Acta Cybernetica — *online–first paper version* — pages 1–12.

Instantiation of Java Generics

Péter Soha[ab] and Norbert Pataki[ac]

Abstract

Type parametrization is an essential construct in modern programming languages. On one hand, Java offers generics, on the other hand, C++ provides 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 an approach in Java which is similar to C++ template construct. We evaluate the runtime performance of instantiated code and we present our tool which is able to use Java generics as templates. This tool generates Java source code. We present how this approach improves the usage of Java generics.

Keywords: Java, generic, instantiation, template

1 Introduction

Nowadays we can choose from many different programming paradigms and languages and all have unique advantages and disadvantages. We have to think different when programming in *Java* or *Clean*, when we use the *object-oriented* or the *functional* paradigm. Sometimes we wish that some elements of a language would be supported by another language though. One of these useful tools is the construct of reusable and parametrizable code which significantly reduces the repetition of the code. For this, the *Java* offers the *generics* that uses a runtime polymorphic solution with some transformations at compilation time. On the other hand, the *C++* language provides the *templates*.

Java is considered as a verbose language, therefore Java source typically contains boilerplate code [10]. To overcode the boilerplate, many libraries have been developed, such as Project Lombok [1]. Lombok decreases the quantity of boilerplate code by generating Java code from annotations. However, this solution is not able to generate similar new classes from templates.

[a]Department of Programming Languages and Compilers, Eötvös Loránd University, Budapest, Hungary

[b]E-mail: sohaur@inf.elte.hu, ORCID: 0000-0003-1556-8267

[c]E-mail: patakino@inf.elte.hu, ORCID: 0000-0002-7519-3367

DOI: 10.14232/actacyb.284073

The generics in Java fit into the object-oriented realm, but they have runtime overhead that can be reduced with compile-time instantiation. The required information can be found in the source, therefore performance can be improved without any limitation.

In this paper, we analyze how the template mechanism can be attempered in the Java programming language. We take into consideration how different languages provide type parametrization. Earlier, we have proposed an alternative syntax for instantiable templates for improved runtime performance [12]. At this time, we deal with standard Java code. For keeping the improved runtime performance, we developed a tool which aims at the instantiation of Java generics. Our tool works in the Java realm, therefore no external dependency is taken advantage of. We present our tools and evaluate this approach.

The rest of this paper is organized as follows. In Section 2, we present the different approaches related to type parametrization. After, we briefly present our existing solution in Section 3. In Section 4, we present our new approach for the instantiation of generics. Further options are presented in Section 5. In Section 6, we evaluate the performance of our method. In Section 7, we briefly present our further aims. Finally, this paper is concluded in Section 8.

2 Theoretical Background

2.1 About Java Generics

The constructs of generic programming paradigm were introduced in Java 1.5 in 2004 [2]. The main idea behind this step was the support of Java programmers with a tool to avoid duplications, and help to write type safe code which is abstract enough to fit as many situations as possible without any modifications [6]. To reach this, all generic code has to contain a type variable section named *parameter list* where the programmer can declare the usable types. This construct has a considerable merit. Since *Generics* applies dynamically typed, a class or function can be used with totally different types without recompile the code but keep type safety. To reach that, *Java* offers the runtime polymorphism, which means the following:
If `X` is a subtype of `T`, every occurance of `T` can be replaced with the objects of `X`. This makes the usage *Generics* versatile but it has its own cost [5].

Restrictions on the type parameters require bounded generic type parameters. If one defines an upper bound for a generic parameter, only its subclasses can be used as generic argument type.

Because of the *type erasure*, we have to deal with the some factors which can affect the performance:

- At compile time, all type parameters will be deleted and replaced with their first bound or `Object` if it is unbounded [15].
- To ensure type safety, *JVM* will generate type casts.

- To preserve polymorphism, bridge methods will be generated as well.

And exactly here is the weakness of the generics. The Java compiler has to modify the written code and insert runtime parts which can significally decrease the speed and effectiveness and may increase the heap memory consumption. Because of the dynamic typing, the construction must be as abstract as possible even if it is not necessary at all. As we showed, sometimes a quazi-statically typed solution can be faster. Moreover, subtype checking is quite difficult in Java [7].

Listing 1: Java generics example

```
1  public class Generic<T> {
2      private T element;
3
4      public Generic(T value) {
5          element=value;
6      }
7
8      public T get() {
9          return element;
10     }
11 }
```

2.2 Templates in C++

C++ provides templates for efficient type parametrization [4]. A template is a code snippet that is parametrized and the C++ compiler instantiates with different arguments at compilation time. Let us consider the following example that is quite similar to the previous one:

Listing 2: C++ template example

```
1  template <class T>
2  class Template {
3  private:
4      T element;
5
6  public:
7      Template( const T& t ): element( t ) { }
8
9      const T& get() const { return element; }
10 };
```

The compiler cannot instantiate the template and cannot generate corresponding low-level code unless the template argument is known, thus the template itself is not compilable code from the view of typical C++ compilers (e.g. g++, clang++).

The compiler generates specific code when the template is instantiated. For instance, in case of `Template<int>`, the compiler generates code from the template by substituted `T` with `int`. This construct enables to instantiate templates with arbitrary, previously unknown classes. The compiler is aware of the called functions related to the template parameter, so it can optimize many calls [9]. Moreover, type safety is an essential aspect of templates. On the other hand, code bloat of binary code may appear and template instantiation increases compilation time.

C++ provides function templates and class templates [13]. However, a template can be parametrized not only types but integral constant values, pointers, pointer-to-members, etc. If the compiler does know what the argument is, it is able to generate code. However, string literals are not supported to be template arguments [14].

C++ templates offer an interesting approach. In case of class templates, one can write special implementation for specific class parameters with partial and full specializations. Utilization of this possibility leads us to C++'s template metaprogramming feature that is a Turing-complete subset of C++ [11]. Metaprograms are beneficial since they can speed-up the execution time, enable the development of active libraries that make decisions at compilation time and evaluate compile-time asserts.

3 Previous Works

To create templates in Java, we investigated two different ways to instantiate a generic class. In our previous paper, we offered a new keyword and a different class structure to transform a generic class to a template one called *Java Template* [12]. This constructed originated from the C++ templates. It was quite useful and comfortable but already had limitations. In this case, we had to manually rewrite the existing code and declare the template variables. The work was getting more complex when we tried to transform some of the standard containers, and that was the inspiration to try a totally different way and instead of transforming code just create a tool which can work with pure *Generics* without any structural modification (later we will see that it is impossible because of the context-sensitive parts).

First of all we summarize the results of the *Java Templates* and after all we show how can we instantiate the generic classes directly.

3.1 Templates and Packages

The *Java Template* is a very similar class construct to C++ templates, but mixing the benefits of the *Generics*. At the beginning of the class definition, there is the `template` keyword followed by the identifiers of type parameters. To insure the instantiation, we restricted the following:

- If `T` is primitive type or literal of primitive types (String literals inclusive) there is no limitation.

- If `T` is an object type, we recommend to use the fully qualified name to avoid any importing issue.

At this point, we have some other requirements by Java that we have to meet. Every Java class has its own and unique identifier called *fully qualified name*. This is a composition of the name of the class and its place of the package hierarchy. The Java compiler does not allow the programmer to create two different classes in the same package with the same name. And well, if we use generics, we have to create only one class to many types, but with templates (since it is statically typed) every single type needs its own instance. To solve this issue, we decided that the package which contains the template instance will build up from the actual parameter types. For this, we made some restrictions:

- If `T` is a primitive type, or a literal of primitve types (String literals included) we add a prefix to it.
- If `T` is an object type, we use the fully qualified name, by escaping the separators with backslash.
- All package names must be able to generate with only of the previous two rules.

To avoid any OS specific issue with package names, the maximum length should not exceed 260 characters.

3.2 Instantiation

The *Java Templates* have a special way to instantiation since this construct is not the part of Java. For this, we developed a tool, which can tokenize the source and turn it into a standard C++ macro. Considering that the macro is a low-level language feature, to guarantee the success, need some restrictions:

- The number of declared type parameters at most the number of given parameters.
- For object types, the *fully qualified name* is preferred.
- A formal variable name can be replaced with a given literal if, and only if the specific variable name is only occurs at right hand side of an expression or where literals are allowed by Java.

4 New Results

Since the transformation and templatize not really decent when the class structure getting more complex, we had to have find another way. The new idea was that instead of creating a new language feature, and depend on external tools (especially `g++` compiler for the *Java Templates*) we shall use only what Java gives us. In our

new tool, we totally left the `g++` and other external dependecies and build up the preprocessing with pure Java. In this version, we introduced a *typetable* in the lexer tool, which acts as a field memory and store the formal and actual parameters in an associative container (list of key-value pairs). This makes the identification and replace lot easier and let us use this information in the future. The main steps now are the following:

- Load the source and localize the generic type declarations.
- Parse the type placeholders and store them in the *typetable*. To ensure correct run, we made a check to determine that the tool get enough arguments, because as we discussed, the number of actual types must be greater than or equal to the placeholders. In this step, the generic bounds are irrelevant because they will be replaced with an exact type, which will be an upper bound of the acceptable object at that point (this can be possible because of the Liskov substitution principle).
- Assemble the package of the class. This step is the first which clearly gains advantage by using the *typetable*. To create the package, we read the values one-by-one, and concatenate to the package name. To meet the rules of package name declared in Java, we have to replace the dot character to backslash when the current type is an object. For this step, we implement a minor feature as the support of generic type arguments. To reach this, we treat the characters of the *diamond operator* as dot, and also escape it with backslash.

With these three steps, we get the same result as with *Java Templates* which needed six steps for this. In the next sections, we show small examples to both methods and compare them.

4.1 Examples

First, we create a basic implementation of a `Stack` with the `Java Templates`. To give information to the lexer tool, we had to use the `template` keyword to declare the formal parameters. Since we are not restricted by the Java grammatic rules, we can use arrays typed by placeholders because the final type will be known at last before compile time.

Listing 3: Java template example

```
1 template(T)
2 class Stack {
3     private T[] elements = new T[10];
4     private int c = 0;
5     // ...
6     public void push(T item) { elements[c++] = item; }
7     public T pop() { return elements[--c]; }
8 }
```

Now, the same stack implemented with the toolkit of *Java Generics*. Since our improved tool can work with standardized generic classes, we do not need any special keyword or language feature, because we can extract all necessary information right from the source. However (as we discussed this problem in the next section) this is a special example. In this case, we have only one `Object` array (which the only possibility because the grammatic rules), but we could prepare our tool to handle this, since there are no ambiguous types. Note, if we want to declare another `Object` array, we may run into anomalies because all these will be replaced with a specific type which given as argument.

Listing 4: Java generic class example

```
1 class Stack<T> {
2     private Object[] elements = new Object[10];
3     private int c = 0;
4     // ...
5     public void push(T item) { elements[c++] = item; }
6     public T pop() { return (T)elements[--c]; }
7 }
```

Finally, both versions of our tool will produce the exactly same Java class. This class is now statically typed (as much as Java allows it), and as one may see in section 6, in some cases, the speed-up is considerable.

Listing 5: Generated Java class

```
1 class Stack {
2     private int[] elements = new int[10];
3     private int c = 0;
4     //...
5     public void push(int item) { elements[c++] = item; }
6     public int pop() { return elements[--c]; }
7 }
```

4.2 Compilation with our Tool

We demonstrate the precompiling process of the generic with our tool. A straightforward generic class is `Pair` that we use for presentation.

In this example, the two generic parameters are K and V which denote the key and value types. For instantiation, the tool requires the following arguments in order:

- Tha Java source file contains the implementation of `Pair` generic class.
- Restriction for this step that the file may contain only one top level class which has to match the name of type.

- The actual type of `K` which if it is an object type then it must be the fully qualified name.
- The actual type of `V` that has the same condition as `K`.

Listing 6: Java Pair generic class example

```
1  public class Pair<K, V> {
2      private K key;
3      private V value;
4
5      public K getKey() { return key; }
6      public V getValue() { return value; }
7
8      public void setKey(K _key) { key = _key; }
9      public void setValue(V _value) {
10         value = _value;
11     }
12 }
```

Now let the actual parameters are `int` and `boolean`, therefore we use the hereinafter command for the instantiation:
`$java Tool Pair.java int boolean`

After compilation, we have the instantiated Pair class in a unique path which ensures to avoid the ambiguous references. This class contains the following:

Listing 7: Generated class example

```
1  public class Pair {
2      private int key;
3      private boolean value;
4
5      public int getKey() { return key; }
6      public boolean getValue() { return value; }
7
8      public void setKey(int _key) { key = _key; }
9      public void setValue(boolean _value) {
10         value = _value;
11     }
12 }
```

This tool is now a standalone component of the building process right before the compilation. We are working on an improved integration of compilation task.

This tool at the moment has two major deficiencies. The first one is the requirement of explicit enumeration of all actual types, although if we want to replace for example the `T extends Number` parameter, it shall be guessed and replace `T` with `Number`. This feature is useful every time when the bound is a class/typename, so

giving command line arguments is necessary only when we want to use primitives or the bound is an interface. The second one is a more serious limitation, since only one class can be given to the tool but in enterprise environment there are hundreds of classes in every single project and instantiate one by one requires countless hours. So we want to introduce a JSON-like format which contains cases of every single preprocessing task. A task describes the name of the file which will be instantiated and the actual type parameters.

5 Improvements

Although the new methods are really beneficial, we find some questions which still waiting for to being aswered. One of them is the problem of the ambiguous field typing. Let us suppose that we created a generic container class that uses an `Object` array to store elements. Now we declare another `Object` array for reasons. If we use the new tool, the clear way is to replace the storage array's type from `Object` to the given one (let it be now `int`). But we also have the other array named `otherarray` which has a different (but not less important) functionality, and it must remain `Object`. In this case, we have the following possible solutions:

1. Replace `Object` with `int`: In this case, the functionality of `otherarray` will be damaged, since the role of the `Object` array is context-sensitive.

2. Train the tool to identify the possible fields: This way is significantly more complex because of the context-sensitive grammar rules and the chance of mistakes even more higher than the previous solution.

3. Pass the chosen identifiers as tool arguments: Now, we can decide in every case whether the current field is modifiable. Although from the point of the input this is one of the best choice, with the extra arguments can make the usage of the tool more complicated.

4. In Java, one can use annotations for the member declarations. Members can be distinguished by their annotations.

6 Measurements

In this section, we present the performance of our solution. We focus on the runtime performance because this property is more important than the duration of compilation time. The preprocessor has I/O-intensive tasks, so its performance depends on the storage device [3]. 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 instantiated generic version can be instantiated with `int`. This approach avoids the autoboxing between `int` and `Integer` and overhead of many memory allocations can be eliminated.

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 instantiated generic 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 [8].

The speed-up is significant, therefore we should realize what are the main reasons behind this effect, cache consistency or other JVM runtime optimazations. As future work, we evaluate the proposed approach with more generics and explore how the instantion can be utilized much more effectively.

7 Future Work

As we discussed earlier, the most difficult issue that waiting for solution is the problem of context-sensitive code parts. Of course not just the ambiguous fields, but in Java Standard especially, since the container classes take advantage of the toolkit of the *Generics*. First of all, we have to explore and classify the parts which can lead to anomalies or errors if we directly transform them. Comprehensive evalutaion is necessary, as well. In the future, we want to implement an integrated development environment (IDE) plugin which wraps our solutions and provides wide support to the users. All in all, our most ambitious goal is to become part of the Java language.

8 Conclusion

Type parametrization is an essential construct in modern programming languages with different backgrounds. For instance, C++ provides templates that are instan-

tiated by the compiler during compilation. Java offers generics that are based on type erasure. According to the measurements, the runtime performance can be significantly better when templates are in-use.

Previously, we created Java templates and a tool that instantiates them. A new syntax was offered that was not compatible with existing code bases. Therefore, our aim became the instantiation of Java generics: standard Java generics to instantiate with a new tool. In this paper, we introduced the background of our proof-of-concept tool and we have measured and evaluated the runtime efficiency of the proposed approach. The instantiated generic performs significantly better compared to the standard solution.

References

[1] Project Lombok. `https://projectlombok.org/`.

[2] Arnold, Ken, Gosling, James, and Holmes, David. *Java(TM) Programming Language, The (4th Edition)*. Addison-Wesley Professional, 2005.

[3] Babati, Bence, Pataki, Norbert, and Porkoláb, Zoltán. C/C++ preprocessing with modern data storage devices. In *Proceedings of the 13th IEEE International Scientific Conference on Informatics*, pages 36–40. IEEE, 2015. DOI: `10.1109/Informatics.2015.7377804`.

[4] Burrus, Nicolas, Duret-Lutz, Alexandre, Duret-Lutz, Re, Geraud, Thierry, Lesage, David, and Poss, Raphael. A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming. In *Proceedings of the Workshop on Multiple Paradigm with OO Languages (MPOOL)*, 2003.

[5] Dragan, Laurentiu and Watt, Stephen M. Performance analysis of generics in scientific computing. In *Proceedigs of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*, pages 93–100, 2005. DOI: `10.1109/SYNASC.2005.56`.

[6] Ghosh, Debasish. Generics in Java and C++: A comparative model. *ACM SIGPLAN Notes*, 39(5):40–47, 2004. DOI: `10.1145/997140.997144`.

[7] Grigore, Radu. Java generics are turing complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 73–85, New York, NY, USA, 2017. ACM. DOI: `10.1145/3009837.3009871`.

[8] Horváth, Gábor, Pataki, Norbert, and Balassi, Márton. Code generation in serializers and comparators of Apache Flink. In *Proceedings of the 12th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS'17, pages 5:1–5:6, New York, NY, USA, 2017. ACM. DOI: `10.1145/3098572.3098579`.

[9] Meyers, Scott. *Effective STL*. Addison-Wesley, 2001.

[10] Nam, Daye, Horvath, Amber, Macvean, Andrew, Myers, Brad, and Vasilescu, Bogdan. MARBLE: Mining for boilerplate code to identify API usability problems. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 615–627, 2019. DOI: 10.1109/ASE.2019.00063.

[11] Porkoláb, Zoltán. Functional programming with C++ template metaprograms. In Horváth, Zoltán, Plasmeijer, Rinus, and Zsók, Viktória, editors, *Central European Functional Programming School: Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*, pages 306–353, Berlin, Heidelberg, 2010. Springer. DOI: 10.1007/978-3-642-17685-2_9.

[12] Soha, Péter and Pataki, Norbert. Effective type parametrization in Java. *AIP Conference Proceedings*, 2116(1):350007, 2019. DOI: 10.1063/1.5114360.

[13] Stroustrup, Bjarne. *The C++ Programming Language (special edition)*. Addison-Wesley, 2000.

[14] Szűgyi, Zalán, Sinkovics, Ábel, Pataki, Norbert, and Porkoláb, Zoltán. *C++ Metastring Library and Its Applications*. In *Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*, pages 461–480. Springer, Berlin, Heidelberg, 2011. DOI: 10.1007/978-3-642-18023-1_15.

[15] Torgersen, Mads, Hansen, Christian Plesner, Ernst, Erik, von der Ahé, Peter, Bracha, Gilad, and Gafter, Neal. Adding wildcards to the Java programming language. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, pages 1289–1296, New York, NY, USA, 2004. ACM. DOI: 10.1145/967900.968162.