Programmentwicklung Advanced
Übungen Generics

Aufgaben

Inhaltsverzeichnis

1 Generics

1.1 Umbau zu Generics

Analysieren Sie die bestehenden Klassen `MyList` und `MySortedList`. Diese sind nicht generisch implementiert und arbeiten intern mit „`object`“. Das Ziel der Aufgabe ist es, die bestehenden Klassen so zu ändern, dass diese am Ende komplett generisch sind.

Nehmen Sie folgende Arbeitsschritte vor:

- Ändern Sie die Definition von Node
  von `private object data;`
  nach `private T data;`
  wobei T ein Typparameter ist.
- Die neue Liste heisst nun `MyList<T>`
- Passen Sie den restlichen Code entsprechend an.
- Erstellen Sie auch eine typsichere Variante der `MySortedList`. Stellen Sie sicher, dass als Typparameter nur Typen verwendet werden können, die das Interface `IComparable<T>` implementieren.
- Passen Sie die Klasse Person so an, dass diese `IComparable<T>` implementiert
- Verwenden Sie in der Main-Methode die typsichere `MySortedList`.

1.2 .NET List<T>

Die gegebene Ausgangslage entspricht von der Struktur her der vorangehenden Übung. Der Code verwendet aber anstelle einer eigenen Liste die in .NET vorhandene List<T>.
Verwenden Sie deren Methoden für die Sortierung und Filterung der Elemente. Halten Sie sich dabei ab die TODO-Anweisungen.

Beachten Sie auch die den Auszug aus der Klassendefinition von List<T>:

`void List.Sort ()`	Sorts the elements in the entire List using the default comparer.
`void List.Sort (Comparison<T> comp)` `delegate int Comparison<T> (T x, T y)`	Sorts the elements in the entire List using the specified System.Comparison.
`void List.ForEach (Action<T> action)` `delegate void Action<T> (T obj)`	Performs the specified action on each element of the List.
`List<T> List.FindAll (Predicate<T> match)` `delegate bool Predicate<T> (T obj)`	Retrieves all the elements that match the conditions defined by the specified predicate.

1.3 Nullable Types

Füllen Sie die folgende Tabelle aus. Gehen Sie davon aus, dass keiner der Ausdrücke einien Compilerfehler ergibt.

Es gelten die folgenden Variablen:

```
int a = 0;
bool b = false;
int? c = 10;
bool? d = null;
int? e = null;
```

Ausdruck	Ausdruckstyp	Resultat
`c + a`	`int?`	10
`a + null`		
`a < c`		
`a + null < c`		
`a > null`		
`(a + c - e) *9898+1000`		
`d`		
`d == d`		
`c ?? 1000`		
`e ?? 1000`		

1.4 Type Constraints

1.4.1 Aufgabe 1

Korrigieren Sie die Definition der folgenden generischen Klasse, ohne an der Methode GetNewInstance() etwas zu verändern.

```
class Test<T>
{
    public T GetNewInstance()
    {
        return new T();
    }
}
```

1.4.2 Aufgabe 2

Versuchen Sie die folgende ausserhalb von MyClass<T> zu instanziieren. Die Aufgabe macht von der Logik her keinen Sinn, es geht rein um das Verständnis von Generics und Type Constraints. Tipp: Sie müssen dazu zuerst einen neuen Typen (Klasse) definieren.

```
class MyClass<T> where T : MyClass<T> {}
```

Folgendes sollte möglich sein, die drei ??? müssen schlussendlich noch ersetzt werden:

```
MyClass<???> x = new MyClass<???>();
```

1.4.3 Aufgabe 3

Versuchen Sie, folgende Methode nur mit Type Constraints kompilierfähig zu machen. Die Methode soll sämtliche Elemente von "source" in eine neue Liste abfüllen.

Hinweis: Um über eine Variable in einem foreach-Loop zu iterieren, muss eine GetEnumerator()-Methode auf dem Typen vorhanden sein, siehe IEnumerable<T>

```
static class MyHelpers
{
    static TDest CopyTo<TSource, TDest, TElement>(TSource source)
    {
        TDest dest = new TDest();
        foreach (TElement element in source)
        {
            dest.Add(element);
        }
        return dest;
    }
}
```

1.5 Delegates

Analysieren Sie die Vorgabe. Sie stellen fest, dass es sich mehr oder weniger um die Musterlösung der letzten Übung „Delegates & Events“ / Übung „1.1 Sortieren beliebiger Arrays“ handelt.

Ersetzen Sie das Delegate `Comparer` durch ein generisches Delegate `Comparer<T>`.

Passen Sie den Programmcode so an, dass:

- Nur noch das generische Delegates verwendet wird
- Keine object-Variablen mehr vorhanden sind
- Keine Type Casts mehr im Code vorhanden sind
  Ausnahme: „`CompareFraction`“ benötigt zwei Type Casts für die Divisionen im Code

Passen Sie die Main()-Methode wie folgt an:

- Ersetzen Sie das Fraction- und das string-Array durch eine jeweilige List<T>
  - Geben Sie das Array dem Konstruktor von List<T> mit für die Initialisierung oder
  - Verwenden Sie Collection Initializers (noch nicht behandelt)
    https://msdn.microsoft.com/library/bb384062.aspx
- Sortieren Sie die Liste mit dem List<T>.Sort Operator
- Geben Sie anschliessend die Liste mit dem List<T>.ForEach Operator und einer anonymen Methode aus
- Konvertieren Sie die Fraction-Liste in eine Liste vom Typ List<string>
  Für die Umwandlung einer Fraction in einen String können Sie Fraction.ToString() verwenden, für die Umwandlung der Liste den Operator List<T>.ConvertAll.

1.6 Zusatzaufgaben

1.6.1 Aufgabe 1 – BinaryTree (leicht)

Die Klasse `BinaryTreeInt` implementiert einen binären Tree, der int-Werte speichern kann. Studieren Sie die Implementation und das Testprogramm.

Leiten Sie aus der vorhandenen Implementation eine generische Klasse BinaryTree<T> ab und testen Sie diese.

1.6.2 Aufgabe 2 – ForAll

Die Klasse Array enthält leider keine ForAll Methode. Dies ist eine Methode, die eine Action für jedes Element eines Arrays, das ein übergebenes Prädikat erfüllt, ausführt. Programmieren Sie ihre eigene ForAll-Methode nach folgender Vorgabe:

Aufruf

```
ForAll(
    new int[] { 1, 2, 3, 4 },
    delegate(int i) { return i > 2; },
    delegate(int i) { Console.WriteLine(i); }
);
```

Resultat

```
3
4
```

Argumente für ForAll:

- Zu traversierendes Array
- Prädikat vom Typ `delegate bool Predicate<T>`
  Nur Elemente, welche das Prädikat erfüllen werden berücksichtigt
- Aktion vom Typ `delegate void Action<T>`
  Aktion welche auf den verbleibenden Elementen ausgeführt wird

1.6.3 Aufgabe 3 – Comparison

Schreiben sie eine statische Methode `DefaultCompare`, die zum Delegate `Comparison<T>` kompatibel ist und beliebige Objekte vergleichen kann, die das Interface `IComparable<T>` implementieren.

```
public delegate int Comparison<in T>(T x, T y);
```

Der folgende Aufruf soll z.B. möglich sein:

```
Comparison<int> comparer = DefaultCompare;
Console.WriteLine(comparer(7, 7));
```

Tipp: Ihre Methode benötigt einen Typparameter sowie einen Type Parameter Constraint.

1.6.4 Aufgabe 4 – Combiner

Schreiben Sie eine `CombineAll` Methode. Diese soll alle Elemente zweier Arrays mit einer beliebigen Methode kombinieren und als Resultat ein neues Array zurückgeben.

```
int[] res = CombineAll<int, int, int>(
    new int[] { 2, 3, 4 },
    new int[] { 1, 2, 5 },
    delegate(int a, int b) { return a * b; }
);
```

Die Variable „res“ soll die Resultate 2, 6, 20 enthalten.

Vorgehen:

- Definieren Sie zuerst den generischen Delegate-Typ `MyConverter`. Dieser legt fest, wie das Delegate-Objekt aussehen muss, das der Methode `CombineAll` übergeben wird.
- Implementieren Sie die `CombineAll` - Methode und testen Sie diese mit den oben aufgeführten Aufrufen.

1.6.5 Aufgabe 5 – Bubble Sort

Sie sollen eine `Sort` Methode definieren, die beliebige Elementmengen vom beliebigen Typ sortieren kann, solange sie die folgenden Eigenschaft haben:

- Die Elemente müssen vergleichbar sein
- Auf die Elementmenge muss per Index lesend und schreibend zugegriffen werden können

Beachten Sie, dass die Methode beliebige Collections funktionieren soll, auch wenn diese keine Interfaces implementieren. Verwenden Sie auch keine Type-Parameter Constraints!

Tipp:
Sie müssen ein Delegate `ElementGetter` für das Lesen eines Elementes der Collection sowie ein Delegate `ElementSetter` für das Setzen des Wertes eines Elementes der Collection definieren.

Beim Aufrufen von `Sort` werden die konkreten Delegate-Objekte für das Lesen und Schreiben der Werte sowie ein Delegate-Objekt vom Typ `System.Comparison<T>` für das Vergleichen von zwei Werten mitgegeben.

Der wichtige Teil der Übung ist die Signatur der Methode. Die Implementierung kann auch weggelassen werden.

1.6.6 Aufgabe 6 – FoldR

Die FoldR Methode ist in der funktionalen Programmierung sehr beliebt. Folgender Code ist dabei schon gegeben:

```
public delegate T FoldHandler<T>(T p1, T p2);

public static T FoldR<T>
    (T start, IEnumerable<T> elements, FoldHandler<T> foldHandler)
{
    T akt = start;
    foreach (T element in elements)
    {
        akt = foldHandler(akt, element);
    }
    return akt;
}
```

Teil a)

Schreiben Sie eine Methode `Smallest`, die das kleinste Element einer Menge zurückliefern soll. Verwenden Sie für ihre Implementierung die `FoldR` Methode. Ein Aufruf von `FoldR` genügt. Übergeben Sie ihm als `FoldHandler` eine anonyme Methode (dieses braucht auch nur eine Zeile Code).

Teil b)

Überladen Sie die Methode `Smallest`, so dass sie keinen comparison Parameter mehr benötigt! Die Signatur wird reduziert auf zwei Parameter. Sie soll aber nur noch mit Mengen arbeiten, welche das Interface `IComparable` implementieren.

Tipp:
Verwenden Sie zur Implementierung die erste `Smallest` Methode (ohne diese zu ändern) und die `DefaultCompare` Methode, aus einer früheren Teilaufgabe.

Lösungen

Siehe Generics_Musterlsg.sln