Generic instances in Java Byte Code

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

It is unlikely that there would be a runtime representation for each distinct function type, meaning developers would be further exposed to and limited by erasure. For example, it would not be possible (perhaps surprisingly) to overload methods `m(T->U)` and `m(X->Y)`.

Brian Goetz, [Goe13]

1 Introduction

The beginning citation from Brian Goetz, the main Java 8 developer, had been the motivation to have look on the realization of generics as type erasures in the Java Byte Code. In a first section will have a look on the type erasures, the main problems and the realization of code generation.
....

2 Type erasures

This section is devieded in two parts. First we will give some prohibited Java examples induced by type erasures. In the second part we present the generated byte code with type erasures.

2.1 Problems

Let the following class `AA` be given.

```
package Testfiles;

class AA<T> {
    T attr;

    void set(T attr) { this.attr = attr; }

    T get () { return attr; }

}
```

The class `AA` has a type parameter `T`, a getter and a setter method. Furthermote `AA` is defined in the package `Testfiles`.

instanceof The first example, where type parameters are prohibited is the `instanceof` operator. The following example would be incorrect.

```
class Main {
    void m () {
      AA<?> ac;
      Boolean b1 = ac instanceof AA<String>;
      Boolean b2 = ac instanceof AA<Character>;
    }
}
```

As `String`, respectively `Character` is erased in the byte code no differentiation would ne possible.

Generic Exceptions While the first example is more or less obvious the prohibition of generic exceptions is surprising. The following example is prohibited.

```
class MyException<T> extends Exception { }
```

The reason is not the declaration, but the use of the exceptions in the *try/catch*-clause.

```
try { }
catch ( MyException<Typ1> e ) { }
catch ( MyException<Typ2> e ) { }
```

In the catch statement no differentiation would be possible.

Generic overloading Let us close the prohibited examples by the motivating example the generic overloading. It is not possible, that the argument type(s) of a method declaration is differentiated only by the type parameters.

```
package Testfiles;

class GenericsTestReduced {
  void meth1(AA<BB> x) { System.out.println("BB"); }

  void meth1(AA<Character> x) { System.out.println("Character"); }
}
```

The problems arises by the method invokation, as after type erasure no differentiation between both method types is possible.

```
//Constant-pool
...
16| tag = CONSTANT_Utf8, length = 5, meth1
17| tag = CONSTANT_Utf8, length = 17, (LTestfiles/AA;)V
18| tag = CONSTANT_Utf8, length = 9, Signature
19| tag = CONSTANT_Utf8, length = 33, (LTestfiles/AA<LTestfiles/BB;>;)V
20| tag = CONSTANT_Utf8, length = 5, meth1
21| tag = CONSTANT_Utf8, length = 40, (LTestfiles/AA<Ljava/lang/Character;>;)V

...
//Methods
...
method[2] = {
  access_flags = 0
  name_index = 16  //meth1
  descriptor_index = 17  //(LTestfiles/AA;)V
  ...
  attribut[1] = {
    attribut_name_index = 18  //Signature
    attribut_length = 2
    descriptor_index = 19 //(LTestfiles/AA<LTestfiles/BB;>;)V }}

...
method[3] = {
  access_flags = 0
  name_index = 20  //meth1 (meth2)
  descriptor_index = 17  //(LTestfiles/AA;)V

  attribut[1] = {
    attribut_name_index = 18  //Signature
    attribut_length = 2
    descriptor_index = 21  //(LTestfiles/AA<Ljava/lang/Character;>;)V }}
```

Fig. 1. Byte code of `AA`

2.2 Byte-code realization

We close the by presenting the realization in the byte code. We give a cutout of the translation of the class `GenericsTestReduced` in Figure 1. Method 2 and method 3 represents the overloaded methods `meth1`. In both cases the `descriptor_index` is 17 (`(LTestfiles/AA;)V`), that stands for `void meth1(AA x)`. The parameters `BB` and `Character` are erased.
Nevertheless, the complete types with instaciated parameters are stored in the `descriptor_index`es of the corresponding attributes. This is done for the compiler type check, if an instance of `GenericsTestReduced` is used in a class and only the class file of `GenericsTestReduced` is available.

3 Introduction of generics into Java Byte Code

The storage of the complete type in the class files as descriptor_index of the corresponding attribute, induced the idea to use these constant-pool references.

3.1 First approach: Change the links

The first idea is only to change the link, which means to set the descriptor_index to the string in the constant-pool, where the type parameters are stored.

The following cutout show ths approach for the given example

```
//Constant-pool
...
16| tag = CONSTANT_Utf8, length = 5, meth1
17| tag = CONSTANT_Utf8, length = 17, (LTestfiles/AA;)V
18| tag = CONSTANT_Utf8, length = 9, Signature
19| tag = CONSTANT_Utf8, length = 33, (LTestfiles/AA<LTestfiles/BB;>;)V
20| tag = CONSTANT_Utf8, length = 5, meth1
21| tag = CONSTANT_Utf8, length = 40, (LTestfiles/AA<Ljava/lang/Character;>;)V

//Methods
...
method[2] = {
  access_flags = 0
  name_index = 16  //meth1
  descriptor_index = 19  //(LTestfiles/AA<LTestfiles/BB;>;)V
  ...
  attribut[1] = {
    attribut_name_index = 18  //Signature
    attribut_length = 2
    descriptor_index = 19  //(LTestfiles/AA<LTestfiles/BB;>;)V }}
```

If this class file is loaded by the JVM, the following error mesaage arises:

```
> java Testfiles.GenericsTestReduced
Exception in thread "main" java.lang.ClassFormatError:
  Method "meth1" in class Testfiles/GenericsTestReduced
  has illegal signature "(LTestfiles/AA<LTestfiles/BB;>;)V"
  at java.lang.ClassLoader.defineClass1(Native Method)
  at java.lang.ClassLoader.defineClass(ClassLoader.java:760)
  at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
  at java.net.URLClassLoader.defineClass(URLClassLoader.java:455)
  ...
  at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:495)
```

This means that the JVM has a checker, which checks that no type with parameters is used.

3.2 Second approach: New JVM Type syntax

As the original Syntax of generics are not allowed to use, we have changed the syntax by mapping. Basically parametrized type is changed to constant type. This done by eplacing the menaningful characters in a parametrized type other characters. The parameter start and end characters < is replaced by %. The path separator characters / is replaced by # and the type end characters ; is replaced by %. This means the JVM type

```
LTy<Lp11/.../p1m1/T1;Lp21/.../p2m2/T2;...;Lp1n/.../pnmn/Tn;>
```

becomes

```
LTy%Lp11#...#p1m1#T1%Lp21#...#p2m2#T2%...%Lp1n#...#pnmn#Tn%%
```

References

[Goe13] Brian Goetz. State of the lambda, September 2013.