Generic Functions in Java

António Menezes Leitão

April, 2016

1 Introduction

The Common Lisp Object System (CLOS) is an object-oriented layer for Common Lisp that is very different from the typical object-oriented languages such as Java. In CLOS, the programmer organizes its program using the concept of *generic function*. A generic function is a function containing different specialized implementations for different types of arguments. Each specialization of a generic function is a *method*.

Everytime a generic function is applied to a set of arguments, the actual types of the arguments are used to find the set of applicable methods, which are sorted according to its specificity and, then, combined to create the effective method. The application of the effective method to the provided arguments will then produce the intended results. This form of operation supports multiple dispatch, where the applicable methods are determined based on the types of all arguments.

Although generic functions were invented for Common Lisp, nothing prevents us from applying the same ideas to Java. Unfortunately, Java lacks metaprogramming capabilities, which makes it a bit more difficult to hide the syntatical complexity. Taking this into account, here is an example of the use of this concept:

```
final GenericFunction add = new GenericFunction("add");
add.addMethod(new GFMethod() {
        Object call(Integer a, Integer b) {
            return a + b;
        }});
add.addMethod(new GFMethod() {
        Object call(Object[] a, Object[] b) {
            Object[] r = new Object[a.length];
            for (int i = 0; i < a.length; i++) {
                r[i] = add.call(a[i], b[i]);
            }
            return r;
        }});
```

The previous fragment creates the `add` generic function, and adds two methods to the function, one specialized for `Integer`s and the other specialized for `Object[]`s.

In order to visualize the results of the application of the generic function, we will use the following Java method definition:

```
public static void println(Object obj) {
    if (obj instanceof Object[]) {
        System.out.println(Arrays.deepToString((Object[])obj));
    } else {
        System.out.println(obj);
    }
}
```

Here is an example of the use of the generic function `add`:

```
 println(add.call(1, 3));
 println(add.call(new Object[] { 1, 2, 3 }, new Object[] { 4, 5, 6 }));
 println(add.call(new Object[] { new Object[] { 1, 2 }, 3 },
                  new Object[] { new Object[] { 3, 4 }, 5 }));
```

that produces the following output:

```
4
[5, 7, 9]
[[4, 6], 8]
```

When the generic function does not contain methods applicable to a particular set of arguments, an error is generated. For example:

```
println(add.call(new Object[] { 1, 2 }, 3));
```

causes

```
Exception in thread "main" java.lang.IllegalArgumentException:
No methods for generic function add with args [[1, 2], 3]
of classes [class [Ljava.lang.Object;, class java.lang.Integer]
```

Naturaly, many more methods can be added to the generic function:

```
add.addMethod(new GFMethod() {
        Object call(Object[] a, Object b) {
            Object[] ba = new Object[a.length];
            Arrays.fill(ba, b);
            return add.call(a, ba);
        }});
add.addMethod(new GFMethod() {
        Object call(Object a, Object b[]) {
            Object[] aa = new Object[b.length];
            Arrays.fill(aa, a);
            return add.call(aa, b);
        }});
add.addMethod(new GFMethod() {
        Object call(String a, Object b) {
            return add.call(Integer.decode(a), b);
        }});
add.addMethod(new GFMethod() {
        Object call(Object a, String b) {
            return add.call(a, Integer.decode(b));
        }});
add.addMethod(new GFMethod() {
        Object call(Object[] a, List b) {
            return add.call(a, b.toArray());
        }});
```

allowing the function to be used with many different kinds of arguments:

```
println(add.call(new Object[] { 1, 2 }, 3));
println(add.call(1, new Object[][] { { 1, 2 }, { 3, 4 } }));
println(add.call("12", "34"));
println(add.call(new Object[] { "123", "4" }, 5));
println(add.call(new Object[] { 1, 2, 3 }, Arrays.asList(4, 5, 6)));
```

producing the following output:

```
[4, 5]
[[2, 3], [4, 5]]
46
[128, 9]
[5, 7, 9]
```

Another important feature of the concept of generic function is the role that methods can take in the effective method. In fact, the *standard method combination* allows not only *primary methods*, just like the ones we saw previously, but also *before methods* and *after methods* that, when applicable, run before or after the primary methods.

Here is one example of the use of *before* and *after* methods:

```
final GenericFunction explain = new GenericFunction("explain");
explain.addMethod(new GFMethod() {
        Object call(Integer entity) {
            System.out.printf("%s is a integer", entity);
            return "";
        }});
explain.addMethod(new GFMethod() {
        Object call(Number entity) {
            System.out.printf("%s is a number", entity);
            return "";
        }});
explain.addMethod(new GFMethod() {
        Object call(String entity) {
            System.out.printf("%s is a string", entity);
            return "";
        }});
explain.addAfterMethod(new GFMethod() {
        void call(Integer entity) {
            System.out.printf(" (in hexadecimal, is %x)", entity);
        }});
explain.addBeforeMethod(new GFMethod() {
        void call(Number entity) {
            System.out.printf("The number ", entity);
        }});
println(explain.call(123));
println(explain.call("Hi"));
println(explain.call(3.14159));
```

When executed, the previous code fragment prints

```
The number 123 is a integer (in hexadecimal, is 7b)
Hi is a string
The number 3.14159 is a number
```

2 Goals

Implementation of the concepts of `GenericFunction` and `GFMethod`, following the syntax and semantics previously described. These classes should be implemented in the package `ist.meic.pa.GenericFunctions`.

Note that the implementation only needs to support the *standard method combination*, with primary, before, and after methods. It is not necessary to implement *around* methods or mechanisms to call the next applicable method (`call-next-method`, in CLOS parlance).

Note also that the specificity of methods should be based on the Java type hierarchy.

Finally, in order to better adapt to Java's type system, all methods will be implemented as annonymous inner classes extending `GFMethod` with a `call` Java method. Similarly, all `GenericFunction`s will be invoked by calling the `call` Java method. All primary methods will have `Object` as return type, while all before and after methods will have `void` return type.

2.1 Extensions

You can extend your project to further increase your grade above 20. Note that this increase will not exceed **two** points that will be added to the project grade for the implementation of what was required in the other sections of this specification.

Be careful when implementing extensions, so that extra functionality does not compromise the functionality asked in the previous sections. In order to ensure this behavior, you should implement all your extensions in a different package named `ist.meic.pa.GenericFunctionsExtended`.

3 Code

Your implementation must work in Java 7 or Java 8.

The written code should have the best possible style, should allow easy reading and should not require excessive comments. It is always preferable to have clearer code with few comments than obscure code with lots of comments.

The code should be modular, divided in functionalities with specific and reduced responsibilities. Each module should have a short comment describing its purpose.

You must implement two Java classes named `ist.meic.pa.GenericFunctions.GenericFunction` and `ist.meic.pa.GenericFunctions.GFMethod` containing the necessary methods and auxiliary classes to support the interactions presented above.

4 Presentation

For this project, a full report is not required. Instead, a public presentation is required. This presentation should be prepared for a 15-minute slot, should be centered in the architectural decisions taken and may include all the details that you consider relevant. You should be able to "sell" your solution to your colleagues and teachers.

5 Format

Each project must be submitted by electronic means using the Fénix Portal. Each group must submit a single compressed file in ZIP format, named as `project.zip`. Decompressing this ZIP file must generate a folder named `g##`, where `##` is the group's number, containing:

- the source code, within subdirectory `/src`
- an Ant file `build.xml` file that, by default, compiles the code and generates `genericFunctions.jar` in the same location where the file `build.xml` is located.

Note that it should be enough to execute

```
$ ant
```

to generate (`genericFunctions.jar`). In particular, note that the submitted project must be able to be compiled when unziped.

The only accepted format for the presentation slides is PDF. This PDF file must be submitted using the Fénix Portal separately from the ZIP file and should be named `p2.pdf`.

6 Evaluation

The evaluation criteria include:

- The quality of the developed solutions.
- The clarity of the developed programs.
- The quality of the public presentation.

In case of doubt, the teacher might request explanations about the inner workings of the developed project, including demonstrations.

The public presentation of the project is a compulsory evaluation moment. Absent students during project presentation will be graded zero in the entire project.

7 Plagiarism

It is considered plagiarism the use of any fragments of programs that were not provided by the teachers. It is not considered plagiarism the use of ideas given by colleagues as long as the proper attribution is provided.

This course has very strict rules regarding what is plagiarism. Any two projects where plagiarism is detected will receive a grade of zero.

These rules should not prevent the normal exchange of ideas between colleagues.

8 Final Notes

Don't forget Murphy's Law.

9 Deadlines

The code must be submitted via Fénix, no later than 19:00 of **May**, **13**. The slides must be submitted via Fénix, no later than 19:00 of **May**, **16**.

The presentations will be done during the classes after the deadline. Only one element of the group will present the work and the presentation must not exceed 15 minutes. The element will be chosen by the teacher just before the presentation. Note that the grade assigned to the presentation affects the entire group and not only the person that will be presenting. Note also that content is more important than form. Finally, note that the teacher may question any member of the group before, during, and after the presentation.