Generics in Java – Part III

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

Abstract

In Part-I and II we discussed the benefits and usage of Java Generics, and how it is implemented under the hood. In this Part-III, we conclude with discussions on issues with mixing generic and non-generic (raw-type) code, and the issues of converting a non-generic legacy code to generics.

Mixing Generic and non-generic code

Let's consider the following example:

```
import java.util.ArrayList;

public class Test
{
    public static void addElements(Collection list)
    {
        list.add(3);
        //list.add(1.2);
    }

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

        addElements(lst);

        //lst.add(3.2);

        int total = 0;
        for(int val : lst)
        {
            total += val;
        }

        System.out.println("Total is : " + total);

    }
}
```

In the above example, `lst` refers to an instance of generic `ArrayList`. I am passing that instance to the `addElements()` method. Within that method, I add `3` to the ArrayList. Back in the `main()` main, I iterate though the `ArrayList` extracting one integer value at a time from it, and total it. The output from the above program is:

```
Total is : 3
```

Now, in `main()`, if I uncomment the statement, `lst.add(3.2);`, I get a compilation error as shown below:

```
Error:  line (18) cannot find symbol method add(double)
```

On the other hand, if I leave that statement commented, but uncomment the statement `list.add(1.2);` in the method `addElements()`, I don't get any compilation errors. When I run the program, however, I get a runtime exception as shown below:

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.Double
      at Test.main(Test.java:21)
      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
      at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.ja
va:39)
      at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccesso
rImpl.java:25)
      at java.lang.reflect.Method.invoke(Method.java:585)
      at
com.intellij.rt.execution.application.AppMain.main(AppMain.java:78)
```

What went wrong? In `main()` I am assuming that the `ArrayList<Integer>` contains integer values. At runtime though, that assumption is proved wrong by the addition of the value `1.2` in the `addElements()` method.

You may agree that getting a compile time error is better than getting a runtime error. However, Generics don't fully provide the type-safety they were intended to provide. If we are going to get runtime exception, it would be better to get that within the `addElements()` method, where we are adding the value `1.2` to the `ArrayList`, instead of in the `main()` when we are trying to fetch the elements out of the list. This can be realized by using `Collections` class's `checkedList()` method as shown below:

```
        //addElements(lst);
       addElements(Collections.checkedList(lst, Integer.class));
```

The `checkedList()` method wraps the given `ArrayList` in an object that will ascertain that the elements added through it are of the specified type, in this case, `Integer` type.

When I execute this program, I get the following runtime exception:

```
Exception in thread "main" java.lang.ClassCastException: Attempt to
insert class java.lang.Double element into collection with element type
class java.lang.Integer
      at
java.util.Collections$CheckedCollection.typeCheck(Collections.java:2206
)
```

```
      at
java.util.Collections$CheckedCollection.add(Collections.java:2240)
      at Test.addElements(Test.java:11)
      at Test.main(Test.java:19)
      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
      at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.ja
va:39)
      at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccesso
rImpl.java:25)
      at java.lang.reflect.Method.invoke(Method.java:585)
      at
com.intellij.rt.execution.application.AppMain.main(AppMain.java:78)
```

Compare this exception message with the previous one. The exception is reported in this case in line number 11 within the `addElements()` method instead of the previously reported line number 21 within the `main()` method.

If you have to pass generic types to methods that accept non-generic types, consider wrapping the objects as shown in the above example.

Converting non-generic code to generics

In Part –II we discussed the type erasure technique and saw how the parameterized types are converted to `Object` type or one of the types specified in the bound. If we have to convert from non-generic type to generic type, is it simply the question of adding the parameterized type `E` or replacing `Object` with `E`? Unfortunately, life's not that simple.

Consider the following example:

```
import java.util.ArrayList;
import java.util.Collection;

public class MyList
{
    private ArrayList list = new ArrayList();

    public void add(Object anObject)
    {
        list.add(anObject);
    }

    public boolean contains(Object anObject)
    {
        if (list.contains(anObject))
            return true;
        return false;
    }

    public boolean containsAny(Collection objects)
    {
        for(Object anObject : objects)
        {
```

```
            if (contains(anObject))
                return true;
        }

        return false;
    }

    public void addMany(Collection objects)
    {
        for(Object anObject : objects)
        {
            add(anObject);
        }
    }

    public void copyTo(MyList destination)
    {
        for(Object anObject : list)
        {
            destination.list.add(anObject);
        }
    }
}
```

`MyList` is a class that represents my own collection. Let's not get too technical about whether the `addMany()` method or the `containsAny()` method should actually belong to the class `MyList`. From the design point of view, if you think these should not belong here, they may belong elsewhere – in a façade – and the problems we will discuss will then extend to that class. Now, let's look at a sample code that uses this class:

```
class Animal {}
class Dog extends Animal { }
class Cat extends Animal { }

public class Test
{
    public static void main(String[] args)
    {
        MyList lst = new MyList();

        Dog snow = new Dog();
        lst.add(snow);

        System.out.println("Does list contain my snow? "
                  + lst.contains(snow));

        Cat tom = new Cat();
        lst.add(tom);

        System.out.println("Does list contain tom? "
                  + lst.contains(tom));
    }
}
```

The above program produces the desired result as shown below:

```
Does list contain my snow? true
Does list contain tom? true
```

Now, let's set out the change the `MyList` to use generics. The simplest solution – modify `Object` with parameterized type `E`. Here is the result of that code change:

```
import java.util.ArrayList;
import java.util.Collection;

public class MyList<E>
{
    private ArrayList<E> list = new ArrayList<E>();

    public void add(E anObject)
    {
        list.add(anObject);
    }

    public boolean contains(E anObject)
    {
        if (list.contains(anObject))
            return true;
        return false;
    }

    public boolean containsAny(Collection<E> objects)
    {
        for(E anObject : objects)
        {
            if (contains(anObject))
                return true;
        }

        return false;
    }

    public void addMany(Collection<E> objects)
    {
        for(E anObject : objects)
        {
            add(anObject);
        }
    }

    public void copyTo(MyList<E> destination)
    {
        for(E anObject : list)
        {
            destination.list.add(anObject);
        }
    }
}
```

We modify the `main()` method to use the generic type (actually no change is needed to `main()` if you will continue to use the non-generic style).

The only statement modified is shown below:

```
MyList<Animal> lst = new MyList<Animal>();
```

The program compiles with no error and produces the same result as before. So, the conversion from raw-type to generics went very well right? *Let's ship it?*

Well, this hits right on the head with the issue of testing the code. Without good test, we would end up shipping this code, only to get calls from clients who write code like the following:

```
Dog rover = new Dog();
ArrayList<Dog> dogs = new ArrayList<Dog>();
dogs.add(snow);
dogs.add(rover);
System.out.println(
    "Does list contain snow or rover? " +
        lst.containsAny(dogs));
```

We get a compilation error:

```
Error:  line (29)
containsAny(java.util.Collection<Animal>)
in MyList<Animal> cannot be applied to java.util.ArrayList<Dog>)
```

What's the fix? We need to tweak the `containsAny()` method a little to accommodate this reasonable call. The change is shown below:

```
public boolean containsAny(Collection<? extends E> objects)
```

(Refer to Part I and II of this article for details about lower-bounds, upper-bounds, and wildcard).

Now, the program works fine again. However, if the `main()` is modified as follows, we get a compilation error yet again:

```
lst.addMany(dogs);
```

Once again, this requires tweaking the code, this time the `addMany()` method.

```
public void addMany(Collection<? extends E> objects)
```

Now, let's take a look at the `copyTo()` method. Here is an example to use this method:

```
MyList<Dog> myDogs = new MyList<Dog>();
```

```
myDogs.add(new Dog());

myDogs.copyTo(new MyList<Dog>());
```

In the above code, we are copying Dogs from one MyList<Dog> to another MyList<Dog>. Seems reasonable? Yep and it works. It is also legitimate to copy Dogs from a MyList<Dog> to a MyList<Animal> isn't it? After all, a list of Animals can hold Dogs. So, let's give that a shot:

```
MyList<Dog> myDogs = new MyList<Dog>();
myDogs.add(new Dog());

myDogs.copyTo(new MyList<Animal>());
```

This code, however, results in a compilation error as shown below:

```
    Error:   line (36) copyTo(MyList<Dog>) in MyList<Dog> cannot be
applied to (MyList<Animal>)
```

In this case, however, we do want a collection of base to be sent to this method. We have to tweak again, this time the copyTo() method. We want this method to accept a MyList of Dogs or MyList of Dogs base class. In general terms, we want it to accept MyList of the parameterized type or MyList of the parameterized type's base type. So, here is the code for that:

```
public void copyTo(MyList<? super E> destination)
```

Depending on the situation, you may have to use the parameterized type E, a lower-bound, an upper-bound or a wildcard. Unfortunately, this requires quite some thinking. You may easily miss out on these details (like we saw in the above example) and the problem may not surface until someone actually writes a piece of code that exercises your code in a way so as to bring out the problem.

Conclusion

To summarized, Generics were developed to provide type-safety. They accomplish that goal to a certain degree. However, they don't provide type-safety to the full extent. This is largely due to the design goals and implementation constraints. First learn Generics in Java. Then have the wisdom to decide when and how (much) to use it.

References

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