

Exercises for Tuesday 31 January 2006

Kasper Østerbye and Peter Sestoft 2006-01-31

The first two pages of exercises concern generic types and methods in Java; the remainder concern generic types and methods in C#. You need not solve the exercises in this order. If you run out of interesting exercises, then:

- Try to solve some of the C# exercises in Java instead; that will throw much light on what can and cannot be done with Java generics.
- Download the C5 collection library and implement some algorithms using it. (Currently this probably requires Microsoft VS 2005).

Exercise Java 1 The purpose of this exercise is to make sure you are using the new compiler. Consider the following two classes (which will be used in the next exercise):

```
class Person {
    String name;
    Integer birthYear;
    Person(String name, int birth){
        this.name = name;
        birthYear = birth;           // boxing
    }
    Integer age(){
        return 2004-birthYear;     // unboxing & boxing
    }
}

class Student extends Person{
    String university;
    Student(String name, int birth){
        super(name, birth);
        university = "Copenhagen";
    }
    Student(String name, int birth, String university){
        super(name, birth);
        this.university = university;
    }
}
```

The file can be found at <http://www.dina.kvl.dk/~sestoft/tmp/PersonStudent.java>

Compile the two classes using the Java 5.0 compiler. The usage of auto-boxing in Person serves only to make sure that we are really using the Java 5.0 compiler and virtual machine. There is no backward capability with old virtual machines.

Exercise Java 2 The purpose of this exercise is to let you use the generic List interface and ArrayList class of the Java 5.0 collection library. The exercise will have a lot of focus on which types are compatible. They can be solved without actually running the program you develop, as it only concerns compile time aspects.

Write a new class with a main method. The rest of the exercise can be done by adding code fragments to this main method. You will need access to the Person and Student classes from exercise Java-1.

1. Declare a List lp of Persons and assign an ArrayList of Person to it. Remember to import java.util.* to get access to the collection library. Your task is to get the type parameters right in this declaration. Use the compiler to verify your results.
2. What error message do you get if you try to insert a string into lp?
3. Make statements that insert a Person object first and a Student object second into lp.

4. Declare a variable `p` of type `Person` and a `p` and a variable `s` of type `Student`.
 - a. Will `lp.get(1)` return a `Person` or a `Student` object?
 - b. Why does `s = lp.get(1)` give a compile time error?
 - c. Why does `s = (Student)lp.get(1)` not give a compile time error?
 - d. If the right thing is to cast, what good are generics then?
5. Declare a list `ls` of `Students`.
 - a. Remember, if `p` is declared as `Person`, and `s` is declared as `Student`, one can assign as: `p = s;` and `s = (Student)p;`
 - b. What error message is given if you make the assignment `ls = lp?`
 - c. Why is it illegal to write `ls = lp?`
 - d. Why is it also illegal to write `lp = ls?`
 - e. Note: array assignments like this are not often used in practice, but the same rules apply when you pass lists as parameters to methods, which happens more frequently.
6. Declare a variable `lep` of type `List<? extends Person>`.
 - a. How can you initialize the `lep` variable? Why can you not write `new ArrayList<? extends Person>()`?
 - b. Can you make the assignment `lpp = lp?`
 - c. Can you make the assignment `lpp = ls?`
 - d. Can you insert elements into `lpp`? Why (not)?
 - e. Can you get elements from `lpp`?
7. Declare a map that can map a name (a `String`) to a `Person`. Use the interface `Map` as type for the variable, and `HashMap` as the class to instantiate.
8. Make a statement that inserts a `Person` and one that inserts a `Student`.
9. Make a statement that looks up a `Person`. Can the result be assigned to variable `p`? can it be assigned to variable `s`?

Exercise Java 3 This Java program (from <http://www.dina.kvl.dk/~sestoft/tmp/Try.java>) contains a type cast to a generic parameter. What does the Java compiler say about that? Where is the `ClassCastException` thrown when you run the program? Explain why, possibly by disassembling the bytecode with `javap -c Garage` and `javap -c Try`.

```
class Car { }
class Sedan extends Car { }

class Garage<T extends Car> {
    private T myCar;
    public void put(Car car) {
        myCar = (T)car;
    }
    public T get() {
        return myCar;
    }
}

class Try {
    public static void main(String[] args) {
        Garage<Sedan> garage = new Garage<Sedan>();
        System.out.println("Putting Car in Garage<Sedan>");
        garage.put(new Car());
        System.out.println("Getting Sedan from Garage<Sedan>");
        Sedan sedan = garage.get();
    }
}
```

Exercise C# 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 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# 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 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# 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# 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.

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# 6 As you know, Java has wildcard type arguments, but C# does not. However, many 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);
```