Generic Functions in Java

António Menezes Leitão

April, 2018

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.

Currently, it is not possible to implement this concept natively in Java, but we can implement a poor man's approach using Java's static methods and annotations. Here is an example:

```java
@GenericFunction
interface Com {

    public static Object bine(Object a, Object b) {
        Vector<Object> v = new Vector<Object>();
        v.add(a);
        v.add(b);
        return v;
    }

    public static Integer bine(Integer a, Integer b) {
        return a + b;
    }

    public static Object bine(String a, Object b) {
        return a + ", " + b + "!";
    }

    public static Object bine(String a, Integer b) {
        return (b == 0) ? "" : a + bine(a, b - 1);
    }
}
```

The previous fragment creates the `Com.bine` generic function, and adds four methods to the function, one specialized for `Object`s, another specialized for `Integer`s and, finally, two additional methods specialized on `String` and `Object` and on `String` and `Integer`.

Here is a fragment of a Java program that uses the previous generic function:

```java
Object[] objs1 = new Object[] { "Hello", 1, 'A' };
Object[] objs2 = new Object[] { "World", 2, 'B' };
for (Object o1 : objs1) {
  for (Object o2 : objs2) {
    System.out.println("Combine(" + o1 + ", " + o2 + ") -> " + Com.bine(o1, o2));
  }
}
```

Now, given that Java only uses the static type of the arguments to resolve static method calls, it is not surprising to see that the execution of the fragment above does not produce the intended results:

```
Combine(Hello, World) -> [Hello, World]
Combine(Hello, 2) -> [Hello, 2]
Combine(Hello, B) -> [Hello, B]
Combine(1, World) -> [1, World]
Combine(1, 2) -> [1, 2]
Combine(1, B) -> [1, B]
Combine(A, World) -> [A, World]
Combine(A, 2) -> [A, 2]
Combine(A, B) -> [A, B]
```

Obviously, we would prefer to see the following behavior:

```
Combine(Hello, World) -> Hello, World!
Combine(Hello, 2) -> HelloHello
Combine(Hello, B) -> Hello, B!
Combine(1, World) -> [1, World]
Combine(1, 2) -> 3
Combine(1, B) -> [1, B]
Combine(A, World) -> [A, World]
Combine(A, 2) -> [A, 2]
Combine(A, B) -> [A, B]
```

As a more dramatic example, consider the following program that mixes colors:

```
@GenericFunction
class Color {
    public static String mix(Color c1, Color c2){
        return mix(c2, c1);
    }

    public static String mix(Red c1, Red c2) {
        return "More red";
    }

    public static String mix(Blue c1, Blue c2) {
        return "More blue";
    }

    public static String mix(Yellow c1, Yellow c2) {
        return "More yellow";
    }

    public static String mix(Red c1, Blue c2) {
        return "Magenta";
    }

    public static String mix(Red c1, Yellow c2) {
        return "Orange";
    }

    public static String mix(Blue c1, Yellow c2){
        return "Green";
    }
}

class Red extends Color {
}

class Blue extends Color {
}

class Yellow extends Color {
}
```

Here is a fragment of code that tries different mixes of colors:

```
Color[] colors = new Color[] { new Red(), new Blue(), new Yellow() };
for (Color c1 : colors) {
    for (Color c2 : colors) {
        System.out.println(Color.mix(c1, c2));
    }
}
```

Unfortunately, when we execute the program under the standard Java evaluation rules, we immediately get infinite recursion. On the other hand, using the Common Lisp evaluation rules for generic functions, the program fragment prints:

```
More red
Magenta
Orange
Magenta
More blue
Green
Orange
Green
More yellow
```

Another important feature of the concept of generic function is the role that methods can take in the effective method. In fact, the CLOS *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, respectively.

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

```
@GenericFunction
interface Explain {
    public static void it(Integer i) {
        System.out.print(i + " is an integer");
    }

    public static void it(Double i) {
        System.out.print(i + " is a double");
    }

    public static void it(String s) {
        System.out.print(s + " is a string");
    }

    @BeforeMethod
    public static void it(Number n) {
        System.out.print("The number ");
    }

    @AfterMethod
    public static void it(Object o) {
        System.out.println(".");
    }
}
```

The generic function is used in the following fragment:

```
Object[] objs = new Object[] { "Hello", 1, 2.0 };
for (Object o : objs) {
    Explain.it(o);
}
```

which, under Java semantics, produces the following useless behavior:

```
.
.
.
```

while, under CLOS semantics, we get instead:

```
Hello is a string.
The number 1 is an integer.
The number 2.0 is a double.
```

2 Goals

The main goal of this project is the implementation of generic functions in Java, following the syntax and semantics previously described. The classes, interfaces, and annotations should be implemented in the package `ist.meic.pa.GenericFunctions`.

You must also implement a Java class named `ist.meic.pa.GenericFunctions.WithGenericFunctions` containing a static method `main` that accepts, as arguments, the name of another Java program (i.e., a Java class that also contains a static method `main`) and the arguments that should be provided to that program. The class should (1) operate the necessary transformations to the loaded Java classes so that when the classes are executed, the semantics of method calls to methods defined in classes or interfaces annotated as `@GenericFunction` follow CLOS semantics, and (2) should transfer the control to the `main` method of the program.

Note that the implementation only needs to support *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.

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.

Examples of interesting extensions include:

- Caching of effective methods
- Support for *around* methods
- Additional method combinations (e.g., *and*, *or*, *list*)

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 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.

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/main/java`
- A Gradle file `build.gradle` that, upon execution of the task build, generates a `genericFunctions.jar` in the `build/libs/` folder.

Note that it should be enough to execute

```
$ gradle compileJava build
```

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 `p1.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 **April**, **24**. Similarly, the presentation must be submitted via Fénix, no later than 19:00 of **April**, **24**.

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.