C#/.Net project cluster **ITU, May 2005**

Exercises
for Tuesday 3 May 2005

2005-04-26

The first two pages of exercises concern generic types and methods; the last page concerns attributes.

Exercise C# 2.1 The purpose of this exercise is to understand the declaration of a generic type in C# 2.0. The exercise concerns a generic struct type because structs are suitable for small value-oriented data, but declaring a generic class would make little difference.

A generic struct type Pair<T,U> can be declared as follows (C# Precisely example 182):

```
public struct Pair<T,U> {
  public readonly T Fst;
  public readonly U Snd;
  public Pair(T fst, U snd) {
    this.Fst = fst;
    this.Snd = snd;
  }
  public override String ToString() {
    return "(" + Fst + ", " + Snd + ")";
  }
}
```

(a) In a new source file, write a C# program that includes this declaration and also a class with an empty `Main` method. Compile it to check that the program is well-formed.

(b) Declare a variable of type Pair<String, `int`> and create some values, for instance `new Pair<String,int>("Anders", 13)`, and assign them to the variable.

(c) Declare a variable of type Pair<String, `double`>. Create a value such as `new Pair<String,double>("Phoenix", 39.7)` and assign it to the variable.

(d) Can you assign a value of type Pair<String,`int`> to a variable of type Pair<String,`double`>? Should this be allowed?

(e) Declare a variable `grades` of type Pair<String,`int`>[ ], create an array of length 5 with element type Pair<String,`int`> and assign it to the variable. (This shows that in C#, the element type of an array may be a type instance.) Create a few pairs and store them into `grades[0]`, `grades[1]` and `grades[2]`.

(f) Use the `foreach` statement to iterate over `grades` and print all its elements. What are the values of those array elements you did not assign anything to?

(g) Declare a variable `appointment` of type Pair<Pair<`int`,`int`>,String>, and create a value of this type and assign it to the variable. What is the type of `appointment.Fst.Snd`? This shows that a type argument may itself be a constructed type.

(h) Declare a method `Swap()` in Pair<T,U> that returns a new struct value of type `Pair<U,T>` in which the components have been swapped.

Exercise C# 2.2 The purpose of this exercise and the next one is to experiment with the generic collection classes of C# 2.0. Don't forget the directive `using System.Collections.Generic;`.

Create a new source file. In a method, declare a variable `temperatures` of type List<`double`>. (The C# collection type List<T> is similar to Java's ArrayList<T>). Add some numbers to the list. Write a `foreach` loop to count the number of temperatures that equal or exceed 25 degrees.

Write a method `GreaterCount` with signature

```
static int GreaterCount(List<double> list, double min) { ... }
```

that returns the number of elements of `list` that are greater than or equal to `min`. Note that the method is not generic, but the type of one of its parameters is a type instance of the generic type List<T>.

Call the method on your temperatures list.

Exercise C# 2.3 Write a generic method with signature

```
static int GreaterCount(IEnumerable<double> eble, double min) { ... }
```

that returns the number of elements of the enumerable `eble` that are greater than or equal to `min`. Call the method on an array of type `double[]`. Can you call it on an array of type `int[]`?

Now call the method on `temperatures` which is a List<double>. If you just call `GreaterCount(temperatures, 25.0)` you'll actually call the `GreaterCount` method declared in exercise 2.2 because that method is a better overload (more specific signature) than the new `GreaterCount` method. To call the new one, you must cast `temperatures` to type IEnumerable<`double`> — and that's legal in C#.

In C# it is legal to overload a method on type instances of generic types. You may try this by declaring also

```
static int GreaterCount(IEnumerable<String> eble, String min) { ... }
```

This methods must have a slightly different method body, because the operators (`<=`) and (`>=`) are not defined on type String. Instead, use method `CompareTo(...)`. Maybe insert a `Console.WriteLine(...)` in each method to be sure which one is actually called.

Exercise C# 2.4 The purpose of this exercise is to investigate type parameter constraints. You may continue with the same source file as in the previous two exercises.

We want to declare a method similar to `GreaterCount` above, but now it should work for an enumerable with any element type T, not just `double`. But then we need to know that values of type T can be compared to each other. Therefore we need a constraint on type T:

```
static int GreaterCount<T>(IEnumerable<T> eble, T x) where T : ... { ... }
```

(Note that in C# methods can be overloaded also on the number of type parameters; and the same holds for generic classes, interfaces and struct types). Complete the type constraint and the method body. Try the method on your List<`double`> and on various array types such as `int[]` and `String[]`. This should work because whenever T is a simple type or String, T implements IComparable<T>.

Exercise C# 2.5 Create a new source file `GenericDelegate.cs` and declare a generic delegate type Action<T> that has return type void and takes as argument a T value. This is a generalization of yesterday's delegate type IntAction.

Declare a class that has a method

```
static void Perform<T>(Action<T> act, params T[] arr) { ... }
```

This method should apply the delegate `act` to every element of the array `arr`. Use the `foreach` statement when implementing method `Perform<T>`.

Exercise C# 2.6 (Optional) As you know, C# does not have wildcard type parameters. However, most uses of wildcards in the parameter types of methods can be simulated using extra type parameters on the method. For instance, in the case of the `GreaterCount<T>(IEnumerable<T> eble, T x)` method, it is not really necessary to require that T implements IComparable<T>. It suffices that there is a supertype U of T such that U implements IComparable<U>. This would be expressed with a wildcard type in Java, but in C# 2.0 it can be expressed like this:

```
static int GreaterCount<T,U>(IEnumerable<T> eble, T x)
  where T : U
  where U : IComparable<U>
{ ... }
```

When you call this method, you may find that the C# compiler's type inference sometimes cannot figure out the type arguments to a method. In that case you need to give the type arguments explicitly in the methods call, like this:

```
int count = GreaterCount<Car,Vehicle>(carList, car);
```

Exercise C# 2.7 The purpose of this exercise is to illustrate the use and effect of a predefined attribute.

The predefined attribute Obsolete (see C# Precisely section 28) may be put on classes, methods, and so on that should not be used — it corresponds to the 'deprecated' warnings so well known from the Java class library.

Declare a class containing a method

```
static void AcousticModem() {
  Console.WriteLine("beep buup baap bzfttfsst %^@~#&&^@CONNECTION LOST");
}
```

Put an Obsolete attribute on the `AcousticModem` method and call the method from your `Main` method. What message do you get from the C# compiler? Does the message concern the declaration or the use of the `AcousticModem` method?

Exercise C# 2.8 The purpose of this exercise is to show how to declare a new attribute, how to put it on various targets, and how to detect at run-time what attributes have been put of a given target (in this case, a method).

Create a new source file. Declare a custom attribute BugFixed that can be used on class declarations, struct type declarations and method declarations. It must be legal to use BugFixed multiple times on each target declaration.

There must be two constructors in the attribute class: one taking both a bug report number (an `int`) and a bug description (a string), and another one taking only a description. (Presumably the latter is used when a bug does not get reported through the official channels). When no bug number is given explicitly, the number -1 (minus one) is used. The attribute class should have a `ToString()` method that shows the bug number and description if the bug number is positive, otherwise just the description.

It should be legal to use the BugFixed attribute like this:

```
class Example {
  [BugFixed(4, "Performance: Uses SortedDictionary")]
  [BugFixed(3, "Throws IndexOfOutRangeException on empty array")]
  [BugFixed("Performance: Uses repeated string concatenation in for-loop")]
  [BugFixed(2, "Loops forever on one-element array")]
  [BugFixed(1, "Spelling mistakes in output")]
  public static String PrintMedian(int[] xs) {
    /* ... */
    return "";
  }

  [BugFixed(67, "Rounding error in quantum mechanical simulation")]
  public double CalculateAgeOfUniverse() {
    /* ... */
    return 11.2E9;
  }
}
```

Write an additional class with a `Main` method that uses reflection to get the public methods of class Example, gets the BugFixed attributes from each such method, and prints them. If `mif` is a MethodInfoObject, then `mif.GetCustomAttributes(typeof(t), false)` returns an array of the type `t` attributes.

Some inspiration may be found in the full source code for C# Precisely example 208, which can be downloaded from the book's homepage `http://www.dina.kvl.dk/~sestoft/csharpprecisely/`.