Autoboxing in Java

António Menezes Leitão

March, 2016

1 Introduction

It is known that Java separates types into *primitive* types, such as `int` and `double`, and *reference* types, such as `Long` and `String`, and these two type hierarchies are not compatible. On the other hand, since Java 1.5, primitive types can be automatically converted to their corresponding *wrapper* type. This means that an `int` will be automatically converted to the corresponding `Integer` when the need arises, as is visible in the following example:

```
Vector nums = new Vector();
int i = 5;
nums.add(i);
System.out.println(nums.get(0).getClass());
```

When executed, the previous code fragment prints `class java.lang.Integer`, showing that the value of the primitive `int` variable `i` was converted to the corresponding value of the `Integer` reference type. This language feature is known as *autoboxing*.

Similarly, a value of one of the wrapper types can be automatically converted into a value of the corresponding primitive type, as the following example shows:

```
ArrayList<Integer> Nums = new ArrayList<Integer>();
Nums.add(new Integer(42));
int j = Nums.get(0);
System.out.println(j);
```

When executed, the previous code fragment prints `42`, showing that the value was correctly converted from an `Integer` reference type to an `int` type.

Despite the convenience of automatic boxing and unboxing, there might be significant overheads included which might not be entirely obvious to the programmer. As a concrete example, consider the following fragment:

```
long sum = 0L;
for (int i = 0; i < 1000000000; i++) {
  sum += i;
}
```

and the slightly different version:

```
Long sum = 0L;
for (int i = 0; i < 1000000000; i++) {
  sum += i;
}
```

Although they differ by just one letter, the second fragment runs twenty times slower than the first.

Given that automatic boxing and unboxing can create performance problems that might be difficult to spot, it would be interesting to have a profiling tool that can measure the amount of automatic boxing and unboxing that is made by a given program.

2 Goals

Implementation of a boxing/unboxing *profiler* for Java. During the execution of the Java program the profiler will count all the boxing and unboxing operations that are being done and, at the end of the program execution, for each method, it prints those counters on `System.err`.

The profiler should be implemented as a class named `BoxingProfiler` defined in package `ist.meic.pa`.

As an example, consider the following program in file `SumInts.java`:

```
public class SumInts {

    private static long sumOfIntegerUptoN(int n) {
        long sum = 0L;
        for (int i = 0; i < n; i++) {
            sum += i;
        }
        return sum;
    }

    private static long printSum(long n) {
        System.out.println("Sum: " + n);
        return n;
    }

    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        printSum(sumOfIntegerUptoN(100000000));
        long end = System.currentTimeMillis();
        System.out.println("Time: " + (end - start));
    }
}
```

In my machine, the program execution prints:

```
$ java SumInts
Sum: 4999999950000000
Time: 28
```

When we profile the previous program, we notice that there is almost no penalty in the execution and no statistics are printed:

```
$ java ist.meic.pa.BoxingProfiler SumInts
Sum: 4999999950000000
Time: 29
```

Now, consider the slight variation in file `SumIntegers.java`:

```
public class SumIntegers {

    private static long sumOfIntegerUptoN(Integer n) {
        Long sum = 0L;
        for (int i = 0; i < n; i++) {
            sum += i;
        }
        return sum;
    }

    private static long printSum(Long n) {
        System.out.println("Sum: " + n);
        return n;
    }

    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        printSum(sumOfIntegerUptoN(100000000));
        long end = System.currentTimeMillis();
        System.out.println("Time: " + (end - start));
    }
}
```

This time, we get:

```
$ java SumIntegers
Sum: 4999999950000000
Time: 274
$ java ist.meic.pa.BoxingProfiler SumIntegers
Sum: 4999999950000000
Time: 4436
SumIntegers.main(java.lang.String[]) boxed 1 java.lang.Integer
SumIntegers.main(java.lang.String[]) boxed 1 java.lang.Long
SumIntegers.printSum(java.lang.Long) unboxed 1 java.lang.Long
SumIntegers.sumOfIntegerUptoN(java.lang.Integer) unboxed 100000001 java.lang.Integer
SumIntegers.sumOfIntegerUptoN(java.lang.Integer) boxed 100000001 java.lang.Long
SumIntegers.sumOfIntegerUptoN(java.lang.Integer) unboxed 100000001 java.lang.Long
```

It is important to note that the profiler output must be printed in `System.err`, *sorted*, firstly by Javassists' `getLongName` for behaviors, secondly by Javassists' `getName` for classes, and thirdly, by boxing and then unboxing operations.

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 class named `ist.meic.pa.BoxingProfilerExtended`.

3 Code

Your implementation must work in Java 7 and should use the *bytecode* manipulation tool Javassist, version `3.20`.

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.pa.BoxingProfiler` 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, their execution is profiled, 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 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`
- the slides of the presentation, in a file named `p1.pdf`.
- an Ant file `build.xml` file that, by default, compiles the code and generates `boxingProfiler.jar` in the same location where the file `build.xml` is located.

Note that it should be enough to execute

```
$ ant
```

to generate (`boxingProfiler.jar`). In particular, note that the submitted project must be able to be compiled when unziped and, as such, this means that you must include the file javassist.jar in the ZIP file.

The only accepted format for the presentation slides is PDF. This file must be located at the root of the ZIP file and must have the name `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 19:00 of **March**, **29**.

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.