Effective type parametrization in Java

Cite as: AIP Conference Proceedings **2116**, 350007 (2019); https://doi.org/10.1063/1.5114360
Published Online: 24 July 2019

Péter Soha, and Norbert Pataki

Effective Type Parametrization in Java

Péter Soha[b] and Norbert Pataki[a]

Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University, Pázmány Péter sétány 1/C, H-1117, Budapest, Hungary

[a]Corresponding author: patakino@elte.hu
[b]sohaur@inf.elte.hu

Abstract. Type parametrization is an essential construct in modern programming languages, Java offers generics, C++ offers templates for highly reusable code. The mechanism between these constructs differs and affects usage and runtime performance, as well. Java uses type erasure, C++ deals with instantiations.

In this paper, we argue for a new approach in Java which is similar to C++ template construct. The proposed method contains a template syntax and a developed tool which can instantiate Java templates and generates Java source code. The templates have advantages related to special template parameters that are not supported by generics. On the other hand, the instantiations have to be processed at compilation time, but the runtime performance of the code is improved. We measure how the Java templates speed-up the execution.

INTRODUCTION

Programming languages differ in syntax, semantics and implementation, as well. Similar constructs of different programming languages are able to work in a totally different way, for instance, what is the execution model, how the parameters are passed to subprograms or how the separate modules are merged.

Type parametrization is an essential language construct in modern programming languages which extends type systems with type abstractions. One can write generic data structures and algorithms, type-independent solutions for memory management with this feature. Standard and non-standard libraries take advantage of type parametrization [8].

Programming languages handle type parametrization differently, as well. C++ supports *templates* that contain placeholders for compilation time parameters [9]. Java offers *generics* that use type erasure and have runtime overhead because type parametrization is expressed with inheritance and late binding [1]. C++ templates are more efficient at runtime because the compiler is aware of the template arguments so it can optimize the code (e.g. function calls) [6]. C++ template is an essential construct regarding effective runtime [10]. Generic and template constructs have been analysed regarding their performance, but templates in Java have not been evaluated [3].

In this paper, we argue for a new approach that aims at C++-like templates in the Java programming language. We have developed a tool for this new approach which instantiates Java templates.

This rest of this paper is organized as follows: our solution is presented in section *OUR APPROACH*. We evaluate the performance of the proposed solution in section *MEASUREMENTS* and finally, the paper is concluded in section *CONCLUSION*.

OUR APPROACH

Technical background

Although, Java generics is quite versatile and useful part of the language, all of the advantages has its own cost. Every time, when the the compiler compiles a generic class or function, deletes the type and replace it with `Object` or a more specific interface, and the actual type of the elements only revealed at runtime. On the one hand, this is comfortable for developers, because a generic can be use with many types without re-compilation. But on the other hand this is a very wasteful practice if we want to save resources and CPU time. In contrast of the generics, our solution, the Java templates offers statically typed constructs and faster run for the cost of runtime type change. The structure of Java templates is similar to generic classes in Java only differs in two important parts which are the template keyword at the beginning and the instantiation at the end. We have implemented a tool which generates C++ macro from template and uses C/C++ preprocessor to create Java class from the macro.

About the Java templates

Template parameters and package names

The Java template (as we discussed earlier) consists of three main parts. The first one is the declaration. We granted the template keyword for this which must be followed by the comma-separated sequence of template parameters in curly bracket (which must have at least one element). At compilation time, these parameters are replaced with the given types (will be discussed later). Also, our tool uses these types too and generates the path for the Java class. To assure the correctness of the path, we formulated the following restrictions (let X is the formal parameter of the template):

- If X is a primitive type or literal, there is no limitation.
- If X is an object type the fully qualified name is required, because not the template nor our tool can handle the import declarations right now and this can lead to compile time errors.

To comply with the Java conventions of package names, we declared the following restrictions:

- If type parameter X is a primitive or a literal, we add an "a" prefix (which means the result will be "aX".
- If type parameter X is an object the fully qualified name will be (part of) the package name, but the separator points being replaced with backslash.
- Any package names can be created with the finitely many use of the previous two rules.

To avoid any OS-specific issue we recommend to use shorter names than 260 characters.

The instantiation

Second of all, we discuss the instantiation. After the body of the template (which will be presented later) you can give an instantiation of the template with the actual types just like C/C++ macros. To avoid any possible errors the we suggest the following:

- The number of actual parameters must be equal to (but at least not less) the number of type parameters
- With object types the fully qualified name is required
- A parameter can be replaced with a literal if, and only if that parameter only occurs on right hand side of an expression, or where literals are allowed by grammatical rules of Java.

About our solution

Our tool which instantiates Java templates uses the C++ preprocessor to generate the Java code using the type names given from command line arguments. The progress of processing contains the following steps:

- Read the template and tokenize the specific parts
- Complete the template to standardized C++ macro
- Generate the package name from the given types
- Insert instantiation command to the end of the macro
- Preprocessing the macro with C++ preprocessor and generates Java source code

Template in practice

Last we give an example which demonstrates, how templates improve performance compared to generics with a minimal additional cost. Let us implement a stack type which uses an underlying array to store elements and have the common `push` and `pop` functions (let us suppose that the array has been instantiated with a specific size and the cursor is change correctly after each step). Now let us see how the template looks like:

```
//Stack template
template(T)
class Stack {
  private T[] elements;
  // ...
  public void push(T item) { elements[c++] = item; }
  public T pop() { return elements[c--]; }
}
```

The generic version is the following:

```
//generic Stack
class Stack<T> {
  private Object[] elements;
  // ...
  public void push(T item) { elements[c++] = item; }
  public T pop() { return (T)elements[c--]; }
}
```

Now instantiate these implementations with integer types. With the template, we can use the primitive `int` type but the generics only allows the `Integer` (which is an object and the wrapper of `int`). The following major differences will show up during runtime:

- In the template, all occasions of formal type `T` replaced with `int` which also compiled into internal (or byte) code, but in generics the JVM must erase type `T` and replace it with `Object` (type erasure), and in bytecode, we can see that we work with objects in background.
- In templates, the `push` takes an `int` argument and store it in an array of `int`s. In generic the argument is an `Integer`, so if we pass an `int` as actual parameter, the JVM has to box it into an `Integer` and store it in an array of `Object`s.
- In template, the `pop` also does not need any additional operation to get the top element from `Stack` and give it to the caller. In generic since we store `Object`s, we must use an explicit cast from `Object` to `Integer` (and on the call side also need an unboxing to `int`).

Another possibility with the proposed solution that one cannot catch type parametrized exception classes. So far, the Java generics mechanism does not offer this feature [4]. With our solution, one can write template exception classes and can catch instances of these templates properly.

C++ templates can be parametrized with integral constants (e.g. bool literals, such as `true`, `false` and integers, such as `5`) [7]. The value of these arguments must be known at compilation time. Java generics do not offer this kind of parameters, only type parameters are allowed. Our solution also supports these special template parameters and can be handled by the compiler.

MEASUREMENTS

In this section we present the performance of our solution. We focus on the runtime performance because this property is more important than compilation time. The preprocessor has I/O-intensive tasks, so its performance depends on the storage device [2]. Moreover, a compiler support approach would be more effective in which the instantiation is executed on constructed abstract syntax tree.

We have evaluated how the proposed approach affects the runtime. We have started a cloud-based virtual machine with Ubuntu 16.04 LTS operating system image and Java 8 JVM installed. We evaluate two different scenarios with high number of test cases. We use our stack data structure implementations.

The first scenario is using stack that contains integers. The generic implementation must be used with `Integer`s, the template version can be instantiated with `int`. This approach avoids the autoboxing between `int` and `Integer`.

We have instantiated the stack with `Integer` in the second scenario. In this case, the template and generic parameter is exactly the same. However, the template stack itself knows that it contains `Integer`, not `Object`, so less runtime validations are needed in this case, as well.

Our approach performed better in both scenarios. The performance is improved significantly in the first scenario. The average running time of the long-term performance test has beeen reduced to 2.63% of the generic approach with our template mechanism. We measure this speed-up when the size of stack was 8000000. We fulfilled the stack with 8000000 `push` operations and after we used `pop` functions until the stack becomes empty. High amount of dynamic memory allocation and autoboxing conversion can be avoided with Java templates in this case. The results were rather balanced when both stacks contain `Integer` objects. In the second scenario the average running time has been reduced to 82.645% with the proposed approach. This means more than 20% speed-up in the execution without any special instantiation and special ones are able to speed-up the execution significantly. However, more effective code can be generated with more specific approaches [5].

CONCLUSION

Programming languages may use totally different underlying implementations for similar functionality. For instance, type parametrization does differ in C++ and Java. In this paper, we argue for a template-like type parametrization in Java. This approach supports sophisticated template parameters and improves the runtime performance, as well. We have created a new tool that instantiates Java templates. This solution overcomes many restrictions that Java generics deal with. However, according to our measurement, significant speed-up can be achieved with this approach, but a real compiler support would be more efficient.

ACKNOWLEDGEMENTS

The project was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

REFERENCES

[1] K. Arnold, J. Gosling and D. Holes, *The Java Programming Language*, Fourth Edition, 928 pages (Addison-Wesley, 2005)

[2] B. Babati, N. Pataki and Z. Porkoláb, "C/C++ Preprocessing with Modern Data Storage Devices", in *Proceedings of the 13th International Scientific Conference on Informatics*, IEEE, pp. 36–40 (2015).

[3] L. Dragan and S. M. Watt, "Performance analysis of generics in scientific computing", in *Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2005*, IEEE, pp. 93–100 (2005)

[4] D. Ghosh, *Generics in Java and C++: a comparative model*, ACM SIGPLAN Notices **39(5)**, pp. 40-47, (2004).

[5] G. Horváth, N. Pataki and M. Balassi, "Code Generation in Serializers and Comparators of Apache Flink", in *Proceedings of the 12th Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems Workshop (ICOOOLPS 2017)*, pp. 5(1)–5(6), ACM Digital Library, (2017).

[6] S. Meyers, *Effective STL*, 288 pages, (Addison-Wesley, 2001).

[7] N. Pataki, *Testing by C++ template metaprograms*, Acta Universitatis Sapientiae Informatica **2(2)**, pp. 154–167 (2010).

[8] N. Pataki and Z. Porkoláb, "Extension of iterator traits in the C++ Standard Template Library", in *Proceedings of the 2011 Federated Conference on Computer Science and Information Systems*, IEEE Computer Society Press, edited by M. Ganzha, L. A. Maciaszk, and M. Paprzycki, pp. 911–914 (2011).

[9] B. Stroustrup, *The C++ Programming Language*, Third Edition, 1040 pages (Addison-Wesley, 2000).

[10] Z. Szűgyi, Á Sinkovics, N. Pataki and Z. Porkoláb: *C++ metastring library and its applications*, in *Proceedings of Generative and Transformational Techniques in Software Engineering 2009*, Lecture Notes in Computer Science **6491**, pp. 461–480 (2010).