RECURSION SOFTWARE, Inc.

Generic Programming, Java Generics, and JGL 5.0

By James C. Bender

Recursion Software, Inc.

March 23,2005

TABLE OF CONTENTS

Overview 1
JGL, STL, and JDK 5.0 1
Some History 1
Java Generics 2
Generic Arithmetic 3
More Functions, Predicates, Iterators, and Algorithms 3
Generators 3
Constraints 5
Conclusion 6
About the author 8

Generic Programming, Java Generics, and JGL 5.0

By James C. Bender

Overview

JGL Toolkit 5.0 brings JDK 5.0 features to generic programming in the spirit of the C++ Standard Template Library (STL). The JGL Toolkit has been brought up to date with generic type parameters and an expanded component set and functionality over previous releases. JGL started out as being very much about collections. With the advent of the Java Collections framework, JGL's unique approach to generic programming with specialized iterators, containers, functions, and predicates have endeared themselves to large numbers of programmers. Now, JGL has been updated with generic type parameters to classes and static methods.

JGL, STL, and JDK 5.0

JGL, the Generic Library for Java, is very much in the STL tradition. Recently, JGL was updated to use generics and other features from JDK 5.0. The basic theme, though, is still about generic programming. Alexander Stepanov made generic programming a conversation topic, and more, although the idea had been around since Ada and Common Lisp. The C++ Standard Template Library is the most visible implementation of generic programming ideas that has gained wide use.

There is ongoing research and experimentation going on in the background. David Musser, at Rensselaer, was an earlier collaborator with Stepanov who has gone off in a new direction with generic programming. He is essentially mechanizing patterns. This is in the academic world, however, and hasn't yet impacted the real world generic programming.

The STL approach has gained considerable acceptance in the industry and seems to be here to stay. Now, with JGL 5.0, there are the additional benefits that come with generic parameterization and a more extensive set of functions, predicates, algorithms, and complementary new features such as generic arithmetic for Number classes, generators, and a constraint network.

Some History

About 1996, when Java was very new and lacked good collection and container support, Graham Glass, then CTO of ObjectSpace, decided to not only implement a library that provided a collections framework but also to implement a facility for generic programming based on what was in STL. Graham had been involved with C++ and STL and was interested in the concept. He was also out to gain some "mind share" with a product that people would want to use. The official product name was changed to "Generic Library for Java.", although the JGL initials suggest the original.

For a long time, many people thought of JGL as just being about collections, but when the JDK finally implemented an acceptable collections framework, that part of JGL became largely obsolete. JGL 4.0 eliminated most of the JGL collections and replaced them with the JDK collections framework. The real value of JGL 4.0 was the generic programming facilities, not the collections. Collections may be why STL is so popular in the C++ world, but JGL has always provided Stepanov's generic programming paradigm to the Java world, and continues to do so.

After JGL embraced the JDK collections framework, JGL was left to concentrate on the various components of "Generic Programming," such as functions, predicates, and collections-related algorithms. These are features that the JDK Collections framework largely ignores. With the advent of JDK 5.0 (a.k.a. JDK 1.5.0), JGL has been parameterized to obtain the advantage of Java generics. Additional features added to JGL included generators, in addition to the simple constraint framework. Among the generators is a Markov chain generator, functionality that is finding wider application, including in Bayesian spam filters.

Java Generics

In the process of parameterization and adding generic arithmetic, many lessons were learned. One was the impact of generics on mathematical operations in Java. Generic arithmetic did one major service, which was the elimination of most requirements to do runtime type identification (through "`instanceof`"). Generic arithmetic provides the capacity to operate on Number subclasses, rather than having to explicitly convert to primitive types constantly (even if that is what happens under the wrappers). We can also enjoy the benefits of autoboxing and unboxing, although you may be better served by explicitly doing that yourself, due to efficiency issues.

The parameterization process is very educational, as the compiler enforces relationships between variables and parameters that are not always obvious to anyone other than the original author. The other thing that happens is that assumptions are made about types. These type assumptions are manifested in the methods that are invoked in legacy code that need to be visible. Having the generic parameter extend the type that implements the needed methods provides this visibility.

The greatest benefit to generic parameterization is that the type references are much more consistent than pre-JDK 5.0 code with all the casts from `Collections` and `HashMaps`. Retrofitting existing code with generic parameterization exposes the type assumptions made in the original implementation. The challenge is to parameterize in such a way as to retain the original intent, while achieving a logically defensible result. All too often, there were attempts to achieve polymorphism on the insides, using runtime type identification to reach resolution. That approach was problematic, as generics would naturally be used to provide the type information. In some cases, to reduce the risk that existing application code would not compile, the runtime type identification was preserved.

As mentioned, parameterizing the JGL code exposed many underlying assumptions, especially regarding types. That may well be a common experience among programmers retrofitting Java generics in legacy code. Often, the generic parameters reveal a need for them to be subtypes of classes like Number (in the case where arithmetic is to be performed). Actually, the compiler does the revealing when otherwise good code develops compile errors during the process. In JGL, the parameters often needed to extend collection classes and iterators, not just Number.

One interesting fact that may not be obvious is that Class is parameterized, and can have a generic instantiation, as well, that forces consistency between a Class parameter and the associated generic parameters. Generic arithmetic employs this feature for the option where a Class is passed to select the correct generic arithmetic sole instance.

Generic Arithmetic

While David Hall is using his version of generic arithmetic in JGA for implementing a spreadsheet Swing component, JGL is using its own release for implementing statistics, constraints, and a new group of science and engineering-oriented mathematical functions. Generic arithmetic is parameterized so that the appropriate `Number` subclass can be used to fit the application. Where accurate decimal calculations are required, we may well be reduced to using `BigDecimal`, despite the efficiency costs. There seems to be no viable alternative, without enduring the endless travails of binary arithmetic and the odd results when applied to decimal arithmetic. Otherwise, we are left to deal with the mess that results from doing decimal computation with binary numbers.

One place where this was manifested was when generic arithmetic was tested with the constraint framework. The corresponding Junit tests had to be carefully rounded, simply to allow for reasonable testing, since `BigDecimal` is only one of the Number subclasses that were tested. As you would expect, generic arithmetic applied to `BigDecimal` is using the correct underlying API to do accurate computation.

This whole computational accuracy issue is receiving more attention, and the availability of Java generics and the accompanying generic arithmetic found in JGL should be very beneficial. `BigDecimal` seems like a heavy hammer to use for computation, but seems to be the easiest path to accuracy in Java. Conceivably, a programmer could implement his own large integer, programmer-scaled arithmetic, and possibly it might be more efficient, but the safest route would seem to be to stick with `BigDecimal`, when the rounding errors from binary arithmetic can't be tolerated.

A nice side effect of how JGL implements generic arithmetic is that there is only one instance of each "arithmetic box" for a given Number subclass. There are not a multitude of instances. This is possible because there is nothing preserved in the instance between method invocations.

More Functions, Predicates, Iterators, and Algorithms

JGL has long been equipped with many functions and predicates for use in generic programming of the STL sort. JGL 5.0 has primarily extended the function set, with only a smaller extension to the predicate set. That is largely a consequence of the extensive predicate list already implemented and the relatively shorter function list in the previous JGL releases. The likeliest candidates for adding to the functions were mathematical in nature, simply because the obvious exponential, trigonometric, and hyperbolic functions either are part of the `java.lang.Math` class, or are easily implemented from methods on that class. There are also good descriptions of how the functions are defined, so that when we implemented a `CothNumber<T extends Number> function` class, there was an accepted definition to use that was a simple extension of what is provided by the `java.lang.Math` class.

Given the existence of generic arithmetic, then providing a Statistics class seemed like another obvious step. Again, the computations are very well-known and well-documented, so a class with parameterized static methods could be provided that would take advantage of type-secure computation with Number subclasses stored in Collections instantiated for the chosen Number subclass. A new-age (JDK 5.0) static method that has been parameterized for generics infers types from the method arguments. That implies that the arguments all need to be instantiated with Number subclasses. That will provide the types needed so that internally, generic arithmetic can select the correct algorithms for the types used.

Generators

Generators are functions that have persistent state between invocations and produce an output in a sequence. Generators seem not that different from Iterators, in the sense that they respond to the `next()` method and return a value. Sequence generators are a well-known hardware artifice, but are less seen in common business-oriented Java code. One application for generators is for producing Markov chains. This

seems to be a topic that has generated a good deal of interest. The concept originated from the dawn of communications theory, and has recently been used in a variety of applications, including Bayesian spam filters. JGL now has a basic framework for implementing Markov chains. What is provided is a means for building networks of nodes with transitions, which produce a sequence of values that are selected by probabilistic traversal of the network. Figure 1 illustrates the example network, where the boxes are nodes and the lines represent transitions.

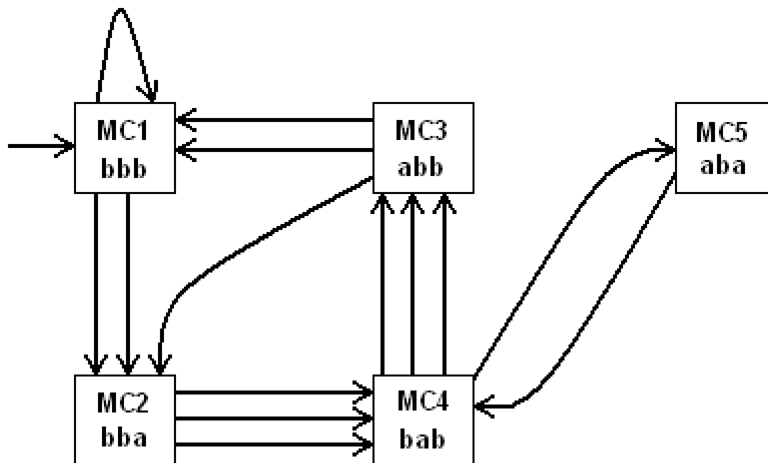

Figure   Nodes and Transitions for Markov Chain Generator

The following is the code to implement the network shown in Figure 1:

```
public void testMarkovChain(){
        MarkovChain<String> mc1 = new MarkovChain<String>();
        mc1.setValue("bbb");
        MarkovChain<String> mc2 = new MarkovChain<String>();
        mc2.setValue("bba");
        MarkovChain<String> mc3 = new MarkovChain<String>();
        mc3.setValue("abb");
        MarkovChain<String> mc4 = new MarkovChain<String>();
        mc4.setValue("bab");
        MarkovChain<String> mc5 = new MarkovChain<String>();
        mc5.setValue("aba");

        mc1.addTransition(mc1);
        mc1.addTransition(mc2);
        mc1.addTransition(mc2);

        mc2.addTransition(mc4);
        mc2.addTransition(mc4);
        mc2.addTransition(mc4);

        mc5.addTransition(mc4);

        mc4.addTransition(mc5);
        mc4.addTransition(mc3);
        mc4.addTransition(mc3);
        mc4.addTransition(mc3);

        mc3.addTransition(mc1);
```

```
            mc3.addTransition(mc1);
            mc3.addTransition(mc2);

            MarkovChainGenerator<String> mcg =
new MarkovChainGenerator<String>(mc1);

            String value = null;
            Set<String> set = new HashSet<String>();

            for (int i=0; i<50; i++){
                  value = mcg.next();
                  set.add(value);
                  System.out.println(value);
            }
            System.out.println("number of unique values"+set.size());
            assertTrue(set.size()>1);
      }
```

When the generator is run, this is an example of the output that is produced:

```
      bbb bba bab aba bab abb bbb bba bab abb bba bab aba …
```

Constraints

As previously mentioned, generic arithmetic is also used in the constraint examples provided with JGL 5.0. The basic constraint network is independent of numeric computation, and has been used in symbolic reasoning systems in the past. Generic arithmetic is used here, as there are simple examples employing constraints for doing implicit arithmetic with equations. Constraint nodes and connectors are used to “wire” an equation. Values can be inserted and results computed. Instead of the one-way computation with equations that is usual with Java, equations can be driven multiple ways. The toy example provided in the test code is for the Fahrenheit to Centigrade (Celsius) conversion equation. With that constraint network, you can insert a Fahrenheit temperature and obtain the Centigrade equivalent, or do the reverse by inserting a Centigrade temperature and obtaining the Fahrenheit conversion. Figure 2 illustrates how such a constraint network would be “wired up” (the connectors are shown as lines).

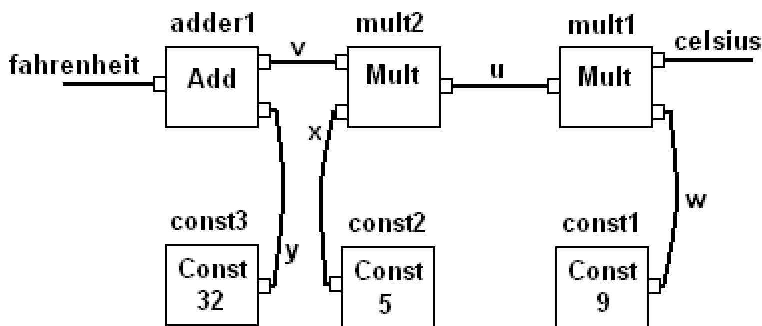

Figure 1 Fahrenheit to Celsius Conversion Constraint Network

The following is an example of how a constraint network is assembled, using the JGL constraint framework:

```
      public CelsiusFahrenheitConverter(Class<T> number,
                                        Connector<T,T> celsius,
                                        Connector<T,T> fahrenheit)
                        throws ConstraintException,
```

```
                                        InstantiationException,
                                        IllegalAccessException,
                                        ClassNotFoundException {
        super();
        c = celsius;
        f = fahrenheit;
        u = new Connector<T,T>("u");
        v = new Connector<T,T>("v");
        w = new Connector<T,T>("w");
        x = new Connector<T,T>("x");
        y = new Connector<T,T>("y");
        mult1 = new Multiplier<T>("mult1",number,c,w,u);
        mult2 = new Multiplier<T>("mult2",number,v,x,u);
        adder1 = new Adder<T>("adder1",number,v,y,f);
        GenArithI<T> arithBox = ArithmeticUtil.arithType(number);
        const1 =
        new ConstConstraint<T>(number,(T)arithBox.toValue(9.0),w);
        const2 =
        new ConstConstraint<T>(number,(T)arithBox.toValue(5.0),x);
        const3 =
        new ConstConstraint<T>(number,
                    (T)arithBox.toValue(32.0),y);
}
```

The code that uses this component is quite simple, and illustrates the multi-way computation that is possible (the code also illustrates accessing the generic arithmetic facility):

```
public void testConstraintsFahrenheit32BigDecimal()
                                            throws Exception{
        Constraint<BigDecimal,BigDecimal> user =
                                          new User<BigDecimal>();
        Connector<BigDecimal,BigDecimal> c =
                      new Connector<BigDecimal,BigDecimal>("c");
        Connector<BigDecimal,BigDecimal> f =
                      new Connector<BigDecimal,BigDecimal>("f");
        CelsiusFahrenheitConverter<BigDecimal> converter =
              new CelsiusFahrenheitConverter<BigDecimal>
                                       (BigDecimal.class,c,f);
        GenArithI<BigDecimal> arithBox =
              ArithmeticUtil.arithType(BigDecimal.class);
        c.setValue(arithBox.toValue(0),user);
        assertEquals("Correct value calculated for a Celsius of 0",
                                 arithBox.toValue(32),f.getValue());

}
```

Using constraints is a concept that has generated a considerable research effort since the seminal PhD thesis from MIT in 1980. That was actually preceded by work with Smalltalk in the latter 1970's, and was probably a factor in the MIT work. One method for resolving constraints is local propagation, which is what is used in the framework provided with JGL.

Conclusion

JGL 5.0 continues to be the JGL we all know and love. We just took advantage of what JDK 5.0 has to offer, and as a result, code that makes generic instantiations of JGL components will be better as a consequence. No longer do we have to hope that we have correctly cast what comes from Collections, but we have some degree of confidence, thanks to the compiler, that we have consistency between Collections and what we believe to be their contents. This holds the promise of reducing runtime ClassCastExceptions.

Once JGL was parameterized for generics, one obvious feature to add was generic arithmetic. JGA had this feature, although for a specialized role for use in a Swing spreadsheet component. JGL generic arithmetic is

targeted for all applications that would use Number subclasses. Not having to move between Numbers and primitive numbers seems appealing, and generic arithmetic offers that promise.

Once we had generic arithmetic, some obvious applications were for mathematically oriented functions and for new algorithms. As we noted, JGL now has new functions for use in engineering and science applications. Another application was in the Statistics algorithm class, which provides some typical statistical algorithms as generic-parameterized static methods. Generic arithmetic was also used in the constraint framework, more as a demonstration of what is possible with this technology, than for immediate application.

Generators are an obvious feature to add, partly because other STL-like Java libraries have them, and partly just because they are common in environments such as Smalltalk and Scheme. They are easily implemented, and this is a convenient way to show a simple implementation of Markov chains, in the form of a generator and a helper class.

JGL Toolkit 5.0 brings the JDK features to JGL, supports more reliable code, and demonstrates new features that are readily implemented in the new environment. Generic parameterization, alone, is a great boon, and will result in more reliable code at runtime, for those applications that use generics with JGL in the JDK 5.0 environment.

About the author

James C. Bender is a software engineer at Recursion Software, Inc. He may be contacted by email at engineer@recursionsw.com.

Copyright © 2005 Recursion Software, Inc. All rights reserved. Recursion Software, its logo and JGL Toolkit are trademarks or registered trademarks of Recursion Software, Inc. in the United States and other countries. All other names and trademarks are the property of their respective owners.

Recursion Software, Inc.
2591 North Dallas Parkway
Suite 200
Frisco, Texas 75034
1.800.727.8674 or 972.731.8800
www.recursionsw.com

RECURSION SOFTWARE, Inc.