Generics in Java – Part II

Venkat Subramaniam
venkats@agiledeveloper.com
http://www.agiledeveloper.com/download.aspx

Abstract

In Part-I we showed the benefits and usage of Generics in Java 5. In this part (Part-II), we discuss how it is implemented in Java, and we delve into a number of issues with it. In Part-III we will discuss the problems with mixing generic and non-generic code, and the issues with converting a non-generic legacy code to generics.

Unchecked Warning

The Java compiler will warn you if it can't verify type-safety. You would see this if you mix generic and non-generic code (which is not a good idea). Developing applications, while leaving these kinds of warnings unattended is a risk. *It is better to treat warnings as errors.*

Consider the following example:

```
public class Test
{
    public static void foo1(Collection c)
    {
    }

    public static void foo2(Collection<Integer> c)
    {
    }

    public static void main(String[] args)
    {
        Collection<Integer> coll = new ArrayList<Integer>();
        foo1(coll);

        ArrayList lst = new ArrayList();
        foo2(lst);
    }
}
```

You have a method `foo1` which accepts a traditional `Collection` as parameter. Method `foo2`, on the other hand, accepts a generics version of the `Collection`. You are sending an object of traditional `ArrayList` to method `foo2`. Since the `ArrayList` may contain objects of different types, within the `foo2` method, the compiler is not able to guarantee that the `Collection<Integer>` will contain only instances of `Integer`. The compiler in this case issues a warning as shown below:

```
Warning:  line (22) [unchecked] unchecked conversion
                    found   : java.util.ArrayList
                    required:
        java.util.Collection<java.lang.Integer>
```

While getting this warning is certainly better than not being alerted about the potential problem, it would have been better if it had been an error instead of a warning. Use the compilation flag `-Xlint` to make sure you do not overlook this warning.

There is another problem. In the `main` method, you are sending generics `Collection` of `Integer` to the method `foo1`. Even though the compiler does not complain about this, this is dangerous. What if within the `foo1` method you add objects of types other than `Integer` to the collection? This will break the type-safety.

You may be wondering how in the first place the compiler even allowed you to treat a generic type as traditional type. Simply put, the reason is, **there is no concept of generics at the byte code level**. I will delve into the details of this in the "Generics Implementation" section.

Restrictions

There are a number of restrictions when it comes to using generics. You are not allowed to create an array of generic collections. Any array of collection of wildcard is allowed, but is dangerous from the type-safety point of view. You can't create a generic of primitive type. For example, `ArrayList<int>` is not allowed. You are not allowed to create parameterized static fields within a generic class, or have static methods with parameterized types as parameters. For instance, consider the following:

```
class MyClass<T>
{
    private Collection<T> myCol1; // OK
    private static Collection<T> myCol2; // ERROR
}
```

Within generic class, you can't instantiate an object or an array of object of parameterized type. For instance, if you have a generic class `MyClass<T>`, within a method of that class you can't write:

```
new T();
```

or

```
new T[10];
```

You may throw an exception of generic type, however, in the catch block, you have to use a specific type instead of the generic.

You may inherit your class from another generic class; however, you can't inherit from a parametric type. For instance, while

```
class MyClass2<T> extends MyClass<T>
{
}
```

is OK,

```
class MyClass2<T> extends T
{
}
```

is not.

You are not allowed to inherit from two instantiations of the same generic type. For example, while

```
class MyList implements MyCollection<Integer>
{
    //...
}
```

is OK,

```
class MyList implements MyCollection<Integer>, MyCollection<Double>
{
    //...
}
```

is not.

What is the reason for these restrictions? These restrictions largely arise from the way generics are implemented. By understanding the mechanism used to implement generics in Java, you can see where these restrictions come from and why they exist.

Generics Implementation

Generics is a Java language level feature. One of the design goals of generics was to keep binary compatibility at the byte code level. By requiring no change to JVM, and maintaining the same format of the class files (byte code), you can easily mix generics code and non-generics code. However, this comes at a price. You may end up loosing what generics are intended to provide in the first place – type-safety.

Does it matter that generics are at the language level and not really at the byte code level? There are two reasons to be concerned. One, if this is only a language level feature, what would happen if and when other languages are expected to run on the JVM? If the other languages to run on JVM are dynamic languages (Groovy, Ruby, Python, …), then it may not be a big deal. However, if you attempt to run a strongly typed language on JVM, this may be an issue. Second, if this is simply a language level features (*one heck of a macro essentially*), then it would be possible to pass in correct types at runtime, using reflection, for instance.

Unfortunately, generics in Java does not provide adequate type-safety. It does not fully serve what it was created for.

Erasure

So, if generics is a language level feature, what happens when you compile your generics code? Your code is striped out of all parametric types and each reference to parametric type is replaced with a class (typically Object or something more specific). This process is given a fancy name – type erasure.

According to the documentation "The main advantage of this approach is that it provides total interoperability between generic code and legacy code that uses non-parameterized types (which are technically known as *raw* types). The main disadvantages are that parameter type information is not available at run time, and that automatically generated casts may fail when interoperating with ill-behaved legacy code. There is, however, a way to achieve guaranteed run-time type safety for generic collections even when interoperating with ill-behaved legacy code."

While this provides interoperability with generic and non-generic code, it unfortunately compromises type-safety. Let's look at the effect of erasure on your code.

Consider the example code:

```
class MyList<T>
{
    public T ref;
}
```

By running javap –c, you can look at what's in the byte code as shown below:

```
javap -c MyList
Compiled from "Test.java"
class com.agiledeveloper.MyList extends java.lang.Object{
public java.lang.Object ref;

com.agiledeveloper.MyList();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return
```

The type `T` of the `ref` member of the class has been *erased* to (replaced by) type `Object`.

Not all types are always erased to or replaced by `Object`. Take a look at this example:

```
class MyList<T extends Vehicle>
{
    public T ref;
}
```

In this case, the type T is replace by Vehicle as shown below:

```
javap -c MyList
Compiled from "Test.java"
```

```
class com.agiledeveloper.MyList extends java.lang.Object{
public com.agiledeveloper.Vehicle ref;

com.agiledeveloper.MyList();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return
```

Now consider the example:

```
class MyList<T extends Comparable>
{
    public T ref;
}
```

Here the type `T` is replace by `Comparable` interface.

Finally, if you use the multi-bound constraint, as in:

```
class MyList<T extends Vehicle & Comparable>
{
    public T ref;
}
```

then the type `T` is replaced by `Vehicle`. The first type in the multi-bound constraint is used as the type in erasure.

Effect of Erasure

Let's look at the effect of erasure on a code that uses a generic type. Consider the example:

```
ArrayList<Integer> lst  = new ArrayList<Integer>();
lst.add(new Integer(1));
Integer val = lst.get(0);
```

This is translated into:

```
ArrayList lst = new ArrayList();
lst.add(new Integer(1));
Integer val = (Integer) lst.get(0);
```

When you assign `lst.get(0)` to `val`, type casting is performed in the translated code. If you were to write the code without using generics, you would have done the same. Generics in Java, in this regards, simply acts as a syntax sugar.

Where are we?

We have discussed how Generics are treated in Java. We looked at the extent to which type-safety is provided. We will discuss some more issues related to generics in the next Part (Part III).

Conclusion

Generics in Java were created to provide type-safety. They are implemented only at the language level. The concept is not carried down to the byte code level. It was designed to provide compatibility with legacy code. As a result, generics lack what they were intended for – type-safety.

References

1. Generics in Java, Part-I at http://www.agiledeveloper.com/download.aspx (look at references in Part-I)