Java From The Trenches:
Dealing with Object Orientation and Generics

Jaime Niño
Computer Science department
University of New Orleans
New Orleans, LA 70148
jaime@cs.uno.edu

ABSTRACT

Generic typing is a very desirable and useful language feature; in particular it is a welcome feature in the teaching of data structures in CS2. Hence its introduction in Java in 1.5 version was indeed a welcome addition to one of the most popular languages of current use in academia. On initial inspection the definition and use of generics is straightforward, but on closer inspection generic types do not interact smoothly with the object orientation paradigm. Wilcards is a Java 1.5 mechanism introduced to deal with generics and Object Orientation, in particular with subtypes. This papers presents the use of wildcards and provides guidelines in teaching its use in the design of generic classes and generic methods.

INTRODUCTION

Java 1.5 introduced genericity to the language for interface and class definitions. Unlike the previous use of `Object` to try to attain the same results, Java 1.5 generics is supported by the compiler via type checking. The genericity introduced allow us to define generic types as well as generic methods where in either instance *type* is a parameter.

It is now the job of the educator to bring its use to the students, in particular those in CS2 and Software Engineering. Initial experience tells that the teaching of the definition of Java generic classes should go further than covering the mechanics of how to abstract over a group of types to define one generic class, say a generic `List` type, and then how to instantiate them; or how to abstract over a type to define one generic method, say a generic `swap` method. One must also discuss the assumptions about the operations the parameter type may require as well as the interaction of between generic types and the types resulting from their instantiations with subtypes. Finally one must prepare the student to read Java generic code in general, and the re-engineered Java API for generics in particular.

A Java generic type is used by providing an instantiation, whereby type arguments are provided for the generic type parameters specified in the generic definition; the resulting instantiation yields reference types called *parametric types*. For instance, a generic list type `List<T>`, gives rise to the parametric type `List<Integer>` via instantiation of `T` with `Integer`. One of the basic principles of object orientation, the Liskov substitutability principle or LSP, is the interaction between subtypes introduced via inheritance and their supertypes, whereby subtype instances can substitute supertype instances where supertype instances are expected. The question whether generic instantiation preserves LSP yields a negative answer as we will see; thus, although `Integer` is a subtype of `Number`, `List<Integer>` is not a subtype of `List<Number>`.

This negative result gives rise to two questions: First, what is the supertype of all generic instantiations? Second, can we still write generic methods as done when using `Object` but type checked? The *wildcard* type extension also introduced in Java 1.5 allow us to answer both questions satisfactorily. By learning how to best use unbounded and bounded wildcards we can greatly improve the expressibility of generic method specifications in spite of the negative result on preservation of the subtype relation via generic instantiations. We must teach this issue to our students to allow them deal effectively with generic code.

From the point of view of teaching we must ultimately explain why `List<Circle>` is not a subtype of `List<Figure>`; we also need to explain that if we specify a method that computes the area of a list of `Figure` as:

```
public double totalArea(List<Figure> figures) { ... }
```

it cannot be invoked with an instance of `DefaultList<Circle>`. or `DefaultList<Rectangle>`.

This paper starts with a review of generics specification and instantiation and then looks closer into the problem of generic instantiation and subtyping. It proceeds to introduce both versions of Java wildcards, unbounded and bounded; in a tutorial form the paper shows how to best use wildcards in the specification of methods for parametric types. It closes with the use of wildcard capture to reduce method specification pollution when the use of genericity is not deemed completely necessary. For the purpose of discussion throughout this paper we will use the generic interface `List<T>` [3] and two implementing classes `DefaultList<T>` backed a `Vector` to maintain the elements, and `LinkedList<T>`. We also assume interface `Figure` that abstracts operations such as `area`, `perimeter`, etc. and two subclasses `Circle` and `Rectangle`.

GENERIC TYPE SPECIFICATION

We start with a review of the basic syntax and semantics of Java generic for types. A generic type definition has at least one type parameter; each type parameter is defined using a *type variable* which is a simple unqualified identifier standing for a *reference* type. As example consider a list type, with operations such as `add`, `remove`, `contains`; we can define it generically by abstracting over the type of its entries; the type parameter variable appears within angular brackets:

```
public interface List<T> {  ... } // usual list operations found there.
```

We can provide two implementations of this generic interface via two generic classes: `DefaultList<T>` using a `java.util.Vector` and `LinkedList<T>`. See [3] for details on these implementations. We must note that parameter type T not only stands for a reference type, but the operations assumed to have are only the ones inherited from `Object`.

If we need to abstract over types that must have operations above those from `Object`, Java allows a constraint or *bound* on the parameter type. As an example consider the specification of a generic container class whose elements must be maintained in order; this will require the elements of the container to be `Comparable`; thus we must constraint the type parameter:

```
public class OrderedList < T extends java.lang.Comparable<T>> {...}
```

Here `OrderedList` is as a generic class with parameter T with a constraint: T must implement the `Comparable` interface. The `Comparable` interface is a built-in generic interface in `java.lang` with one method that returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object:

```
public interface Comparable<T> {public int compareTo(T other);}
```

We invite the reader to consult [1] for a complete specification of the constraints for a generic type parameter.

A generic type is *instantiated* by providing one reference type argument per type parameter. The type argument list is a comma separated list that is delimited by angle brackets following the name of the generic type being instantiated. The type arguments can be any reference type and instantiation produces a reference type called a *parameterized type*. `DefaultList<Integer>` and `DefaultList<String>` are examples of instantiations of `DefaultList<T>`. Note that type arguments for instantiations cannot be primitive types as T stands for reference types only.

Due to the presence of generic types, the types available for a student and a programmer now include the usual non-generic types as well as parameterized types resulting from the instantiations of generic types.

Java 1.5 also provides a mechanism to specify generic methods; a generic method specifies one or more type variables which appear in a list in angle brackets before the return type of the method:

```
public static <T> void concatenate(List<T> list1, List<T> list2);
public static <T> void swap(List<T> list, int i, int j);
```

T may also have constraints as specified above for generic types. Java infers the argument type in an invocation of a generic method, hence no special syntax is required:

```
List<Integer> l1 = new List<Integer>();  // proceed to fill l1.
List<Integer> l2 = new List<Integer>(); // proceed to fill l2.
concatenate(l1, l2);
swap(l1, 0, list.size()-1);
```

In this examples, Java will infer `Integer` as the type for the parameter T in both cases.

Generic Instantiation and Subtyping

The following questions must be answered when wanting to give values to a parameterized class reference:

- `DefaultList` is subclass of `List`, is `DefaultList<Circle>` a subclass of `List<Circle>`?

In other words, when declaring `List<Circle> circles;` we need to know what to use to instantiate it.

- `Circle` a subclass of `Figure`, is `DefaultList<Circle>` a subclass of `DefaultList<Figure>`? In other words, can `DefaultList<Circle>` be used as an instance of `DefaultList<Figure>`?

These answers must also need to be known when matching method parameters specified via parameterized types.

Clearly instantiations of generic types have supertype-subtype relationships with instantiations of generic subtypes provided they all have the *exact* same type arguments. Thus, after instantiation of the type hierarchy on the left of figure 1, we get

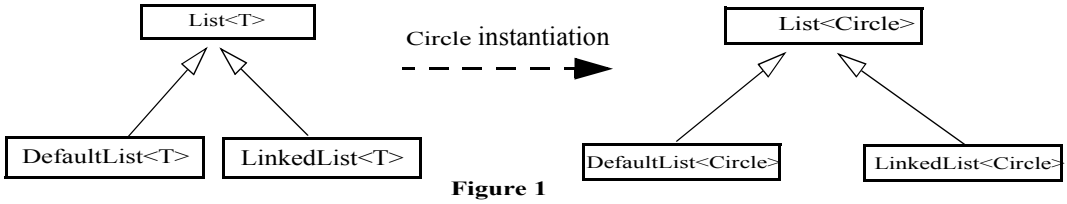

Figure 1

Type relationships to other concrete instantiations do not exist. In particular, the supertype-subtype relationship among the *type arguments* does not extend to the parameterized types resulting from instantiations. The class hierarchy in figure 2 does not yield to a class hierarchy after instantiation of `List<T>`.

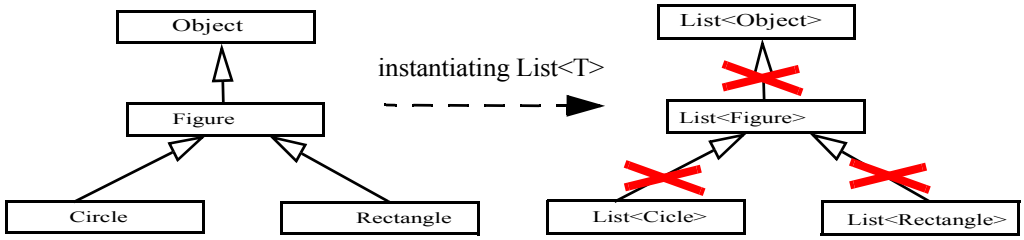

Figure 2

For example, although `Figure` is a supertype of `Circle`, `List<Figure>` is not a supertype of `List<Circle>`. Our intuition will lead us stray as the relation is only true if the resulting types were *immutable*. But the `add` method in `List<T>` shows the problem. Assume we have the following method using a parametric `List`:

```
public static void func(List<B> col, B el){col.add(el); }
```

If we were to assume that `List<A>` is a subtype of `List<B>` when `A` is a subtype of `B`, we could write:

```
List<A> myA = new List<A>();  // proceed to fill in myA.
func(myA, new B());
```

By LSP since `myA` is assumed to be a subtype of `List<B>` the invocation would be legal, which in turn would result in adding an instance of `B` to a collection of `A`'s with no runtime errors until we access `List<A>` values later on. Hence the assumption is false, and Java will flag it as a compile-time error. Thus we need to learn how to deal with subtypes on the face of generic type instantiations and the resulting parametric types.

Wildcard types

The failure of the maintenance of the subtype relation among type argument in the resulting instantiations of generic types will lead us to write code that is not as general as we would like; for instance the method:

```
public double totalArea(List<Figure> figures) {
   double sum = 0;
   for (int i = 0; i < figures.size(); i++)
      sum = sum + figures.get(i).area();
   return sum;
}
```

allows invocations only with `DefaultList<Figure>` or subclasses of it. It cannot be matched with `DefaultList<Circle>` or `DefaultList<Rectangle>`. The obvious question is then, is there a supertype to all the generic instantiations of `List<T>` besides the class `Object`? The answer is in the affirmative using *wildcards*. (We will come back later to specify `totalArea` in a satisfying way when we complete our discussion of wildcards.)

Wildcards are an extension to the type system intended to improve the flexibility of the use of generic types. Syntactically, a wildcard is an expression of the form `?`, called a wildcard type, that can be used to instantiate generic interfaces or classes. "`?`" stands for an unknown type. Parameterized wildcards types can be used as arguments or return types in method signatures. The goal and effect is that the method accepts arguments of a larger set of types, namely all types that belong to the type family denoted by the wildcard. `List<?>` ("List of unknown") an instantiation of `List<T>` is a list whose element type matches anything, and it is the supertype of all generic instantiations of `List<T>`. We can, for example, write the general method:

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

which can be invoked with any parameterized type resulting from `List<T>`. It must be noted that in the body of `printCollection()`, although we won't know the actual type of the entries, we can still read each element and assign them to a variable of type `Object`. This is always safe, since whatever the actual type of the list may be it does contain objects. However it isn't safe to add arbitrary objects to a `List<?>`:

```
List<?> c = new DefaultList<String>();
c.add(new Object()); // compile error
```

The first line is legal, as we said that `List<?>` is a supertype to all instantiations of `List<T>`. But since we don't know the actual type of the `c` entries, we cannot add objects to it, not even instances of `Object` because `add()` takes arguments of

type T, the element type of the list; but when the actual type argument is ?, it stands for some unknown type. Any parameter we pass to add would have to be a subtype of this unknown type. Since we don't know what type that is, we cannot pass anything in. The sole exception is null, which is a member of every type.

Specifying methods via parameterized types with ?, as in the print method above, is appropriate when the method is applicable to any type, and that type poses no constraints to any other method parameters. When matching a wildcard parameter the compiler creates an anonymous type variable, referred as the *"capture of ?",* that represents the type that matched the wildcard. The compiler uses the capture internally for evaluation of expressions. The term "capture of ?" occasionally shows up in compiler's error messages, as it will for the compilation error message produced by `c.add(new Object())` seen above.

Unbounded and Bounded Wildcards

An unbounded wildcard is simply the "?" wildcard introduced above. It stands for the family of all types and can be used as argument for instantiations of generic types. As we've seen above, it is useful in situations where no knowledge about the type argument of a parameterized type is needed, and where a specific type (such as `Object`) would restrict its generality.

A bounded wildcard is a wildcard with either an *upper* or a *lower* bound. A wildcard with an upper bound is written via the syntax `" ? extends Type "` and stands for the family of all types that are subtypes of `Type` , `Type` itself included. `Type` is called the upper bound. A wildcard with a lower bound is written via the syntax `" ? super Type "` and stands for the family of all types that are supertypes of `Type` , `Type` itself included. `Type` is called the lower bound.

Bounded wildcards are useful for specifications of constraints of parameterized types as well as to deal with subtypes of the type argument to match a type parameter in a generic type. As an example, with these two kinds of bounded wildcards we can form two types of parameterized `List` types:

i. `List<? extends Figure>` : a list whose entries can be any subtype of `Figure`.

ii. `List<? super Figure>` : a list whose entries can be any supertype of `Figure`.

We can now re-write the method `totalArea`, given in the introduction, in a way that is fully satisfying and general. We will use the parameterized type `List<? extends Figure>` as a type parameter; and as such it will indicate that matching values must be parameterized `List` classes resulting from subtypes of `Figure`.

```
public static double totalArea(List<? extends Figure> list){
    // same code as above
}
```

We can now successfully match the argument list with `List<Circle>` or `List<Rectangle>`.

It must be noted in passing that a wildcard parameterized type (A<?>, A<? **extends** B>, A<? **super** B>) is not a type in the regular sense (different from a non-parameterized class/interface). You cannot make instances of it. Wildcard parameterized types can be used for typing parameters and return types of methods, type of a field or local variable, array component, type argument of other parameterized types, and as target in a type cast. It cannot be used for creation of arrays, in `instanceof` expressions (except unbounded wildcards), or as supertypes for inheritance.

Specifying lower bound constraints using bounded wildcards

As we saw earlier in the definition of `OrderedList`, it is often necessary to define generic types or generic methods with a bound specified as `<T extends Comparable<T>>.` Thus type arguments for T must be types which implement the bound as well. But then we cannot instantiate `T` with a *subtype* of a type that implements the bound! This is because T itself would not implement the bound, only its supertype does. Let's clear this with an example.

Assume class `Person` implements `Comparable<Person>`, and `Student` is a subclass of `Person`:

```
class Person implements Comparable<Person>  {
   ...
     public int compareTo (Person other) { ... }
}
class Student extends Person {...}
```

`Student` is a subtype of `Person` by inheritance; but `Student` does not, and in this case, cannot implement `Comparable<Student>`; it cannot because it would be a subtype of two different instantiations of the same generic type `Comparable`, (the other one being the inherited one from `Person`) and that is illegal in Java 1.5 [1].

Consider now a generic method to sort a list of objects. Since the objects must be `Comparable`, we would need to require that the list elements must be comparable to one another; thus we arrive at the following specification for the sort:

```
public static <T extends Comparable<T>> void sort(List<T> list){...}
```

Now, we can invoke `sort` on `List<Person>`, as `Person` implements `Comparable<Person>`, bu we cannot invoke it on `List<Student>`, because `Student` is not `Comparable<Student>` although is `Comparable<Person>`. Hence, it turns out that it is too strong to require that `T` implements `Comparable<T>`. In order to make `sort` applicable to subtypes

of `Comparable,` we need to be able to specify that T either implements `Comparable` or a superclass does. We can then proceed to specify it via a wildcard with a lower bound: `Comparable < ? super T >.` Thus sort specification becomes

```
public static <T extends Comparable < ? super T > > void sort(List<T> list) {...}
```

and we can now invoke `sort` on `List<Student>` as `Student` is a subtype of `Comparable.` Likewise a better specification for the `OrderedList` seen earlier will be :

```
public class OrderedList < T extends java.lang.Comparable<? super T>> {...}
```

As another example of judicious use of upper bounds and lower bounds for wildcards, consider the method to copy a `src` list into a `dest` list. An initial specification could be:

```
public static <T> void copy(List<T> dest, List<T> src);
```

This specification is correct but it is more restrictive than need be; it requires both input and output collections to be lists with the exact same entry type. Hence the following perfectly sensible invocation would yield an error message:

```
List<Figure> output = new DefaultList<Figure>();
List<Circle>  input = new DefaultList<Circle>();
  ...
Collections.copy(output,input);  // error: illegal argument types
```

As specified both list must be of type `List<Figure>` or of type `List<Circle>` .

We could try to relax the method's requirements and declare it using unbounded wildcard parameterized types:

```
public static void copy( List<?> dest, List<?> src);
```

But its implementation will not compile! The problem is that `List<?>`'s `get()` used on `src` entries returns a reference pointing to an object of unknown type. On the other hand, `List<?>`'s `set()` used on `dest` entries requires something of type unknown, given by the argument matching the second wildcard card capture. Hence the compiler issues an error message: `get()` returns `Object` and `set()` expects a more specific, yet unknown type. As written, the signature specifies that it takes one type of list as a source and copies the content into another - totally unrelated - destination list. Conceptually it would allow things like copying a list of apples into a list of oranges, a red herring. (Note each occurrence of a wildcard stands for a potentially different type). We need a specification that allows copying elements from a source list into a destination list; thus, if the source type is T, the destination type can be T or any supertype of T so that we can perform assignment from `src` to `dest` as we can assign subtypes to supertypes. Now, the source type could be T, as well as any subtype of T, hence:

```
public static <T> void copy(List<? super T> dest,List<? extends T> src) ;
```

This specification makes the above invocation correct; but the compiler needs to do a bit of extra checks. The specification requires that a type T must exist that is subtype of `Figure`, the `dest` element type, and supertype of `Circle`, the `src` element type. In the above case the compiler successfully finds T = `Figure.` While for the invocation below:

```
List<String> output = new DefaultList< String >();
List<Long>    input = new DefaultList< Long >();
Collections.copy(output,input);  // compilation error
```

the compiler finds no type that is subtype of `String` and supertype of `Long`, flagging the invocation as incorrect.

Bounded wildcards yields a very flexible mechanism to work around the fact that subclass parameters do not yield subclass parameterized types. Specifically, specifying a parameter to a method via a parameterized type with upper bounds will allows us to invoke it via any subclass of the bounded type. Likewise, a parameterized type with lower bounds will allow invocations with superclasses of the bounded type; while unbounded wildcards can be matched with any type.

Other Examples to Specify methods for generality

We include two more examples of the judicious use of wildcards that allow us to specify method interfaces that are flexible.

1. Consider the specification of a static method to do binary search on a list using a `java.lang.Comparator`:

```
public static <T> int binarySearch (List<T> list, T key, Comparator<T> comp);
```

Although correct, this specification is too restrictive in two counts:

a. It requires `List` entries and `key` to be of same type; but `list` entries could be from a subclass of T as all that is required is to compare them against `key`. Specifying list of type `List<? extends T>` fulfills and generalizes that requirement. For containers whose data is meant to be **read-only** the use of a wildcard upper bound gives the best specification.

b. The `Comparator` must compare T values. Again, the type of entries passed to the `List` may not directly implement the `Comparator` but can inherit it from a supertype. As we seen earlier, a more flexible specification for the comparator is `Comparator<? super T>`. With this we attain a more general specification of `binarySearch`

```
public static <T> int binarySearch (List<? extends T> list, T key,
                                    Comparator<? super T> comp);
```

2. When the requirement is to specify generic methods where the container is to be written to, again consider bounds for flexibility. Consider specifying a method to fill a list with copies of one item; we could start with the following specification:

```
public static <T> void fill (List<T> list, T obj);
```

Again, although fine it's restrictive. Thinking of the most general relationship between the types of the arguments `obj` and `list` entries, since we will be assigning to the components `list` entry type cannot be a subtype of T, but it could be a supertype as we can assign a subtype reference to a supertype reference. Hence, we can specify `fill` more flexibly as:

```
public static <T> void fill(List<? super T> list, T obj);
```

This is a more general solution; the `List` entries can be instances of a super type of T, the type of the object being used to initialize the list. In general for **writing** a container the use of a wildcard lower bound is the most flexible specification.

In closing, note that `java.util.Collections` gives very good examples of the use of these three wildcard kinds.

Use of Wildcard Capture

As mentioned earlier, although the use of parameterized wildcards in parameter specification of methods provides a very flexible way to specify methods, it comes with a cost nonetheless. The values of references to parameterized type using the wildcard can only be read in the implementation of the methods. If the body of the method using a wildcard parameter requires more than reading the actual argument, we can use the wildcard capture to invoke a private method with non-wildcard type parameters to get full argument access. For illustration purposes, let's specify and implement a method to reverse the order of a list's entries. Since reverse is applicable regardless of the list entry type, we can write the following specification:

```
public static void reverse(List<?> list);
```

The implementation needs to know the actual type matched with "?", to use the list's `set` method; but wildcard parameters do not provide such access. Using "capture of ?", we invoke a *private* method using type parameters to get full list's access:

```
public static void reverse(List<?> list) {
  revHelper(list);  // uses wildcard capture to pass the type matched with ?
}
private static <T> void revHelper(List<T> list) {
  int j;
  for (int i = 0, mid = list.size() >> 1; i < mid; i++) {
   T temp = list.get(i);
   j = list.size()-1-i;
   list.set(i, list.get(j));
   list.set(j, temp);
  }
}
```

Private methods can be employed as above to "open up" type arguments of types with wildcards avoiding that implementation details such as the need for explicit type arguments influence the public signatures of a class or interface declaration (interface pollution). In closing it must be noted that use of wildcard capture in this fashion can only occur for methods with only one wildcard parameter type as programmatically we cannot distinguish between type captures for more than one wildcard.

CONCLUSION

The use of generic types in Java is not as straightforward as one would expect on a first look. As we seen, generic type instantiation does not preserve the subtype relation of instantiation types. Further, the use of parameterized types in the specification of methods may turn out to be incorrect or too restrictive. The addition of unbounded and bounded wildcards to the type system gives us a set of choices that allow us to provide a more general specification of method interfaces; and although generic type instantiations do not preserve subtyping wildcards provide us with a mechanisms to overcomes this limitation.

It is a very good and illustrative exercise, for both instructors new to Java generics and as well as for students, to start with the 1.4 or earlier specification of `java.util.Collections` and proceed to re-engineer the specification for generics stepwise while comparing results with a java 1.5 or later version of that library. With the exception of a couple of methods for which legacy issues needs to be addressed with particular wildcard bounds, you should be able to arrive to the new API.

The ultimate purpose of this paper was to provide some light to instructors on a couple of fundamental issues in the use of the Java generic mechanism; due to space the interested reader is invited to consult [1,2] for other issues in the use of Java generics with the same level of importance.

1. REFERENCES

[1] Gosling J., Joy B., Steel G., Bracha G. *The JavaLanguage Specification*. Third Edition. Addison-Wesley.Budd. 2005.

[2] Naftalin, M., Wadler, P. *Java Generics and Collections.* O'Reilly, 2007.

[3] J.Niño, F. Hosch. *An introduction to programming and Object Oriented design using Java 1.5*. Wiley 2005. 2nd edition.

[4] Sun Corporation. *Java API* for version 1.5 or later.