Keyword Arguments in Java

António Menezes Leitão

March, 2017

1 Introduction

Modern programming languages, such as Python or Common Lisp, support *keyword arguments*, also known as *named parameters*. This feature allows a function call to specify the name of the parameter to which a given argument applies. It also simplifies the use of optional arguments, as it makes it easy to pass only a subset of the needed arguments. As an example, consider the following Python function:

```
def create_widget(width=100, height=50, margin=10):
    ...
```

The previous function expects three arguments but all of them are optional. This means that all the following function calls are valid:

```
create_widget()
create_widget(width=80)
create_widget(height=30)
create_widget(height=20, width=90)
```

This feature is particularly useful in the construction of objects that contain many fields, as it makes it easier to specify the initialization of those fields.

Unfortunately, the Java programming language does not support keyword arguments and the proposed alternatives also have considerable drawbacks (e.g., the infamous *constructor madness*). This has been a recurrent source of complaints in Java and there is growing interest in supporting keyword arguments.

Interestingly, it is not difficult to imagine a Java extension that supports keyword arguments, at least, for the construction of objects. As an example, consider the following annotated code:

```
class Widget {
    int width;
    int height;
    int margin;

    @KeywordArgs("width=100,height=50,margin")
    public Widget(Object ...args) {}

    public String toString() {
        return String.format("width:%s,height:%s,margin:%s",
                             width, height, margin);
    }
}
```

Note that the constructor takes a variable number of arguments and has an empty body. The `KeywordArgs` annotation is used to inform that `width`, `height`, and `margin` are keyword arguments and the first two have the default values `100` and `50`, while the third one does not have a default value.

In order to create an instance of the class, we need to use the following syntax:

```
new Widget("width", 100, "height", 50, "margin", 5)
```

Obviously, the advantage of keyword arguments becomes visible when we want to change the order of the arguments, or when we do not want to provide all of them. As an example, consider the following code fragment:

```
System.out.println(new Widget());
System.out.println(new Widget("width", 80));
System.out.println(new Widget("height", 30));
System.out.println(new Widget("height", 20, "width", 90));
```

that prints the following:

```
width:100,height:50,margin:0
width:80,height:50,margin:0
width:100,height:30,margin:0
width:90,height:20,margin:0
```

Note that a parameter that does not have a corresponding argument and that does not have a default value receives the Java default initialization for its type.

As a more interesting example, consider the following subclass of `Widget`:

```
class ExtendedWidget extends Widget {
    String name;

    @KeywordArgs("name=\"Extended\",width=200,margin=10,height")
    public ExtendedWidget(Object... args) {}

    public String toString() {
        return String.format("width:%s,height:%s,margin:%s,name:%s",
                             width, height, margin, name);
    }
}
```

and the corresponding example of use:

```
System.out.println(new ExtendedWidget());
System.out.println(new ExtendedWidget("width", 80));
System.out.println(new ExtendedWidget("height", 30));
System.out.println(new ExtendedWidget("height", 20, "width", 90));
System.out.println(new ExtendedWidget("height", 20, "width", 90, "name", "Nice"));
```

Now, the output is as follows:

```
width:200,height:50,margin:10,name:Extended
width:80,height:50,margin:10,name:Extended
width:200,height:30,margin:10,name:Extended
width:90,height:20,margin:10,name:Extended
width:90,height:20,margin:10,name:Nice
```

Note, from the previous output, that defaults are inherited but can be overriden. That is what happens, for example, with the `margin` parameter.

Another important feature is that incorrect keywords are detected and reported. Consider, for example, the call:

```
System.err.println(new ExtendedWidget("foo", 1, "bar", 2));
```

which causes the exception:

```
Exception in thread "main" java.lang.RuntimeException: Unrecognized keyword: foo
        at ExtendedWidget.<init>(Test.java)
        ...
```

In fact, whenever a keyword is passed that is not present in the list of accepted keywords, a `RuntimeException` must be throwed, with the message `Unrecognized keyword:` followed by a space and by the name of the first unrecognized keyword.

2 Goals

The main goal of this project is the implementation of keyword parameters in Java, using the syntax and semantics previously described.

More specifically, you need to implement the `KeywordArgs` annotation and a Javassist-based translator named `ist.meic.pa.KeyConstructors`. Assuming we have a Java bytecode file named `Test.class` containing classes which have a `KeywordArgs`-annotated constructor, we should be able to execute the program as follows:

```
$ java ist.meic.pa.KeyConstructors Test
```

In order to simplify your task, you can assume that the default initializer of a keyword argument is a correct Java expression compilable by Javassist.

You need to be careful in order to support the most natural semantics for the keyword arguments. For example, assuming that class `Bar` contains fields `a` and `b`, consider the difference between defining the keyword constructor as:

```
@KeywordArgs("a=1,b=a")
public Bar(Object ...args) {}
```

or as:

```
@KeywordArgs("b=a,a=1")
public Bar(Object ...args) {}
```

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.KeyConstructorsExtended`.

Some interesting extensions are:

- Support repeated keywords in method calls with appropriate semantics.
- Support keyword arguments in method definitions.
- Elimination of keyword arguments at load time, to speedup method calls.

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`

- an Ant file `build.xml` file that, by default, compiles the code and generates `keyConstrutors.jar` in the same location where the file `build.xml` is located.

Note that it should be enough to execute

```
$ ant
```

to generate (`keyConstrutors.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 and the slides must be submitted via Fénix, no later than 23:59 of **March**, **28**.

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.