Multiple Dispatch for Java

António Menezes Leitão

March, 2021

1 Introduction

It is known that Java method calls use dynamic dispatch for the receiver and static dispatch for the arguments. Despite the advantages of this strategy for the efficiency of programs, it also creates a mismatch between the treatment of the receiver and that of the arguments.

Sometimes, this mismatch is not what the programmer wants, in particular, because it can lead to unexpected results. As an example, consider the following program:

```java
class Shape {
}

class Line extends Shape {
}

class Circle extends Shape {
}

class Device {
    public void draw(Shape s) {
        System.err.println("draw what where?");
    }
    public void draw(Line l) {
        System.err.println("draw a line where?");
    }
    public void draw(Circle c)  {
        System.err.println("draw a circle where?");
    }
}

class Screen extends Device {
    public void draw(Shape s) {
        System.err.println("draw what on screen?");
    }
    public void draw(Line l) {
        System.err.println("drawing a line on screen!");
    }
    public void draw(Circle c) {
        System.err.println("drawing a circle on screen!");
    }
}

class Printer extends Device {
    public void draw(Shape s) {
        System.err.println("draw what on printer?");
    }
    public void draw(Line l) {
        System.err.println("drawing a line on printer!");
    }
    public void draw(Circle c) {
        System.err.println("drawing a circle on printer!");
    }
}
```

Given the definitions in the previous program, it is frequently the case that programmers are surprised by the output of the following program fragment:

```
Device[] devices = new Device[] { new Screen(), new Printer() };
Shape[] shapes = new Shape[] { new Line(), new Circle() };
for (Device device : devices) {
    for (Shape shape : shapes) {
        device.draw(shape);
    }
}
```

In fact, instead of seeing the actual shapes being drawn on the actual devices, they see instead:

```
draw what on screen?
draw what on screen?
draw what on printer?
draw what on printer?
```

As is clear, only the devices were properly identified but the specific methods for `Line`s and `Circle`s were not used because the static dispatch used on the arguments only sees them as `Shape`s.

Other languages, including Common Lisp and Julia decided to follow a more dynamic approach by dynamically identifying the types of all participants of a method call and selecting the most specific method applicable to those participants, including not only the receiver of the method call but also all the arguments.

2 Goals

The main goal of this project is the implementation of multiple dispatch in Java. To that end, you will resort to Java's reflective capabilities. More specifically, you need to implement the class `ist.meic.pava.MultipleDispatch.UsingMultipleDispatch` containing a static method with the following signature:

```
static Object invoke(Object receiver, String name, Object... args)
```

The previous method should implement dynamic dispatch on the method named `name` using the actual type of all participants, i.e., receiver and all arguments, to select the most specific method that is applicable. For the purposes of this project, a method is considered applicable if it is a `public` method declared or inherited by the class of the receiver, and the type of each parameter matches (in the sense of the `isinstance` predicate) the corresponding argument. For the purposes of this project an applicable method m_1 is more specific than an applicable method m_2 when the declaring class of m_1 is a subclass of the declaring class of m_2. When the declaring classes are identical, then m_1 is more specific than m_2 if the type of the first parameter of m_1 is a subtype of the type of the first parameter of m_2. If these types are identical then we repeat the process with following parameters, from left to right, until we find one that is a subtype of the other.

As an example of its use, consider the following change in the previous program:

```
Device[] devices = new Device[] { new Screen(), new Printer() };
Shape[] shapes = new Shape[] { new Line(), new Circle() };
for (Device device : devices) {
    for (Shape shape : shapes) {
        //device.draw(shape);
        UsingMultipleDispatch.invoke(device, "draw", shape);
    }
}
```

Using the new version, the output now becomes:

```
drawing a line on screen!
drawing a circle on screen!
drawing a line on printer!
drawing a circle on printer!
```

For a more complex example, consider the following classes:

```
class Shape {
}

class Line extends Shape {
}

class Circle extends Shape {
}

class Brush {
}

class Pencil extends Brush {
}

class Crayon extends Brush {
}

class Device {
    public void draw(Shape s, Brush b) {
        System.err.println("draw what where and with what?");
    }
    public void draw(Line l, Brush b) {
        System.err.println("draw a line where and with what?");
    }
    public void draw(Circle c, Brush b) {
        System.err.println("draw a circle where and with what?");
    }
}

class Screen extends Device {
    public void draw(Line l, Brush b) {
        System.err.println("draw a line where and with what?");
    }
    public void draw(Line l, Pencil p) {
        System.err.println("drawing a line on screen with pencil!");
    }
    public void draw(Line l, Crayon c) {
        System.err.println("drawing a line on screen with crayon!");
    }
    public void draw(Circle c, Brush b) {
        System.err.println("drawing a circle on screen with what?");
    }
    public void draw(Circle c, Pencil p) {
        System.err.println("drawing a circle on screen with pencil!");
    }
}

class Printer extends Device {
    public void draw(Line l, Brush b) {
        System.err.println("drawing a line on printer with what?");
    }
    public void draw(Circle c, Pencil p) {
        System.err.println("drawing a circle on printer with pencil!");
    }
    public void draw(Circle c, Crayon r) {
        System.err.println("drawing a circle on printer with crayon!");
    }
}
```

and the following example that relies on *triple* dispatch:

```java
Device[] devices = new Device[] { new Screen(), new Printer() };
Shape[] shapes = new Shape[] { new Line(), new Circle() };
Brush[] brushes = new Brush[] { new Pencil(), new Crayon() };
for (Device device : devices) {
    for (Shape shape : shapes) {
        for (Brush brush : brushes) {
            UsingMultipleDispatch.invoke(device, "draw", shape, brush);
        }
    }
}
```

Note that the output should be:

```
drawing a line on screen with pencil!
drawing a line on screen with crayon!
drawing a circle on screen with pencil!
drawing a circle on screen with what?
drawing a line on printer with what?
drawing a line on printer with what?
drawing a circle on printer with pencil!
drawing a circle on printer with crayon!
```

You must implement the required functionality in a single file named `MultipleDispatch.java`.

2.1 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:

- Dealing not only with the class hierarchy but also the interface hierarchy.
- Dealing with *boxing* and *unboxing* of arguments.
- Dealing with variable arity methods.

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.pa.MultipleDispatchExtended`.

3 Code

Your implementation must work in Java 8 but you can also use more recent versions.

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 into 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 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**, **9**. Similarly, the presentation must be submitted via Fénix, no later than 23:00 of **April**, **9**.

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.