Generics in Java;
A Quick and Semi-Critical Overview

Ben Munat
March 3, 2004

1. Summary

Generic types allow a single piece of code to be reused with different types inserted. They are therefore considered a powerful tool in the never-ending quest for code reusability. They are "so important that even a language that lacks them may be designed to simulate them." [BOSW98] Java has long been one such language, but with the impending release of version 1.5 – now in beta – Java has now finally joined the stable of "generic-enabled" languages.

This paper will give a quick overview of generic types and then discuss how generic types have been implemented in Java, including their syntax and some ways in which they diverge from previous approaches. The design decisions forced upon Java generics in order to retain backwards compatibility created some unfortunate quirks. These will be illustrated in a section on the caveats of using Java generics.

2. Genericity Overview

Generic types are also known as *parameterized types*, and they allow a programmer to abstract over types. [Bra04, TT99] This means that a section of code – a class, a method or, in Java, an interface – can be defined with parametric, or unknown, type values. This generic type (or method) is then instantiated with actual type arguments, which the compiler inserts in the appropriate slots. This results in a fully defined type that uses the exact component types desired by the author.

For example:

```
class Foo<T> {
      // do something with T
}
...
Foo<Integer> f = new Foo<Integer>();
```

The type variable "T" is a placeholder for some type. When the code is compiled, the T is replaced with the type actually used, `Integer` in this example. This example also points out a difference of Java generics from C++ templates: the type variable in Java cannot be a primitive type. The primitive wrapper classes must be used.

In C++, generics are known as templates. This is because it allows one to define a template for a section of code, into which a variety of parameter types can be inserted to get different code. The implementation of generics in Java has turned out to have some significant differences from C++ templates, as outlined below.

Inheritance – the core mechanism for polymorphism in Java and all object-oriented programming languages – could also be considered a form of genericity. By using an interface or a class higher up the inheritance hierarchy, one can be less specific about what actual type will be used. Generic types are the converse of this. To state the difference in a succinct manner: inheritance allows one to be type specific when *implementing* a class but generic when *using* the class, while generics allow us to be specific when *using* the type but generic when *implementing* it. [Qua03]

The canonical example of the benefits of using generics is collections. Currently in Java, the collection classes are implemented as containers of class Object, the root of the Java class hierarchy. Any subtype of Object—which includes all Java classes—may be inserted into a container. However, in order to restore that object's class identity on removal from the container, it must be cast. Proponents of adding genericity to Java claim that these casts are a potential source of error. [BOWS98, BCK+03] Whether this is true or not, it is clearly less ambiguous to specify the type of object that a collection will hold at the time it is declared.

Here is an example provided by [Bra04]:

Without generics:

```
List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer) myIntList.iterator().next(); // 3
```

With generics:

```
List<Integer> myIntList = new LinkedList<Integer>(); // 1′
myIntList.add(new Integer(0)); //2′
Integer x = myIntList.iterator().next(); // 3′
```

In the generic version, the List is declared to hold Integers. Therefore, the cast on line three is no longer required. Besides the somewhat arguable benefits of being less prone to error and clearer of intent, the generic version has the undeniable benefit of being checked at compile time. The pre-generic version runs the risk of a ClassCastException at runtime and requires the user/programmer to keep track of what type their collections hold.

3. Java Generics vs. C++ Templates

In the interest of backwards compatibility, the creators of Java generics wanted to avoid adding any new keywords to the language. [BOSW98, BCK+03] So, rather than use the "template" or "typename" keywords as in C++, in Java one simply declares the generic

entity with type parameters in angle brackets, as shown above. One very nice detail carefully considered by the Java generics implementers was the problem C++ has with the use of a templated type with a templated type. This results in syntax similar to "`list<vector<T> >`" in which the space between the two ">" symbols is necessary to prevent the compiler to interpreting the ">>" as a right shift operator. The Java compiler is smart enough to recognize the context of the ">>" as being a nested parameterized type.

Beyond simple syntactic differences, there are fundamental differences in the way these two languages implement parametric types. C++ generics are called templates for a reason; the programmer writes a piece of code—a function, class, method, or even a simple expression—that is templated with a placeholder for the type to be replaced. When the program is compiled, this code can be used repeatedly but with different types inserted for the templated parameter. This effectively means that different code is generated, with the exact type inserted. [TT99]

In Java, the declaration of the templated type is replaced by the most general type applicable and, in the bytecode, a cast is inserted in every instance in which the type is used. [BCK+03] This means that generics in java *do not* produce different code for each type used as a parameter. This reality has led some online programming pundits to label Java generics as mere "syntactic sugar". This is a little extreme, however, as it belittles the power of the *constrained* generics that Java offers, which will be explained next. On the other hand, the inherent difference of approach is clearly noteworthy: C++ produces different code based on the templated types; Java produces largely the same code (depending on the type constraints) with behind the scenes type casting.

4. Constrained Genericity

The "constraints" referred to above are the other significant difference between C++ templates and Java generics, and are a strong factor in Java's favor. C++ has what is known as "unconstrained genericity". [TT99] This means that its parameterized types are simply replaced by a specified type when the templated code is instantiated. No further processing is done on the types. It is up to the user to interpret the applicability of the type being inserted (e.g. is a type being used in a sorted collection "comparable"). In the "constrained" generics offered by Java (as well as previously by languages such as Eiffel), the parameterized type may also include a constraint on its place in the type hierarchy. For example, declaring "`List<T extends Shape>`" will tell the compiler to ensure that whatever type is replaced by "T" is a subtype of Shape.

Confusingly enough, the type parameter lists *does not* follow the keyword conventions of class declarations. The "extends" keyword is used for "implements" as well. So, you wouldn't say "`Vector<T implements Serializable>`", but would use the "`extends`" keyword instead. Additionally, since generics can be defined over more than one type, the comma symbol has already been used to set off members in the list of parameterized types. Therefore, if the type variable is to be constrained by more than one interface, the '&' symbol is used instead.

Despite these potential sources of confusion, this ability to specify the type over which a class will operate, but without being *too* specific, is very powerful. Java programmers are no longer required to use impersonal containers, which treat everything as faceless Objects. They can now not only use containers that are a bit more personable—acknowledging that their members are of a particular class—but can also give a collection only as much information as is needed. This is keeping in the classic Java (and OOP) spirit of being as general and therefore polymorphic and flexible as possible.

In addition, Java generics have taken these constraints one step further, implementing what is known as "F-bounded polymorphism". This means that it allows the type variable to appear as part of its own bounds or as bounds of other type parameters declared in the same section. [BCK+03] In other words, the parametric type declarations can be recursive.

For example:

```
class SortedList<T extends Comparable<T>> {

    // do something with T

}
```

Here the type variable T is used within a parametric type that is in turn used as a constraint on another parametric type.

5. Wildcards

Another feature offered by the Java implementation of generics is the "?" wildcard. This can be used in place of the type variable in a parameterized type to tell the compiler that the type is "unknown". This is useful when the actual type is unimportant; in other words, when the type is not going to be replaced but is merely "some" type to be worked with.

Though this single wildcard character may seem trivial and even unnecessary, there are instances when it is indispensable. Consider a method that takes a collection as an argument and performs some action on all members of the collection. If you merely declare your method to work on a collection of some type T, then that collection can only hold objects of whatever type the type variable T is replaced with. This means that the collection argument to your print method *can only take Collections of T*. Thus, you've effectively killed any polymorphic capability for that method.

This may still seem strange and rather un-Java-like. In order to understand why this happens, consider this unexpected subtlety of generic types: the subtyping relationship of two types used as type parameters does not carry through to the parameterized type. In other words, if `A extends B`, it does *not* follow that `SomeClass<A>` is a subtype of `SomeClass<B>`. Therefore, if we pass our method a collection of a specific type, we cannot then use this method with collections of other types, *even subtypes of the original*

specific type. If, however, we declare the method to take a "collection of unknown" – written `Collection<?>` – we can then call this function with a collection of any type. And, if we declare the method with `Collection<? extends Sometype>`, we can pass in collections of any subtype of `Sometype`, therefore getting the polymorphism for which we were originally hoping. As an example, "`Vector<? extends JComponent>`" would be a vector which can hold "any type that extends JComponent".

Now, of course, nothing comes for free. The ? symbol is *not* translated into a specific type, so it cannot be used in a place in which an actual type is needed. For example, our collection of "`<? extends Someclass>`" cannot be written into because the actual type is not known. A collection of unknown effectively becomes *read-only*; we have exchanged writability for polymorphic behavior. We know what the *upper bound* of the unknown type is, so we can read things out of the collection knowing that they are all that type or a subtype.

The Java generics implementation also allows a wildcard to act as a *lower bound*, by writing "`<? super Someclass>`". Interestingly enough, this offers us the ability to make *write-only* collections. Knowing the lower bound of our collection means that we can put in objects of that lower bound, or some subtype of that lower bound.

Another way of putting this is that the "`<? extends …>`" notation indicates a *covariant* subtyping relationship, while the "`<? super …>`" notation expresses a *contravariant* relationship. [THE+04] For example, a `List<? extends Integer>` is a subtype of `List<? extends Number>` and `Integer` is a subtype of `Number`, so this is a covariant relationship. Meanwhile, a `List<? super Number>` is a subtype of `List<? super Integer>`, leading to a contravariant relationship; i.e. the types of the collections vary inversely to the types of the arguments.

6. Another Caveat

The somewhat counterintuitive aspects of Java generics already described may seem like enough of a potential source of confusion. There is, however, another major caveat that comes as a result of the implementation decisions of the Java generics committee. As mentioned earlier, it was important to Sun Microsystems and to the designers of Java generics that adding generics to the language (and indeed adding all of the new features in Java 1.5) not break backwards compatibility. In other words, they could not make any changes to the Java virtual machine.

This means, as mentioned in part three above, that Java generics do not produce different code depending on the type passed as its parameter. The desired restriction of type is achieved by casts in the bytecode. The important consideration here is that casts do not make for different resulting types. Therefore, a `List<Integer>` has the same type as a `List<String>`, and if one performs reflection on them, they will find that they both have the same run-time class. [Bra04]

The result of this intriguing detail is that generic types cannot be used in casts or with the `instanceof` operator. In other words,

```
if (foo instanceof List<T>) {…}
```

and

```
foo = (List<T>)bar;
```

are illegal. Also, since generic types share the same class, they also share the same static members; *all* instances of a class, regardless of the generic parameter, share the same static methods and variables. This caused some strange quirks in earlier versions of Java generics: ClassCastExceptions from code that contains no casts! [All03] However, in the Java 1.5 release, it is now a compiler error to reference a generic type *or* a type variable in a static context.

Another potential source of phantom `ClassCastException`s was when using generic types as the component type of an array. The array could be declared to be an array of a generics, but then cast to an array of Object, after which some other type could be stored in it. Then, when later removing items from the array into a variable of the original type, one would get the ClassCastException when removing the extraneous element. Here is an example from [Bra04]:

```
List<String>[] lsa = new List<String>[10]; // not really allowed
Object o = lsa;
Object[] oa = (Object[]) o;
oa[1] = new ArrayList<Integer>(); // …passes run time store check
String s = lsa[1].get(); // run-time error - ClassCastException
```

Thus, regular generic types are not allowed as the component type of an array. Unbounded wildcard generics, however, are still okay, because there is no type information to be lost in the array's anonymity.

7. Conclusion

There are other aspects of Java generics not considered here: in particular, generic methods and integrating legacy code with generic code. However, the major characteristics and concerns have been covered.

In review, generic or parameterized types allow one to write a class, interface or method using placeholders for a type, which will be replaced when the class, interface or method is used. Different instantiations of these classes, interfaces or methods can replace the type variable with different types. In addition, Java offers constrained generics, in that a supertype – either a class or interface – can be specified to constrain the type variable.

Java generics also offer the "?" wildcard to represent "some unknown type", which is useful when one does not need the actual type information and does not want to overly restrict oneself. This wildcard character can also be constrained by the "extends" or

"super" keywords, indicating the upper or lower bound, respectively, of the unknown type.

Being basically a "pre-compilation translation", Java generics also have some quirks. These include sharing a common runtime type and common static members regardless of the type variable. This means that they may not be cast or used with the `instanceof` operator, and may only be used in arrays if their type variable is the unbounded wildcard.

Despite these quirks and potential conceptual difficulties, adding generic types to the Java programming language—thus allowing collections with types more specific than Object and allowing programmers to shuffle the responsibility for correct casting off onto the compiler—is a welcome addition to the language. There will undoubtedly be a rough period of transition, but in a few years, Java programmers will likely wonder how they ever lived without generics.

[All03] Eric Allen. Diagnosing Java Code: Java Generics Without the Pain, Part 2. available at <http://www-106.ibm.com/developerworks/java/library/j-djc03113.html>. March 2003

[BCK+03] Gilad Bracha, Norman Cohen, Christian Kemper, Martin Odersky, David Stoutamire, Kresten Thorup, Philip Wadler. Adding Generics to the Java Programming Language: Public Draft Specification, Version 2.0. June 2003

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler. Making the Future Safe for the past: Adding Genericity to the Java Programming Language. *Presented at OOPSLA 1998*

[Bra04] Gilad Brach. Generics in the Java Programming Language. *Tutorial available at:* <http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf> February 2004,

[THE+04] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding Wildcards to the Java Programming Language. SAC'04, March 14-17, 2004

[TT99] Kresten Krab Thorup and Mads Torgersen. Unifying Genericity: Combining the Benefits of Virtual Types and Parameterized Classes. In ECOOP Proceedings. Springer-Verlag, Lisbon, Portugal, June 1999

[Qua03] Matt Quail. Generic <Java> An Introduction to Generic Types in Java. *PDF presentation available at:* <http://www.cjugaustralia.org/slides/generics.PDF>. March 2003