Method Combination in Java

António Menezes Leitão

March, 2022

1 Introduction

The concept of method combination enables a declarative approach for building programs based on independent fragments. Each fragment is called a method and its composition is called an effective method. Each method has an associated role indicating what form of participation the method has in the effective method. The roles allow, for example, methods that run before running an existing method, or that run after an existing method.

2 Goals

The main goal of this project is the implementation, in Java, of mechanisms for method combination as similar as possible to the analogous mechanisms that are pre-defined in CLOS. Note that you don't need to implement multiple dispatch nor the equivalent to CLOS' `call-next-method` form.

At the very least, you need to implement:

- The *simple method combination* supporting the `and` and `or` qualifiers.
- The *standard method combination* supporting the `before` and `after` qualifiers.

In order to use these combination mechanisms, it is necessary to annotate the participant methods. To that end, you need to implement the Java *annotation* `@Combination` (in package `ist.meic.pava`) that signals that a given method declaration must be treated as a fragment for method combination. The annotation needs to be parameterized with the method combination intended.

In the next sections, we exemplify the different kinds of method combination.

2.1 Simple Method Combination

Consider the following example and focus on the predicate `isHardWorker` where, besides the `@Combination` annotation, we also added a debugging `println` to better understand the behavior:

```
class Person {
    String name;

    public Person(String name) {
        this.name = name;
    }

    @Combination("or")
    public boolean isHardWorker() {
        System.out.println("Person.isHardWorker.");
        return false;
    }
}
```

```
class Student extends Person {

    String school;

    public Student(String name, String school) {
        super(name);
        this.school = school;
    }

    @Combination("or")
    public boolean isHardWorker() {
        System.out.println("Student.isHardWorker.");
        return isGoodSchool(school);
    }

    public boolean isGoodSchool(String school) {
        return school.equals("FEUP") || school.equals("FCT");
    }
}


interface HardWorker {
    @Combination("or")
    default boolean isHardWorker() {
        System.out.println("HardWorker.isHardWorker.");
        return true;
    }
}


class ISTStudent extends Student implements HardWorker {

    public ISTStudent(String name) {
        super(name, "IST");
    }
}
```

When the method `isHardWorker` is called, the resulting behavior is the short-circuiting disjunction of all the `isHardWorker` methods that are applicable. As a result, the following program fragment:

```
Person[] people = new Person[] { new Person("Mary"),
                                 new Student("John", "FEUP"),
                                 new Student("Lucy", "XYZ"),
                                 new ISTStudent("Fritz") };
for (Person person : people) {
    System.out.println(person.name + " is a hard worker? " + person.isHardWorker());
}
```

prints:

```
Person.isHardWorker.
Mary is a hard worker? false
Student.isHardWorker.
John is a hard worker? true
Student.isHardWorker.
Person.isHardWorker.
Lucy is a hard worker? false
HardWorker.isHardWorker.
Fritz is a hard worker? true
```

Note that if we were using regular Java (i.e., without method combination), the output would be:

```
Person.isHardWorker.
Mary is a hard worker? false
Student.isHardWorker.
John is a hard worker? true
Student.isHardWorker.
Lucy is a hard worker? false
Student.isHardWorker.
Fritz is a hard worker? false
```

2.2 Standard Method Combination

To illustrate standard method combination, consider the following changes to the previous example to include a `print_name` method, where the role of the method in the combination is described using a prefix in the method name.

```
class Person {
    String name;

    public Person(String name) {
        this.name = name;
    }

    @Combination("standard")
    public void print_name() {
        System.out.print(name);
    }

    @Combination("or")
    public boolean isHardWorker() {
        return false;
    }
}


class Student extends Person {

    String school;

    public Student(String name, String school) {
        super(name);
        this.school = school;
    }

    @Combination("standard")
    public void before_print_name() {
        System.out.print("Student ");
    }

    @Combination("or")
    public boolean isHardWorker() {
        return isGoodSchool(school);
    }

    public boolean isGoodSchool(String school) {
        return school.equals("FEUP") || school.equals("FCT");
    }
}


interface HardWorker {
    @Combination("or")
    default boolean isHardWorker() {
        return true;
    }
}
```

```
class ISTStudent extends Student implements HardWorker {

    public ISTStudent(String name) {
        super(name, "IST");
    }

    @Combination("standard")
    public void before_print_name() {
        System.out.print("IST-");
    }
}


interface Foreign {
    @Combination("standard")
    default void before_print_name() {
        System.out.print("Foreign ");
    }

    @Combination("standard")
    default void after_print_name() {
        System.out.print(" (presently in Portugal)");
    }
}


class ForeignStudent extends Student implements Foreign {
    public ForeignStudent(String name, String school) {
        super(name, school);
    }
}


class ForeignISTStudent extends ISTStudent implements Foreign {
    public ForeignISTStudent(String name) {
        super(name);
    }
}


class ForeignHardworkingPerson extends Person implements Foreign, HardWorker {
    public ForeignHardworkingPerson(String name) {
        super(name);
    }
}
```

Using the previous definitions, the following fragment:

```
Person[] people = new Person[] { new Person("Mary"),
                                 new Student("John", "FEUP"),
                                 new Student("Lucy", "XYZ"),
                                 new ISTStudent("Fritz"),
                                 new ForeignStudent("Abel", "FCT"),
                                 new ForeignISTStudent("Bernard"),
                                 new ForeignHardworkingPerson("Peter") };
for (Person person : people) {
    person.print_name();
    System.out.println(" is a hard worker? " + person.isHardWorker());
}
```

produces the following output:

```
Mary is a hard worker? false
Student John is a hard worker? true
Student Lucy is a hard worker? false
IST-Student Fritz is a hard worker? true
Foreign Student Abel (presently in Portugal) is a hard worker? true
Foreign IST-Student Bernard (presently in Portugal) is a hard worker? true
Foreign Peter (presently in Portugal) is a hard worker? true
```

Without method combination, the output would have been:

```
Mary is a hard worker? false
John is a hard worker? true
Lucy is a hard worker? false
Fritz is a hard worker? false
Abel is a hard worker? true
Bernard is a hard worker? false
Peter is a hard worker? false
```

You must implement the required functionality in a file named `UsingMethodCombination.java`. As an example, if the previous definitions were contained in a Java file named `HardWorkers.java` (alongside an `import` of `ist.meic.pava.Combination` and an appropriate `main` method), after compilation the file could be executed as:

```
$ java -classpath javassist.jar:. ist.meic.pava.UsingMethodCombination HardWorkers
```

2.3 Extensions

You can extend your project to further increase your final grade. Note that this increase will not exceed **two** points.

Examples of interesting extensions include:

- Support *around* methods.
- Support *call next method*.
- Support other forms of simple method combination.

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

3 Code

Your implementation must work in Java 8 and should use the *bytecode* manipulation tool Javassist, version `3.28` available at `https://github.com/jboss-javassist/javassist/releases/tag/rel_3_28_0_ga`.

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 a Java class named `ist.meic.pava.UsingMethodCombination` 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 the method combinations are done, and (2) should transfer the control to the `main` method of the program.

4 Presentation

For this project, a full report is not required. Instead, a public presentation is required. This presentation should be prepared for a 10-minute slot, should be centered on 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. You should be prepared to answer questions regarding both the presentation and the actual solution. If necessary, you will be asked to discuss your work in full detail in a separate session.

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 `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/`.

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 `presentation.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.

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 plagiarism. Any two projects where plagiarism is detected will receive a grade of zero.

These rules should not prevent the healthy 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 23:00 of **April**, **??**. Similarly, the presentation must be submitted via Fénix, no later than 23:00 of **April**, **??**.

The presentations will be done during classes after the deadline. Only one element of the group will present the work. 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.