# Some Notes on Generics in C#

Janusz Jabłonowski

***Abstract:*** *Generics allow programmers to write more abstract code, which is - to some degree - independent from the types involved in the underlying algorithm. For example the code handling stack operations does not depend on the type of stack elements. Many languages support the notion of generics (for example templates in C++ allow writing generic code). Recently generics have been added to the Java programming language and the standardizing committee for the C# programming language is working on the second version of C#, which includes generics. Also beta releases of the new Visual Studio environment are accessible, which include generics support. In this paper we describe our experiences from the work with generics in C#.*

***Key words:*** *Programming Languages, Object-Oriented Programming, Generic types.*

## INTRODUCTION

Generics allow the programmer to write more abstract code, which is - to some degree - independent from the types involved in the underlying algorithm. For example the code handling stack operations does not depend on the type of stack elements. Many languages support the notion of generics (for example templates in C++ [2, 8] allow writing generic code). Recently generics have been added to the Java programming language [1, 3, 4] and the standardizing committee for the C# [6, 7] programming language is working on the second version of C#, which includes generics. Also beta releases of the new Visual Studio environment with generics support are accessible. This paper describes experiences from the work with generics in C#. All examples given in this paper has been tested under the Microsoft Visual Studio 2005 codename "Whidbey" beta (Community Technology Preview dated February 2005).

## WHY GENERICS ARE IMPORTANT

As was previously mentioned generics allow writing more abstract code, where the programmer does no have to worry about the concrete type his data structure will be build on or his method will deal with. But this is not all. Generics allow writing such code in a type-safe manner. In a language like C#, where every value is a descendant of the Object type, it is possible - even without generics - to write a piece of code, which abstracts the type of values stored in a data structure. Let us consider an example.

Here is a simple class implementing the stack data structure (the given implementation restricts the maximal size of the stack only to simplify the presentation):

```
class MyStack {
       Object[] elts;
       int count;
       public void push(Object elt) { elts[count++] = elt; }
       public object pop() { return elts[--count]; }
       public MyStack() { elts = new Object[100]; count = 0; }
  }
```

Having this class one can compile and successfully run code like this:

```
MyStack s = new MyStack();
s.push(new Person("John"));
```

```
    s.push(13);
    int i = (int)s.pop(); // This cast is neccesary
    Person o = (Person)s.pop(); // This one too
    System.Console.WriteLine(o.whatIsYourName());
```

The class Person may be defined like this:

```
    class Person {
            String name;
            public Person(String name) { this.name = name; }
            public String whatIsYourName() { return name;  }
     }
```

It may seem that we have achieved our goal: we have created stack implementation which is independent of the type of objects stored inside it. But unfortunately our solution is error prone. It is enough to switch in the code sample given previously the two lines with invocations of the pop method, to get a program which compiles without a warning but which is not type safe (actually it will throw the InvalidCastException at the first attempt to pop an element form the stack).

It is clear that the use of casts in languages with strict type checking is particularly wrong. If one has chosen a language with type system to have more reliable programs, then cheating the type verification system with casts is at least inconsistent. Doing so not only results in programs with type errors, but still worse, it may give the programmer an illusory feeling that his program is type safe (since the language and the compiler impose type checking).

The idea of generics is of course not new, let us mention for example functional languages like Hope or ML, which allow the programmer to define so called polymorphic functions. The solution adopted there was based on the idea, that the function should be abstract enough, to be able to cope with any types of parameters matching the specified (or calculated by the compiler) types of parameters. A different approach was taken in C++, where the notion of template was introduced. A template is a way to describe how the compiler should generate the executable code for those parameters' types which occurred in the program being compiled. The difference between this two approaches is that the functional one results in one version of generic function emitted by the compiler, version which is abstract enough to deal with any allowed parameter type, whereas in the template approach, the compiler generates as many instances of the template as there are different (in the sense of parameters) uses of the template. The virtue of the second approach is the efficiency – the generated code of a template instance is specifically generated and optimized for the particular parameter type.

The solution adopted in C# is somewhat in between the two presented above. On the one hand the object types are all treated in the same way (i.e. for a generic class Stack, which will be used to store values of merely object types only one version of code will be generated by the compiler). On the other hand for each simple type (called value type in C#) the compiler will generate different code. This approach enables generation of both short (only one copy for each object type) and efficient compiled code.

## DEFINING GENERICS TYPES IN C#

Let us define now a type safe, generic implementation of the stack data structure. The specification of generic type requires special syntax with angle brackets for designating parameters of a generic type. Here is again our sample (and restricted) stack implementation:

```
class MyStack<T>  {
      T[] elts;
      int count;
      public void push(T elt) { elts[count++] = elt; }
      public T pop() { return elts[--count]; }
      public MyStack() { elts = new T[100]; count = 0; }
};
```

Let us note the similarity with the non-generic example given in the previous section. In fact the only changes are:

- the type parameter given in angle brackets at the beginning of the class declaration,
- the replacement of all occurrences of object type with the T type parameter.

But the semantics is of course different. Previously we had a stack of everything, with no way to assure that our use of it is type safe. Now the compiler is able to type check every use of the defined generic type and to find all violations of the type system.

The use of generic type is also quite simple: we just have to state the type of objects to be stored in our data structure (again the syntax uses angle brackets):

```
MyStack<Person> s = new MyStack<Person>();
s.push(new Person("John"));
// s.push(13);  // This will not compile now!
Person o = s.pop(); // No casting is required here any more
System.Console.WriteLine(o.whatIsYourName());
```

Now it is not possible to put by accident an integer into a stack of Persons!

What will happen if one will try to use generics and the old way of declaring type of stack elements as just objects? That is, if someone will write something like that:

```
MyStack<Object> s = new MyStack<Object>();
s.push(new Person("John"));
s.push(13);
int i = (int)s.pop(); // This casting is necessary again
Person o = (Person)s.pop(); // Here also
System.Console.WriteLine(o.whatIsYourName());
```

It is allowed, but again it forces as to using typecasting, so it is not the proper way of programming. But this example leads us to a very interesting problem: how to specify a structure which should be used only for some (not all) types of elements?

## USING CONSTRAINTS

Let us consider a collection of Persons. Each Person object understands a message *whatIsYourName*. Using inheritance we can create Persons hierarchy with for example Children, Students, Employees and then further we can specialize Employees as Clerks, TeachingAssistants, MovieStars etc. Let us assume, that in our application we want to define a collection of Persons (of various kinds), and that we want the collection to be able to print out names of all Persons it contains. At first sight it looks that we can do it as follows (again the implementation of MyCollection is very restricted to make it short):

```
class MyCollection<T>
{
      T[] elts;
      int count;
      public void insert(T elt) { elts[count++] = elt; }
      public void askForNames() {
            for (int i = 0; i < count; i++)
                  System.Console.WriteLine(elts[i].whatIsYourName());
      }
      public MyCollection() { elts = new T[100]; count = 0; }
};
```

And then we can try to use it like this:

```
MyCollection<Person> c = new MyCollection<Person>();
c.insert(new Person("John"));
// …
c.askForNames();
```

But the class declaration will not compile. It is quite natural: the compiler does not know that the MyCollection class is going to be used only with Person type parameter (in fact any subclass of Person as a parameter would be also correct). Hence the compiler rejects the line:

```
System.Console.WriteLine(elts[i].whatIsYourName());
```

because it does not know how to call `whatIsYourName` on value of any type (T). Let us note, that also this version

```
for (int i = 0; i < count; i++)
      System.Console.WriteLine(((Person) elts[i]).whatIsYourName());
```

is incorrect. In this case the compiler does not know how to convert value of any type (T) into Person object.

Since such problems occur quite often when one uses generics there is a commonly accepted solution: constraints (they occur also for example in Java). Defining MyCollection we just have to state that the type parameter T is allowed only to be Person (or any subclass of it). The syntax of constraints declaration involves the **where** keyword, used as follows:

```
class MyCollection<T>
      where T: Person
{
      // … as above …
};
```

Now the previously showed code fragment will compile. And of course the following one will not, because it does not preserve the restriction we have just introduced:

```
MyCollection<Object> c = new MyCollection<Object>();
// …
```

The **where** clause allows also more sophisticated constraints. Let us consider a generic class with two type parameters:

```
class MyGenericClass<T1, T2>
// …
```

We can easily specify, that the first parameter must inherit from (or implement) the second:

```
class MyGenericClass<T1, T2>
      where T1: T2
// …
```

Let us assume now that the first type parameter should be a MyCollection (or its descendant) object, to which we are going to insert elements of the second type. This also can be easily specified:

```
class MyGenericClass<T1, T2>
      where T1: MyCollection<T2>
// …
```

Another interesting example for restricted type parameters is given in [5], where a data structure is considered, which requires comparison operation on stored values (as BST for example does).

In general the **where** clause lets specify that a type parameter is at most as general as the type expression given after colon. It is not possible to specify that it is at least as general as a given type expression.

One constraint which is often very difficult to express (or not expressible at all) concerns constructors. This is due to the fact, that in most object-oriented programming languages they are not inherited (although they usually are called, sometimes automatically, from the constructors of the subclass). Hence it is easy to express the fact, that each value of the parameter type has a *whatIsYourName* method – it is enough to define an interface with that method (or even a class as we did before) and point it out in the constraint clause. Each class implementing that interface (or subclass of the given class) will have such method. But for constructors an additional constraint is needed. In C# it has the form
    new()
and specifies, that the parameter type must have parameterless constructor. Unfortunately there is no way in C# to express that the parameter type must have a constructor with specific parameter types.

## BUILT-IN GENERICS

Generics are so fundamental, that it would be strange, if the language itself (and more precisely its libraries) would not have built-in generic types. The most common use for generics is of course creating collections of various kinds. C# provides the user with a rich set of generic collections. Let us just call List or Dictionary. Of course our MyCollection class could be implemented (in a simpler way) using collections provided by C#:

```
class MyCollection<T>
      where T : Person
```

```
    {
          List<T> elts;  // No count needed any more
          public void insert(T elt) { elts.Add(elt); }
          public void askForNames()
          {
                foreach (Person p in elts)
                      System.Console.WriteLine(p.whatIsYourName());
          }
          public MyCollection() { elts = new List<T>(); }
    };
```

## CONCLUSIONS

It is apparent that generic types are needed by programmers for writing safer, more robust applications. Some commonly used languages have support for generics already for quite long time (e.g. C++), to some other only recently this mechanism was introduced (Java) or is being introduced just now (C#). In this paper a short description was given of the author's experiences with using generics in C#. The overall judgment is positive, the implementation is easy to use and at the same time provides most of the tools (but not all, for example the constructor constraint should be more elaborate) needed for comfortable work with generic types.

## REFERENCES

[1] Bracha G., *Generics in the Java Programming Language*, http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, 2004.

[2] British Standards Institute, *The C++ standard. Incorporating Technical Corrigendum No. 1*. John Wiley & Sons, 2003.

[3] Jabłonowski J., *Typy uogólnione w Javie* (In polish), VI Krajowa Konferencja Inżynierii Oprogramowania (KKIO VI), pp. 479 - 490, Gdańsk, Poland. WNT 2004.

[4] Joy B., Gosling J., Bracha G., *The Java Language Specification*, Second edition. Addison-Wesley Professional, 2000.

[5] Lowy J., *An introduction to C# Generics*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs05/html/csharp_generics.asp, 2005.

[6] Microsoft, Specification of C#, Version 2.0, May 2004, available from http://msdn.microsoft.com/vcsharp/programming/language/

[7] Standard ECMA-334, *C# Language Specification,* 2002.

[8] Stroustrup B., *The C++ Programing Language*, Addison Wesley. Reading Mass., 2000.

## ABOUT THE AUTHOR

Assisting Prof. Janusz Jabłonowski, PhD, Department of Mathematics, Informatics and Mechanics, Warsaw University, Phone: +4822 5544401, E-mail: janusz@mimuw.edu.pl.