The Role of Wildcard Types in Java Generics

Matthew Justin
University of Minnesota, Morris
600 E 4th St.
Morris, MN 56267
just0059@morris.umn.edu

ABSTRACT

This paper describes two language constructs recently added to the Java programming language, as well as the theory behind them. The constructs are *generic types*—also known as *generics*—and *wildcards*. Generics are a feature found in other languages, most notably in C++ as *templates*. Generics allow a class or method to abstract over some type information. Wildcards extend generics by defining a type-safe type hierarchy among generic types. *Variant parametric types*, a concept central to the theory behind wildcards, is discussed.

General Terms

Parametric polymorphism

Keywords

Java generic types, Java wildcards, Variant parametric types

1. INTRODUCTION

The Java programming language recently introduced into its version J2SE 5.0 release (October 2004) a language feature known as *generic types* or *generics*. Generics implement a concept known as *parametric polymorphism* or *genericity*[10], which allows classes and methods to abstract over some type information[9].

Generic types have been available in other languages for many years. The most notable example is *templates* in C++. While Java initially did not include generics in its design, they were eventually added because "many programs and libraries written in the Java (tm) programming language are intrinsically generic. However, the Java programming language lacks the ability to specify generic types. As a result, programs are unnecessarily hard to read and maintain, and are more likely to fail with runtime type errors"[7].

However, Java had unique constraints that made existing solutions unsuitable for inclusion in the language. First, Java is a strongly typed language; type checks are performed by the compiler to guarantee type safety. This guarantees that all operations on each value are defined. A benefit of this is that, barring type casts and certain other potentially unsafe operations, programmers are notified at compile time if their code is not type safe. This is advantageous since it is easier for a programmer when errors are discovered at compile time rather than at run time.

Another central feature of Java is its type hierarchy. Since Java is an object-oriented language, the concepts of inheritance and subtype polymorphism are very important for writing code that follows the best practices of software design. Assuming that a type A is a subtype of a type B, an object of type A should be able to be used anywhere an object of type B is required. For example, if a method requires an object of type Number, and if Integer is a subtype of Number, then that method should be able to accept an Integer object and still work correctly. In Java, this is implemented by inheritance—each class (other than java.util.Object, the root of the type hierarchy) has exactly one direct superclass. In addition, a class can implement any number of interfaces. The class is then a subtype of its superclasses and interfaces.

In order to add generics to Java, it is imperative that type safety be preserved. In addition, to be most useful, it is important that generics fit in well with the existing type hierarchy. Adding generics to Java while meeting these criteria has proven to be challenging. The inclusion of generics in Java therefore was preceded by long academic debate, and a number of different designs were proposed[10]. The length of this debate is indicated by the amount of time it took to add generics to the language. The official proposal to include generics in the language, made through Sun's Java Community Process program, was made in 1999[7]. However, generics were not added to the language until 2004. In addition, various designs were developed for including generics even prior to the official proposal for inclusion in the language.

To meet the type safety and type hierarchy needs of Java, a feature known as *wildcards*, based on a theoretical concept known as *virtual parametric types*, was included with generics.

2. GENERICS

As an example of a typical use of generics, consider a class representing a list of objects. Depending on its context, a given list often represents a more specific list than just that of object, such as a list of integers or a list of strings. The underlying logic to manipulate the list—e.g. inserting and reading objects—-is the same for all lists, regardless of type. Creating a separate class for each type of list (e.g. IntegerList, StringList) results in a large number of classes that differ only in the type of element that can be stored in the list. Generics solve this problem by allowing one class, in this case List, to be written. When the class is used, the type of the list (String, Integer) is declared. A type that a generic class or method is abstracted over is known as a *type parameter*.

2.1 Basic generics

Prior to the inclusion of generics in Java, genericity would typically be simulated using the "generic idiom"[2]. To simulate generics under this idiom, the type of variables are replaced by their common supertype, and cast to the appropriate type when accessed. This common supertype is typically the type Object, since it is the supertype of all non-primitive types in Java.

As an example, consider a list of integers. We want to return the sum of all elements of the list. Using the generic idiom with the list, we must cast any objects returned by the list:

```
public int sum(List list) {
    int result = 0;
    for (int i = 0; i < list.size(); i++) {
        Object o = list.get(i);
        Integer nextInt = (Integer) o;
        result = result + nextInt;
    }
    return result;
}
```

A major problem with the "generic idiom" is that there is no way to enforce that the variable *list* contains only Integer values. If a list containing a non-Integer value is passed into this method, the programmer will not be notified that there is an error in the program until run time, when the method fails to cast the object to an Integer, resulting in a run-time error.

Another problem with this idiom is that the program is more difficult to read. To indicate that a given list represents a list of Strings under the idiom, the programmer must resort to comments or some other extralinguistic method. In addition, the typecasting makes the program longer in a way that obfuscates the intent of the program.

The addition of generics to the language solves the problems mentioned in the previous example. The same code, using generics, looks as follows:

```
public int sum(List<Integer> list) {
    int result = 0;
    for (int i = 0; i < list.size(); i++) {
        Integer nextInt = list.get(i);
        result = result + nextInt;
    }
    return result;
}
```

Note the syntax used for a generic object—the type parameter is given in angle brackets after the class name.

The generic version of the method enforces that the list contains only Integer values. If any other type of list is passed in to the method, the code will fail at compile time, rather than at run time. This allows the programmer to notice the error earlier, greatly simplifying debugging. Another benefit of the generic version of the method is that since it is known at compile time that the list contains only Integer values, the cast is no longer needed, leading to cleaner code.

2.2 Type Parameters

To indicate that a method or class is generic, *type parameters* are used. A type parameter may be of any object type (thus primitive types, such as int or float, are not allowed). As an example of a type parameter, the following is an excerpt from the definition of the interface List in java.util[8]:

```
interface List<E> {
    boolean add(E o);
    E get(int index);
}
```

In this case, E is the type parameter of the list. When the list is used, E is replaced by a specific type. For example, if the code references List<Integer>, E is treated as having the type Integer. Thus the list behaves as if its signature is:

```
interface List<Integer> {
    boolean add(Integer o);
    Integer get(int index);
}
```

Calling the *get* method on the list will return an Integer, and any value passed to the *add* method must be an Integer.

A method can be generic, even if the class containing it is not. As an example, take the following method, which returns the first element of a list:

```
public <T> T getFirst(List<T> list) {
    T result = list.get(0);
    return result;
}
```

The first <T> in the method declaration indicates that the method is generic, and parameterized on the type variable T. This means that all types T in the method refer to the same type. The rest of the declaration states that the method takes a list of type T and returns an element of the same type T that the list is parameterized on. For example, if the method is called with a list of type List<String>, the type of T is replaced by String for the duration of the method, thus returning a String.

2.3 Subtyping With Generics

The subtyping rules with generics do not work as might be expected. For example, List<String> is not a subtype of List<Object>, even though String is a subtype of Object. The reason for this is that subtyping in this fashion causes issues with type safety. As a specific example, if this type of subtyping were legal, the following unsafe code would be possible[1] (typecasting added for clarity):

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
Object o = (Object) new Integer(5);
lo.add(o);
String s = ls.get(0);
```

The problem demonstrated by this example is that the last line of the code is attempting to assign an Integer value to a String variable. This is a problem since an Integer is not a subtype of String. Since the Java type system guarantees that the value of a variable matches its type, this assignment is illegal.

This works differently than Java arrays, where String[] is a subtype of Object[]. The way Java deals with this issue with array types is to check at runtime whenever an object is assigned to an array, throwing an exception if an incompatible value is assigned[4]. This has the undesirable result of shifting the error from compile time to run time, complicating debugging. The subtyping rules used by generics avoid the performance overhead of runtime checks, and are guaranteed to be safe at compile time.

However, not being able to subtype generics introduces problems. As an example, consider the case where we want to print the elements of a list. A naive implementation using generics might look like this:

```
public void printList(List<Object> list) {
    for (int i = 0; i < list.size(); i++) {
        System.out.println(list.get(i));
    }
}
```

A problem with this method is that it can only print objects of type List<Object>. We can not, for example, call the method with a list of type List<String>, since as demonstrated earlier a List<String> is not a subtype of List<Object>. Ideally, the method should be able to take in any type of list. To do so, we would need a variable type that is a supertype of all variables of type List, regardless of the value of their type parameter.

3. WILDCARDS

To handle subtyping with generics, the feature known as *wildcards* has been created. "A wildcard is a special type argument '?' ranging over all possible specific type arguments, so that List<?> is the type of all lists, regardless of their element type"[10]. Using wildcards, the previous example would look as follows:

```
public void printList(List<?> list) {
    for (int i = 0; i < list.size(); i++) {
        System.out.println(list.get(i));
    }
}
```

Since a List<?> represents the type of all lists, this method may be called with List<Object>, List<String>, or with any other type of List.

3.1 Unbounded Wildcards

A wildcard of the form C<?> is known as an *unbounded* wildcard. As demonstrated, this type of wildcard is useful when the operations to be performed on the parameterized object do not depend on the type of the parameter.

As previously described, List<?> is a supertype of List<T> for any type T, meaning that any type of list can be assigned it. However, in order to maintain type safety, restrictions must be placed on the use of the list.

Since we do not know the actual element type of the list, we cannot put non-null elements into it. We are allowed to read Objects from the list, since the type parameter must be an object type, and all objects are of type Object.

3.2 Bounded Wildcards

Often the element type of a parametric class is not completely irrelevant, but is required to conform to some bound. This is expressed using a *bound* on the wildcard[10]. Syntactically, a type bound is a type parameter of the form <? extends T> or <? super T>, for a type T.

3.2.1 Extends Bounds

The *extends* keyword in a wildcard expresses that the element type is required to be either the given type or a subtype of it. As an example of a use of this, consider the following code:

```
interface Shape {
    public double area();
}
class Square implements Shape {
    public double length;
    public double area() {
        return length * length;
    }
}
class Circle implements Shape {
    public double radius;
    public double area() {
        return PI * radius * radius;
    }
}
```

If we want to sum up the areas of a list of shapes, we could use a method like:

```
double sumAreas(List<Shape> list) {
    double result = 0;
    for (int i = 0; i < list.size(); i++) {
        Shape shape = list.get(i);
        result = result + shape.area();
    }
    return result;
}
```

However, it makes sense to allow a List<Circle> or a List<Square> to be passed in to the method. We cannot use an unbounded wildcard, as it would not produce legal code, since the method *area()* is not defined in Object. To get the desired functionality, the extends bounds can be used:

```
double sumAreas(List<? extends Shape> list) {
    double result = 0;
    for (int i = 0; i < list.size(); i++) {
        Shape shape = list.get(i);
        result = result + shape.area();
    }
    return result;
}
```

Since the actual element type of the list is unknown, non-null objects cannot be written to the list. However, since it is known that the elements of the list are of type Shape, it is legal to read Shapes from it, as shown in the example.

3.2.2 Super Bounds

In addition to the *extends*-bounded wildcards, Java allows *super*-bounded wildcards. Whereas *extends*-bounded wildcards require the type parameter to be a subtype of the given type, types parameterized with *super* require the type parameter to be a supertype of the given type.

Returning to the list example, given List<? super Number>, Numbers can be written to the List. However, all that is known of the specific element type is that it extends Object, and thus reading from the list will return elements of type Object.

One place where a *super*-bounded wildcard is useful is with Comparator[1] objects, as demonstrated in the following example, taken from [10]:

```
interface Comparator<T> {
    int compareTo(T first, T second);
}
```

A typical use of a Comparator is to define the ordering of a sorted collection, such as a TreeSet. For example, when constructing a TreeSet<String> using a Comparator, a Comparator that can compare Strings is needed. Both a Comparator<String> and a Comparator<Object> would work. In this case, Comparator<? super String> is appropriate, as it is the common supertype of types that can compare Strings.

3.3 Type Inference

A generic method can be called with or without explicit type arguments. Explicitly specifying a type parameter has the following syntax, where *ClassName* is the name of the class containing the method, *T* is the type parameter to use, and *methodName* is the name of the method.

```
ClassName.<T>methodName
```

As an example, consider the generic method singleton(T o) in java.util.Collections. This method, parameterized on some type T, takes an element and returns a Set<T> containing only the given element. Given variable *set* of type Set<String>, the following example shows the method being called both with and without an explicit type argument:

```
set = Collections.<String>singleton("a");
set = Collections.singleton("a");
```

In the first line, the *<String>* indicates that the singleton method be called with a type parameter of type String.

In the absence of an explicit type argument, the variable type is inferred by the compiler. "Inferring a type for a type variable T means selecting a type that by insertion produces a method signature such that the given call site is type correct, and ensuring that the type satisfies the bound for T. In the process a subtype is preferred over a supertype because the former generally preserves more information about return values"[10]. This process is known as *type inference.* Type inference is only used on generic methods, as class type arguments are required to be stated explicitly.

Type inference is demonstrated in the following example, similar to one from [10]. Consider the following declarations, noting that Set<T> and List<T> are both subtypes of Collection<T>:

```
<T> T choose(T a, T b) { ... }
Set<Integer> intSet = ...
List<Number> numberList = ...
choose(intSet, numberList);
```

In the call choose(intSet, numberList), a type has to be found for T that is a supertype of both Set<Integer> and List<Number>. Various types are possible, including Object, and Collection<?>. In this case, the most specific legal type is Collection<? extends Number>, which is what the compiler infers the type to be.

While the idea of type inference is relatively easy to understand, the actual algorithm used by the compiler is somewhat complex[4], and beyond the scope of this paper. Notably, "there may be both an upper and lower 'best bound', so the choice between them would have to be arbitrary"[10].

The complexity underlying type inference is due to the existence of wildcards. Using the *choose* method in the previous example, if the type parameters of the method parameters are the same type, the result is the same with or without wildcards. For example, the most common supertype of Set<Integer> and List<Integer> without using wildcards is still Collection<Integer>. However, if the type parameters don't match, as with List<Number> and List<Integer>, the most common supertype that doesn't use wildcards is Object, rather than than the wildcard version List<? extends Number>.

In rare circumstances the type parameters for a generic method have to be explicitly specified, for example when a generic method expects no arguments[3]. As a concrete example, consider the method emptyList() in java.util.Collections, which returns a List<T> containing no elements. If the return value is assigned to a variable, as in the following example, then the compiler can infer the type based on the variable it is being assigned to:

```
List<Integer> list = Collections.emptyList();
```

However, if the return value of the method isn't assigned to a variable, but instead for example is being passed directly to a method, type inference will not infer the desired type. Returning to the method sum(List<Integer> list) that sums a list of integers, we have the following:

```
sum(Collections.<Integer>emptyList());
sum(Collections.emptyList()); //Not legal
```

In this example, the first line will work, as we are explicitly setting the type of T to be Integer. However, the second line will not compile. The compiler's type inference algorithm returns a List<Object>, not a List<Integer>. This results in an illegal type being passed into the sum method, preventing the program from compiling.

3.4 Wildcard capture

The following example, taken from [10], describes an issue with wildcards as presented thus far. Consider the following method, slightly modified from the version in the java.util.Collections class, which returns a read-only view of a Set<T>:

```
<T> Set<T> unmodifiableSet(Set<T> s)
```

This method to be called on a Set<T>, for any type T. However, using wildcards as described thus far, it cannot be called on a Set<?>, because the actual element type is unknown.

[1]A Comparator compares two objects of a given type and returns a value representing which of the objects is "greater" than the other

"The solution is to observe that it is actually safe to allow the method taking a Set<T> to be called with a Set<?>. We may not know the actual element type of the Set<?>, but we know that at the instant when the method is called, the given set will have some specific element type, and *any* such element would make the invocation typesafe. This mechanism of allowing a type variable to be instantiated to a wildcard in some situations is known as *wildcard capture* because the actual run-time type behind the ? is 'captured' as T in the method invocation"[10].

One use of wildcard capture is to allow method signatures to more accurately represent the types of values they accept. Consider the method reverse in java.util.Collections, which reverses the order a list[8]. From the caller's perspective, this method can be called on any type of list, and thus the signature of reverse should be as follows[10]:

```
void reverse(List<?> list)
```

However, the method body needs to know the type of element stored in the list, so it can remove and re-insert elements. Thus, the method needs the signature to be parameterized:

```
<T> void reverse(List<T> list)
```

Wildcard capture allows the programmer to achieve this because it allows "the wildcard version of the method (which should be public) to call the polymorphic version (which should be private and have a different name)"[10].

4. THEORETICAL BACKGROUND

The theory of Java wildcards is based on the theoretical concept of *variant parametric types* (VPTs)[5][6]. "Variant Parametric types are a generalization of standard parametric (or generic) types, where each type parameter may be associated with the variance annotations +, -, or *, respectively referred to as the covariance, contravariance, or bivariance annotation symbols"[5]. The variance annotation symbol precedes the type parameter, as in List<+Integer>. Each of the variant annotation symbols corresponds to a different wildcard type: "type C<? extends T> stands for C<+T>, C<? super T> for C<-T>, and C<?> for C<*>"[6]. As seen here, C<*T> is often abbreviated C<*>.

While wildcards are based on the theory of VPTs, there are some changes made to suit the particulars of the Java programming language. With VPTs, nothing can be read from a List<-T>, and nothing can be written to a List<+T>. However, in Java Object is the common supertype for all object types, so Object can be read from a List<? super T>. In addition, since any object type can be assigned a null value, null can be written to a List<? extends T>. The same differences occur for List<*> and List<?>. More generally, in VPTs methods of a class of type C<+T> or C<*> cannot accept elements of type T as method parameters, and methods of a class of type C<-T> or C<*> cannot return elements of type T.

Type inference is specific to Java wildcards; with VPTs, all types are explicitly stated. Wildcard capture, on the other hand, is supported by the theory of VPTs. However, the theory behind wildcard capture in VPTs is well beyond the scope of this paper.

5. CONCLUSIONS

Generics were added to the Java programming language in its 5.0 release. Generics allow classes and methods to abstract over type information. This allows for greater type safety, as the previously used "generic idiom" relied on unsafe typecasts.

The concept of wildcards has been introduced to preserve type safety while allowing a larger set of generic values to be used. This allows method parameters to be values that they logically should be able to be. In exchange for being able to use this larger set of values, objects parameterized with wildcard values are limited in how their methods can be called. Wildcards introduce some complex issues, such as type inference and wildcard capture. Java wildcards are based on the theory of variant parametric types, with changes made to meet the specific needs of the Java programming language.

6. REFERENCES

[1] G. Bracha. *Generics in the Java Programming Language*. [Online], http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, 2004.

[2] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In C. Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[3] D. Flanagan. *Java In A Nutshell, 5th Edition*. O'Reilly Media, 2005.

[4] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2005.

[5] A. Igarashi and M. Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Trans. Program. Lang. Syst.*, 28(5):795–847, 2006.

[6] G. Rimassa and M. Viroli. Understanding access restriction of variant parametric types and java wildcards. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1390–1397, New York, NY, USA, 2005. ACM Press.

[7] Sun Microsystems, Inc. *Java 2 Platform Standard Edition 5.0 API Specification*. [Online], http://jcp.org/en/jsr/detail?id=14, 1999.

[8] Sun Microsystems, Inc. *Java 2 Platform Standard Edition 5.0 API Specification*. [Online], http://java.sun.com/j2se/1.5.0/docs/api/index.html, 2004.

[9] K. K. Thorup and M. Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 186–204, London, UK, 1999. Springer-Verlag.

[10] M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the java programming language. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1289–1296, New York, NY, USA, 2004. ACM Press.