array elements (in-place):

```
1 private static void Swap(int[] array, int i, int
    j)
```

This initial implementation fixes the base type (`int`) and the comparison fct (`>`).

Structure of in-place quicksort

```
1 private void QuickSort(int[] array, int from, int to
    , int level) {
2 if (to - from < 1) {
3     return;          // a 1 elem list is sorted, per
                        definitionem
4 } else {
5     int pivot = from + (to - from) / 2;
6     pivot = Partition(array, from, to, pivot);
7     // recursive call on lower segment
8     QuickSort(array, from, pivot - 1, level+1);
9     // assert: IsSorted(array, from, pivot)
10    // recursive call on upper segment
11    QuickSort(array, pivot + 1, to, level+1);
12    // assert: IsSorted(array, pivot+1, to)
13 }
14 }
```

Partitioning (idea)

- The interface of the partitioning function is:

```
1  private int Partition(int[] array, int from, int  
    to, int pivot)
```

- It separates all elements less than the pivot element, from those larger than pivot element.
- The implementation uses the `from` and `to` variables as cursors, iterating over the array, and exchanging (swapping) data where needed.

Partitioning

```
1 private int Partition(int[] array, int from, int to,
   int pivot) {
2     int? pidx = null; int pivval = array[pivot];
3     while (from < to) {
4         if (array[from] > pivval) {
5             Swap(array, from, to);      to--;
6         } else {
7             if (array[from] == pivval) { pidx = from; }
8             from++;
9         }
10    }
11    if (!pidx.HasValue) {
12        if (array[from] == pivval) {
13            return from;
14        } else {
15            throw new System.Exception(String.Format("
               Partition: {0} element {1} not found",
               pivval));
16        }
17    }
```

Partitioning (cont'd)

```
1  if (array[from] > pivval) {
2      // bring pivot element to end of lower half
3      Swap(array, (int)pividx, from-1);
4      return from-1;
5  } else {
6      // done, bring pivot element to end of lower half
7      Swap(array, (int)pividx, from);
8      return from;
9  }
10 }
```

Swapping

Swapping the i -th with the j -th element in the array:

```
1  private static void Swap(int[] array, int i, int j){
2      if (i==j) {
3          return;
4      } else {
5          int tmp = array[i];
6          array[i] = array[j];
7          array[j] = tmp;
8      }
9  }
```

Delegate version: class definition

```
1 public class GenSort {
2
3     // enumeration for results of a generic comparison
4     // function
5     public enum Cmp { GT, EQ, LT };
6
7     // this delegate defines a comparison operator over
8     // the list elements
9     // see also (using System): public delegate int
10    // Comparison<in T>
11    public delegate Cmp CmpDelegate(int x, int y);
12
13    // the actual function to compare something
14    private CmpDelegate the_cmp;
15
16    public GenSort(CmpDelegate cmp) {
17        the_cmp = cmp;
18    }
19 }
```

Delegate version: partitioning

```
1 private int Partition(int[] array, int from, int to,
2     int pivot) {
3     int? pidx = null; int pivval = array[pivot];
4     while (from < to) {
5         if (the_cmp(array[from], pivval) == Cmp.GT) { //
6             using DELEGATE
7             Swap(array, from, to);
8             to--;
9         } else {
10             if (the_cmp(array[from], pivval) == Cmp.EQ) {
11                 pidx = from; }
12             from++;
13         } }
14     if (!pidx.HasValue) {
15         if (the_cmp(array[from], pivval) == Cmp.EQ) { //
16             using DELEGATE
17             return from;
18         } else {
19             throw new System.Exception(String.Format("
20
```

Delegate version: partitioning (cont'd)

```
1  if (!pividx.HasValue) {
2      if (the_cmp(array[from], pivot_val) == Cmp.EQ ) {
3          return from;
4      } else {
5          throw new System.Exception(String.Format("
6              Partition: {0} not found",
7              pivot_val));
8      }
9  }
10 if (the_cmp(array[from], pivot_val) == Cmp.GT) {
11     // bring pivot element to end of lower half
12     Swap(array, (int)pividx, from-1);
13     return from-1;
14 } else {
15     // bring pivot element to end of lower half
16     Swap(array, (int)pividx, from);
17     return from;
18 }
```

Using this version

```
1 public class Tester {
2     ...
3     // my sorting function: ascending order of integers
4     public static GenSort.Cmp intCmp(int x, int y) {
5         if (x>y)         return GenSort.Cmp.GT;
6         else if (x==y)    return GenSort.Cmp.EQ;
7         else              return GenSort.Cmp.LT;
8     }
9
10    public static void Main(string []args) {
11        ...
12        // inst. of sorter class, using int comp.
13        GenSort myGenSort = new GenSort(intCmp);
14        ...
15        myGenSort.QuickSort(arr);
16        ...
17    }
```

¹<http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/Samples/GenSort.cs>

Generics version: class definition

```
1 public class GenSort<T> {
2
3     // enumeration for results of a generic comparison
4     // function
5     public enum Cmp { GT, EQ, LT };
6
7     // this delegate defines a predicate over the list
8     // elements; mainly for testing
9     // see also (using System): public delegate int
10    Comparison<in T>
11    public delegate Cmp CmpDelegate(T x, T y);
12
13    // the actual function to compare something
14    private CmpDelegate the_cmp;
15
16    public GenSort(CmpDelegate cmp) {
17        the_cmp = cmp;
18    }
19 }
```

Generics version: partitioning

```
1 private int Partition(T[] array, int from, int to, int
    pivot) {
2     int? pidx = null; T pivval = array[pivot];
3     while (from < to) {
4         if (the_cmp(array[from], pivval) == Cmp.GT) { //
            using DELEGATE
5             Swap(array, from, to);
6             to--;
7         } else {
8             if (the_cmp(array[from], pivval) == Cmp.EQ) {
                pidx = from; }
9             from++;
10        }
11    }
12    if (!pidx.HasValue) {
13        if (the_cmp(array[from], pivval) == Cmp.EQ) {
14            return from;
15        } else {
16            throw new System.Exception(String.Format("

```

Generics version: partitioning

```
1  if (the_cmp(array[from], pivval) == Cmp.GT) {  
2      // bring pivot element to end of lower half  
3      Swap(array, (int)pividx, from-1);  
4      return from-1;  
5  } else {  
6      // done, bring pivot element to end of lower half  
7      Swap(array, (int)pividx, from);  
8      return from;  
9  }  
10 }
```

Using this Generics version

```
1 public class Tester {
2     ...
3     // my sorting function: ascending order of integers
4     public static GenSort<int>.Cmp intCmp(int x, int y) {
5         if (x>y)           return GenSort<int>.Cmp.GT;
6         else if (x==y)      return GenSort<int>.Cmp.EQ;
7         else                return GenSort<int>.Cmp.LT;
8     }
9     public static void Main(string []args) {
10        ...
11        // inst of sorter class, using int comp.
12        GenSort<int> myGenSort = new GenSort<int>(intCmp);
13        ...
14        myGenSort.QuickSort(arr);
```

¹<http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/Samples/GenGenSort.cs>

Summary

- Delegates and Generics are powerful mechanisms of abstraction.
- Delegates *abstract over code*, allowing to pass methods as arguments to other methods.
- The keyword `delegate` is used to define the interface for the parametric method.
- Generics *abstract over types*, allowing to define data-structures over different types.
- The notation `<T>` is used to specify a type argument to a class definition.