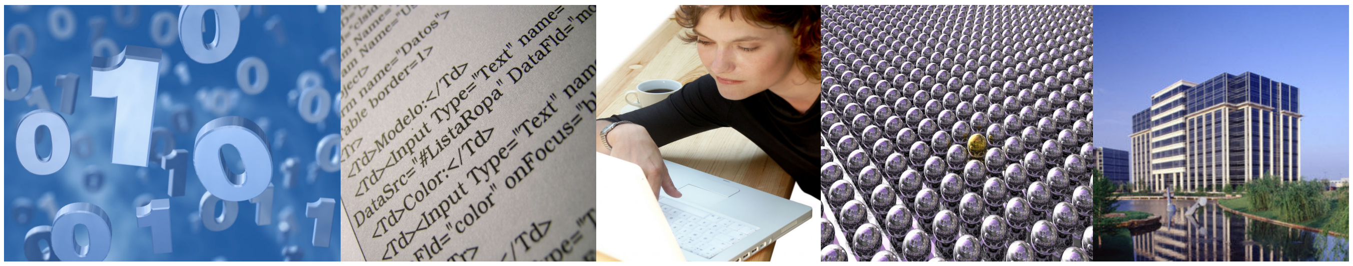
RECURSION SOFTWARE, Inc.

JGL® toolkit

Generic Programming with JGL 4

By John Lammers

Recursion Software, Inc.
July 2004



RECURSION SOFTWARE, Inc.

TABLE OF CONTENTS

Generic Programming with JGL 4

By John Lammers

Abstract

This article provides a brief introduction to generic programming concepts and discusses how JGL®, the Generic Library for Java™, from Recursion Software, Inc. supports generic programming.

The intended audience for this article is the Java developer with moderate familiarity with Java 2 Collections API concepts, such as collections and iterators.

Introduction to Generic Programming

Generic programming is a technique in which reusable algorithms or operations are represented as first class entities, completely independent of the data or objects upon which they operate. The most familiar examples of generic programming are those dealing with collections; that is, the application of an algorithm to the elements in a collection.

Let's take, for example, a sorting algorithm. A simple design results in a sort method on each container type. The appeal of this is that it yields a pretty intuitive syntax: `myContainer.sort().` The downside, though, is that this approach results in a lot of nearly-identical code in each container implementation. Another issue, and one that's more difficult to deal with, is that as we add more algorithms or containers, our containers' interfaces start to bloat. Worse, development schedules being what they are, we may not be able to update all the containers every time we produce a new algorithm, or we may add containers but not have time to implement support for each algorithm. Pretty soon, we have a set of containers with bloated interfaces, each supporting some subset of the algorithms we've written.

Generic programming solves this problem by separating the algorithms from the objects they act upon. The algorithms are separate pieces of code, written in such a way as to allow them to be applied in the same manner to any class that conforms to the minimal requirements of the algorithm.

So, what are these "minimal requirements"? These are methods or data members that the algorithm requires the objects on which it acts to support. In theory, these could be anything[1]; in practice, most algorithms dealing with containers rely on the container's ability to produce an iterator, and upon that iterator's support of certain

[1] Indeed, many of the most interesting applications of generic programming techniques do not involve containers at all. Interested readers are directed to Andrei Alexandrescu's excellent book on generic programming, "Modern C++ Design: Generic Programming and Design Patterns Applied." This work is very much oriented to C++ and templates, and for readers unfamiliar with those concepts, it may present quite a challenge. However, the core ideas and design principles expressed in the book are quite relevant to Java programmers interested in applying generic programming concepts.

methods; for instance, `hasNext()` and `next()`. In Java, all classes implementing `java.util.`Collection are capable of returning an iterator[2], as are all the STL containers in C++.[3]

How does Java support generic programming?

Often, C++ programmers are more familiar with generic programming concepts because of the STL (Standard Template Library), a collection of containers, iterators, and algorithms based on parameterized types, or templates. While the Java language does not include support for templates, this does not mean that Java programmers cannot use generic programming techniques or that they are more limited in how they can apply generic programming. From a generic programming standpoint, a Java algorithm operating on an object whose class implements a specific interface is functionally identical to a C++ algorithm operating on an object whose class is specified by a type parameter. There are a number of non-functional differences between generic programming via parameterized types and generic programming via interfaces. However, a discussion of these differences is beyond the scope of this article. For the purposes of this article, it is sufficient to recognize that we can apply generic programming techniques in Java through classes that operate on objects that support a common interface.

The cost and the value of generic programming

Let's return to our sorting algorithm. In our initial design, we had a `sort()` method on each container. Refactoring our design to use a generic programming approach, we separate the algorithm from the container. To do this, we create a new class, `Sorting`, which has a single method, `sort( List list )`. This method can sort the elements of any container that implements `java.util.List`, because it relies only on the `List` interface, rather than any details of the underlying `List` implementation.

Viewed as an isolated example, this may not seem like much of an advantage. However, when *all* of your algorithms work this way, you have a rich toolset that can be applied to many classes in a consistent, easily recognized manner. In addition, simply grouping algorithms into separate classes like this helps the developer of new algorithms know where to put them, and helps algorithm users know where to find them.

It's true that generic programming is a little unfamiliar to many Java developers, a little outside their "comfort zone". This is indicative of a very real cost associated with generic programming. If other developers don't understand a particular piece of code, they will not be as efficient when working with that code.

There are a couple of factors that offset this concern. Practices such as code reviews and pair programming help to spread knowledge about techniques, and thus help to ensure that we aren't going to be producing code that no one else is comfortable with. Additionally, one can make the argument that once developers become accustomed to generic programming techniques, the code that employs these techniques is actually more readable than other code. These considerations notwithstanding, if your development team is like many development teams, there will be some learning involved in becoming comfortable with generic programming techniques and deciding where and how to apply them. We should insist that generic programming add enough value to offset these costs. There are several ways in which generic programming demonstrates its worth. These are outlined below.

Generic programming supports separation of concerns

Separation of concerns, or the addressing of orthogonal problems independently, is critical for large-scale development. Generic programming, by its very nature, supports separation of concerns by encapsulating operations that can be applied generically to any object supporting a minimal interface. Benefits of separating

[2] JGL supplies a set of iterators that are more flexible and more expressive than the standard iterator classes in the JDK. While a developer does not have to know anything about JGL iterators to use the algorithms described here, and the topic of JGL iterators is beyond the scope of this document, it is worth pointing out that the JGL iterators remain a powerful and important part of the library and are available for use with JGL container classes as well as JDK collection classes.
[3] As you might expect, C++ iterators look a bit different from Java iterators, but the idea is the same--abstract the act of iterating over the elements of a collection from the implementation details of that collection.

concerns include reduced cognitive overhead, improved testability, and greater potential for parallel efforts. Generic programming reduces cognitive overhead because one is able to work with small, focused components with minimal and explicit requirements of the environment in which they are used. Testability is improved because the modular nature of the components produced by generic programming techniques lend themselves to independent testing. Finally, because the products of generic programming are independent units and have well-defined dependencies, it's easier to divide work across a number of developers to allow work to proceed in parallel.

Generic programming supports reuse

Because generic programming separates logic from the objects to it is applied, the code produced tends to be easier to reuse than other code. Generic code is designed, from the start, to be used on objects of a variety of types and to impose as few restrictions as possible on those types. While these are good practices in any style of development, generic programming pushes them to the forefront, and as a result, the code produced is often easier to reuse than traditional code.

Generic programming promotes good analysis and design

Writing generic code is, at first, not as straightforward as writing non-generic code. The developer writing generic code is trying to solve a problem in a way that will allow the solution to be applied in more than one context. This requires the developer to identify and abstract design elements that are (or should be) orthogonal to the specific solution being coded. For instance, contrast a sort algorithm written to operate on a linked list with a generic sort algorithm that accomplishes its work via iterators. In the solution developed specifically for a linked list, the algorithm is likely to contain references to concepts like a *current node*, a *next node*, etc. The design decision of whether to make the list singly-linked or doubly-linked may also show up in the algorithm. What has happened, is that design elements and design decisions from one concept, a linked list, have gotten mixed in with the implementation of a sort algorithm. Not only is the sort algorithm now harder to reuse, it is also harder to understand and maintain. To understand the algorithm now requires knowledge of the internals of the linked list for which it was written. A change in the algorithm has the potential to break other parts of the linked list implementation. Likewise a change in the linked list implementation could cause the algorithm to fail. The generic solution separates the sorting and linked list concepts, leading to a sorting algorithm that deals only with sorting--not with linked list concepts. This makes it easier to understand and maintain.

Generic programming enhances readability

Generic programming allows us to clearly and concisely express the intent behind a piece of code. An example of this is the replacement of looping constructs with generic algorithms. Take, for example, the task of clearing items from a cache if they have not been used recently. We'll look at two approaches to this task, one traditional and one using a generic algorithm. In both cases, we'll assume that we have access to a variable, `cache`, that is a `Vector` of `CacheEntry` objects that support a `getLastAccessed` method. We will further assume that we have a threshold variable of type Date that has been set such that all `CacheEntry` objects last accessed before threshold should be removed.

First, here's the traditional, loop-based approach. (Often, you will see index-based looping instead of iterator-based looping, but since we're actually removing elements during the loop, it's important that we use an iterator.)

```
Iterator iter = cache.iterator();
while ( iter.hasNext() )
{
  CacheEntry entry = (CacheEntry) iter.next();
  if ( entry.getLastAccess().before(threshold) )
  {
    iter.remove();
  }
}
```

What's wrong with this? It's simple, it's familiar, and it works. Taken out of context, as a code snippet, it's even pretty easy to understand. Part of the reason that it is easy to understand, though, is because the looping idiom is so often used and so well understood that to experienced developers, the looping mechanisms are practically invisible. They are so used to seeing these mechanisms that they focus on the work being done in the loop. However, if you put this snippet in a section of code where you're doing more than just draining a cache, all that extra machinery for controlling the looping starts to clutter the code, making it less readable. Additionally, the power of the idiom recognition among experienced developers is such that errors in the looping machinery are actually more likely to be overlooked because the developer sometimes sees what he *expects* to see, rather than what is actually written.

Now, compare that with the following approach that uses a generic algorithm.

```
Removing.removeIf(cache, new LastAccessBefore(threshold));
```

That line of code accomplishes the same task as the previous looping construct. The difference is two-fold: the iteration, removal, and conditional (i.e., if the item was last accessed before the threshold) aspects of the problem are separated, and they are defined elsewhere. As we discussed earlier, that's advantageous from the perspective of reuse, but here we see that it also makes the code easier to read. Even though we have not shown the `Removing` class that implements the `removeIf` algorithm, and have not shown the implementation of `LastAccessBefore`, anyone reading this code can easily see what it does. By using generic programming techniques, we have clearly and concisely communicated the intent behind the code.

More information on generic programming

There are many generic programming resources available on the web. Despite the fact that many of them are language-specific or product-specific, they often contain information about generic programming that is generally applicable. Readers interested in learning more about generic programming are encouraged to check out these resources.

- "Chapter 12: Generic Programming and Collection Classes" of "Introduction to Programming Using Java", by David J. Eck, provides a good description of generic programming applied specifically to Java collections.
  http://math.hws.edu/javanotes/
- While very much STL-focused, David R. Musser's generic programming site has good information for those interested in generic programming in general.
  http://www.cs.rpi.edu/~musser/gp/
- "Combining OO Design and Generic Programming", by Angelika Langer, is also C++ oriented, but the concepts are applicable to other languages.
  http://www.langer.camelot.de/Articles/OOPvsGP/Introduction.htm

JGL: the Generic Library for Java

The Generic Library for Java, or JGL®, from Recursion Software, Inc., is a powerful add-on for the JDK that augments the Java Collections API with several additional collections, more than 50 generic algorithms, and over 80 functions and predicates. JGL is designed to work seamlessly with the Java Collections API, leveraging the developer's existing knowledge while providing access to a more advanced set of tools.

The original version of JGL was created to augment the JDK's support for collections, which at that time was extremely limited. Since then, the JDK has received significant enhancement to its collections capabilities, via the Java 2 Collections API. In the spirit of the original, the latest version of JGL augments the JDK by filling gaps in the Collections API and extending powerful generic programming practices to Collections API users.

How JGL Supports Generic Programming

JGL supports generic programming with a wide variety of algorithms that can be applied to standard `Collection` or `List` classes (classes implementing `java.util.Collection` or `java.util.List`, respectively). All JGL algorithms operate not only on JGL containers, but also on Java arrays of objects, Java arrays of primitives, and JDK Collections and Lists.

Using JGL algorithms

JGL algorithms are represented as static methods grouped into an algorithm class with other closely related algorithms. For example, all of the sorting algorithms are represented by static methods in the `Sorting` class. The JGL algorithm classes can be found in the `com.recursionsw.jgl.algorithms` package. A complete list of the JGL algorithms is provided at the end of this document.
Applying an algorithm to a collection is simple. The algorithm methods each take a `java.util.Collection` or `java.util.List` as an argument. The following line of code sorts the elements of a `Vector` named `myVector`:

```
Sorting.sort(myVector);
```

The above line of code sorts the vector according to the natural ordering of its elements. For more sophisticated needs, one can supply a predicate that is used to compare objects during the sort. Sorting a Vector of String objects in descending order, for instance, is nearly as simple as the previous sort example:

```
Sorting.sort(myVector, new GreaterString());
```

Don't be fooled by my use of a canonical algorithm, sorting, in these examples: JGL goes well beyond sorting in the algorithms it provides. It's just as easy to, for instance, reverse the elements:

```
Reversing.reverse(myVector);
```

…or transform the vector of strings into a Vector of the corresponding string lengths:

```
Transforming.transform(myVector, new LengthString());
```

…or shuffle the contents of the Vector:

```
Shuffling.randomShuffle(myVector);
```

…or replace the string "yellow" with "green":

```
Replacing.replace(myVector, "yellow", "green");
```

This is just a very small sample of the algorithms at the disposal of the JGL user. As we'll see in the next couple sections, JGL algorithms support a great deal of customization via predicates and can be applied to Java arrays in addition to collections.

Applying algorithms to Java arrays

An extremely useful feature of JGL is its ability to work with Java arrays of objects or primitives. If you find yourself working with Java array types, rather than collections, you can still use JGL algorithms--without the need to copy the data into a Collection. JGL provides a set of array adapters that allow you to treat an array of any primitive type, or an array of objects, as if it were a `Collection`. The ability to treat an array as a collection without copying data can be incredibly helpful in terms of keeping complexity low and performance high.

The following example demonstrates the use of JGL's `IntArray` adapter class, in conjunction with a Transmuting algorithm and the `NegateNumber` function, to negate the values in an array of type `int[]`. Note that the `IntArray` class is an adapter for the `int[]` and does not copy the values from the `int[]`.

```
        int[] array = { -1, 3, -5, 2 };
        Transforming.transformInPlace(new IntArray(array), new NegateNumber());
```

After this transformation, the contents of the array would be: `{ 1, -3, 5, -2 }`

Applying algorithms to part of a collection

Sometimes, you want to apply an algorithm to just part of a collection. JGL supports this via the concept of a *series*. `Series` is a class that represents a subset of the members of a given `Collection`. You can specify the membership of a `Series` using indices or iterators representing the starting and ending positions of the desired set of elements within the `Collection`.

The following line of code, for example, uses a `Series` to apply the `Printing` algorithm to 10 elements of a `Vector`, starting at index 25.

```
        Printing.println(SeriesFactory.create(myVector, 25, 10));
```

For more sophisticated needs, a `Series` can be specified using JGL iterators instead of indices. Refer to the JGL documentation for more information on JGL iterators and details on creating and using Series objects.

Parameterizing JGL algorithms with functions and predicates

While all JGL algorithms have simple, base forms that implement the algorithm using default options, many of the JGL algorithms allow you to customize their behavior using JGL function and predicate classes.

JGL predicate classes represent an operation to be applied to one or more arguments to produce a boolean result. They are often used for ordering objects or triggering actions. JGL functions are operations that apply to one or more arguments and return an `Object`. They are often used for selection or transmutation of their arguments.

For a better look at how predicates and functions are used to modify the behavior of JGL algorithms, lets take a brief look at the Counting algorithm.

The `Counting.countIf` algorithm counts the number of objects in the collection that meet the criteria specified by the given predicate. The following line of code returns the number of positive values in a `Vector` of `Integer` objects.

```
        Counting.countIf( myVector, new PositiveNumber() );
```

Here's a more complex example that showcases the ability to compose JGL functions and predicates. In this example, we sort a `Vector` of `String` objects by length. Since we don't already have a compare-by-string-length predicate at our disposal, we compose one from the `GreaterNumber` predicate, which is used to compare numbers, and the `LengthString` function, which will be applied to the arguments to produce their string lengths.

```
        BinaryPredicate orderByStringLength = new BinaryComposePredicate(
                new GreaterNumber(),
                new LengthString(),
                new LengthString());

        Sorting.sort(myVector, orderByStringLength);
```

The `BinaryComposePredicate` class allows you to build up complex predicates from other predicates and functions. The example above used `BinaryComposePredicate` to create a `BinaryPredicate` that applies the `LengthString` function to each of its two arguments, and uses the resultant values as arguments to the `GreaterNumber` predicate. This ability to compose new functions and predicates using existing ones provides us with flexibility in expressing how we want the algorithm to do its job.

Comparison of JGL algorithms with JDK algorithms

It's worth noting that the JDK does have some algorithms of its own. The `Collections` (not to be confused with `Collection`) class in the `java.util` package contains a number of algorithms that can be applied to collections.

The primary difference between the algorithms supplied by the JDK `Collections` class and those in JGL is one of breadth. The `Collections` class offers some canonical examples of algorithms, such as a binary search, a sort, and a handful of other algorithms. In all, the JDK `Collections` class provides about 18 distinct algorithms.[4] In addition, the JDK Arrays class, also in the `java.util` package, extends these algorithms to array types.

JGL, by comparison, offers well over 50 algorithms, and while the JDK lets you customize algorithms with a `Comparator`, it provides just one implementation of that interface; whereas JGL allows you to customize algorithms using predicates and functions, and provides you with well over 80 of those. Combine that with the fact that the functions and predicates both support composition, the fact that you can easily apply the JGL algorithms to a subset of the elements of a collection, and the more uniform application of algorithms to array types[5], and you very quickly begin to realize that the scope and power of JGL goes well beyond that provided by the JDK.

Does this mean that JGL is a replacement for the JDK's support for generic programming? No. The goal of JGL and the spirit of JGL has always been one of augmentation, not replacement. There is no need to choose between JDK classes and JGL classes. You can mix and match JDK and JGL classes as you see fit. For instance, if you prefer the JDK's sorting algorithm, you're not stuck writing your own Comparator implementations from scratch. JGL's `PredicateComparator` class allows you to use any JGL predicate as a `Comparator.` This is just one example of JGL's philosophy of enhancement-not-replacement in action.

Conclusion

Generic programming techniques allow Java programmers to make their algorithms more maintainable and reusable. The Generic Library for Java™, or JGL®, from Recursion Software, Inc., extends the Java Collections API to support generic programming with a powerful and extensive set of algorithms, functions, and predicates.

Through extending and enhancing the Collections API, rather than seeking to replace it, JGL gives the Java developer more power, more choices, and the ability to derive more value from the tools and the knowledge he already has.

JGL® is available from Recursion Software, Inc. for purchase and for evaluation. Visit http://www.recursionsw.com for details.

[4] I am not counting every method signature as a distinct algorithm, because several algorithms, such as sort, provide variant signatures to allow customization or additional control. This is a fine practice, and JGL does it as well--I'm just not counting those as distinct algorithms. Also not counted as algorithms, are the methods on the `Collections` class that create wrappers that deal with concurrency and modifiability.

[5] The `java.util.Arrays` class extends (not in the sense of formal Java inheritance) the algorithms of `java.util.Collections` to array types. However, this is done on an algorithm-by-algorithm basis. That is, Arrays contains a method for each algorithm, for each array type. In contrast, JGL uses an adaptation approach to make array types appear to implement `java.util.List`. As a result, the issue of application of algorithms to array types is solved once for all algorithms. Any new algorithms in future releases of JGL or algorithms that you develop on your own are immediately applicable to array types.

Appendix A: JGL Algorithm Roster

One of the main strengths of JGL is its complement of over 50 reusable algorithms that can perform tasks ranging from a simple filtering operation to a complex quicksort. Instead of embedding algorithms into containers, JGL algorithms are encapsulated within their own classes and can operate upon a wide variety of data structures, including Java `Collection` and `List` objects, native Java arrays of primitives or objects, and of course, JGL containers. Algorithms that are closely related are encapsulated as static methods of a single class. For example, all of the sorting algorithms are methods in `Sorting`.

The following is a roster of the JGL algorithms listed by their encapsulating classes. For the sake of brevity, the arguments to these methods are not shown. Many of these algorithms have several variants that allow the user to tailor their behavior to the problem at hand.

Applying

forEach()	apply a function to every element
inject()	iteratively apply a binary function to every element

Comparing

equal()	check that two collections match
lexicographicalCompare()	lexicographically compare two collections
median()	return the median of three values
mismatch()	search two collections for a mismatched element

Copying

copy()	copy the elements
copyBackward()	copy the elements backwards

Counting

accumulate()	sum the values of the elements
adjacentDifference()	calculate and sum the difference between adjacent element values
count()	count elements that match a value
countIf()	count elements that satisfy a predicate

Filling

fill()	set every element to a particular value
fillN()	set N elements to a particular value

Filtering

reject()	return all elements that do not satisfy a predicate
select()	return all elements that satisfy a predicate
unique()	collapse adjacent elements with the same values in a sequence
uniqueCopy()	copy elements, collapsing consecutive values

Finding

adjacentFind()	locate a sequence of adjacent elements
detect()	return first element that satisfies a predicate
every()	return true if every element satisfies a predicate
find()	locate an element
findIf()	locate an element that satisfies a predicate
some()	return true if at least one element satisfies a predicate

Heap

makeHeap()	make a list into a heap
popHeap()	pop the top value from a heap
pushHeap()	place the last element into a heap
sortHeap()	sort a heap

MinMax

minElement()	return the minimum element
maxElement()	return the maximum element

Permuting

nextPermutation()	change sequence to next lexicographic permutation
prevPermutation()	change sequence to previous lexicographic permutation

Printing

print()	prints elements to standard output
println()	prints elements to standard output followed by a newline
toString()	returns a description of a collection

Removing

remove()	remove all matching elements from a collection
removeCopy()	copy a collection, removing all matching elements
removeCopyIf()	copy sequence, removing all elements that satisfy a given predicate
removeIf()	remove elements that satisfy predicate

Replacing

replace()	replace a specified element in a collection with another
replaceCopy()	copy collection, replacing matching elements
replaceCopyIf()	copy collection, replacing elements that satisfy predicate
replaceIf()	replace specified elements that satisfy a predicate

Reversing

reverse()	reverse the elements in a list
reverseCopy()	create a reversed copy of a list

Rotating

rotate()	rotate a list by n positions
rotateCopy()	copy a list, rotating it by n positions

SetOperations

isSuperset()	returns true if the 1st collection is a superset of the 2nd
setDifference()	returns a collection containing the elements in 1st sequence that are not in 2nd
setIntersection()	returns a collection containing the elements that are in both given collections
setSymmetricDifference()	returns a collection containing the elements that are in one or the other, but not in both given collections
setUnion()	returns a collection containing the elements that are in either of two given collections

Shuffling

randomShuffle()	randomly change the positions of elements within a List

Sorting

iterSort()	create an iterator that will traverse a container in a sorted manner without reordering the container itself
sort()	order the elements according to their "natural order" or according to a specified predicate

Swapping

iterSwap()	swap the elements indicated by two iterators
swapRanges()	swap two ranges of elements

Transforming

collect()	return result of applying a function to every element
transform()	apply a transformation function to each element, putting the transformed elements into another collection
transformInPlace()	apply a transformation function to each element

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