Java theory and practice: Going wild with generics, Part 2

The get-put principle

Brian Goetz July 01, 2008

Wildcards can be very confusing when it comes to generics in the Java™ language, and one of the most common mistakes is to fail to use one of the two forms of bounded wildcards ("`? super T`" and "`? extends T`") when needed. You've made this mistake? Don't feel bad, even the experts have, and this month Brian Goetz shows you how to avoid it.

View more content in this series

In the Java language, arrays are covariant (because an `Integer` is also `Number`, an array of `Integer` is also an array of `Number`), but generics are not (a `List<Integer>` is *not* a `List<Number>`.) People can argue over which choice was "right" and which was "wrong" — of course, both have pros and cons — but there is no question that having two similar mechanisms for constructing derived types with subtly different semantics is a substantial source of confusion and mistakes.

Bounded wildcards (those funny "`? extends T`" generic type specifiers) are one of the tools the language provides for dealing with the lack of covariance — they let classes declare when method arguments or return values are covariant (or the opposite, *contravariant*). While knowing when to use bounded wildcards is one of the more complicated aspects of generics, the burden of using them falls mostly on library writers, rather than library users. The most common mistake with bounded wildcards is forgetting to use them at all, restricting the utility of a class or forcing users to jump through hoops to reuse an existing class.

The need for bounded wildcards

Let's start with a simple generic class, a value container called `Box`, which holds a value of known type:

```
public interface Box<T> {
    public T get();
    public void put(T element);
}
```

Because generics are not covariant, a `Box<Integer>` is not a `Box<Number>`, even though an `Integer` is a `Number`. But for simple generic classes like `Box`, this is not a problem, and in fact we might not even notice, because the interface to `Box<T>` is specified entirely in terms of variables of type T — and not types generified over T. Dealing directly in terms of type variables allows the polymorphism we want to come along for free. Listing 1 shows two examples of this sort of polymorphism: retrieving the contents of a `Box<Integer>` as a `Number` and putting an `Integer` into a `Box<Number>`:

Listing 1. Exploiting inherent polymorphism with generic classes

```
Box<Integer> iBox = new BoxImpl<Integer>(3);
Number num = iBox.get();

Box<Number> nBox = new BoxImpl<Number>(3.2);
Integer i = 3;
nBox.put(i);
```

Our experience with this simple `Box` class could convince us that covariance isn't needed, because in the places where we expect polymorphism, the data is already in a form where the compiler is able to apply the appropriate subtyping rules.

Things get more complicated, however, when we want the API to deal not only in variables of type T, but in types generified over T. Let's say we want to add a new method to `Box` that allows us to fetch the contents from another `Box` and put it in this one, as shown in Listing 2:

Listing 2. An extended Box interface, which is not as flexible as it looks

```
public interface Box<T> {
    public T get();
    public void put(T element);
    public void put(Box<T> box);
}
```

The problem with this extended `Box` is that we can only put the contents in a `Box` whose type parameter is exactly the same as the receiving box. So, for example, the code in Listing 3 won't compile:

Listing 3. Generics are not covariant

```
Box<Number> nBox = new BoxImpl<Number>();
Box<Integer> iBox = new BoxImpl<Integer>();

nBox.put(iBox);     // ERROR
```

We get an error message that tells us it cannot find the method `put(Box<Integer>)` on a `Box<Number>`. This error makes sense when we consider that generics are not covariant; a `Box<Integer>` is not a `Box<Number>`, even though an `Integer` is a `Number`, but this makes the `Box` class feel less "generic" than we'd hoped it would be. To increase the usefulness of our generic code, instead of specifying the exact type of a generic type parameter, we can specify an upper (or lower) bound instead. To do so, we use a bounded wildcard, which takes the form "`? extends T`" or "`? super T`". (Bounded wildcards can only be used as type parameters; they cannot appear as types themselves — for that, a bounded named type variable is required.) In Listing 4, we change

the signature of `put()` to use an upper-bounded wildcard —`Box<? extends T>`, which means that the type parameter of the `Box` can be `T` or any subclass of `T`.

Listing 4. Improved version of Box class from Listing 3, which accounts for covariance

```
public interface Box<T> {
    public T get();
    public void put(T element);
    public void put(Box<? extends T> box);
}
```

Now the code in Listing 3 will compile and do what we want, because we've said that the parameter to `put()` can be a `Box` whose type parameter is T or any of its subtypes. Because `Integer` is a subtype of `Number`, the compiler is able to resolve the method reference `put(Box<Integer>)` because `Box<Integer>` matches the bounded wildcard `Box<? extends Number>`.

The "mistake" in the early version of `Box` is an easy one to fall into. Even the experts make this mistake — you can find numerous places in the platform class libraries where a `Collection<T>` is used instead of a `Collection<? extends T>`. For example, in `AbstractExecutorService` in the java.util.concurrent package, the argument to `invokeAll()` was originally a `Collection<Callable<T>>`. This made using `invokeAll()` fairly cumbersome, though, as this required that the set of tasks be held by a collection parameterized exactly by `Callable<T>`, and not by a collection parameterized by some class that implements `Callable<T>`. In Java 6, this signature was changed to `Collection<? extends Callable<T>>`— but to illustrate how easy it is to make this mistake, the correct fix would have been to make `invokeAll()` take an argument of `Collection<? extends Callable<? extends T>>`. The latter is definitely uglier but has the benefit of not boxing in your clients.

Lower-bounded wildcards

Most bounded wildcards are bounded above; the "`? extends T`" notation places an upper bound on the type. It is also possible, though less common, to place a lower bound on the type with the notation "`? super T`", meaning "T or any superclass of T". Lower-bounded wildcards show up when you want to specify a callback object, such as a comparator, or a data structure into which you are going to place a value.

Let's say we want to enhance our `Box` with the ability to compare the contents with that of another box. We can extend `Box` with a `containsSame()` method, and the definition of a `Comparator` callback object, as shown in Listing 5:

Listing 5. Too-restrictive an attempt at adding a comparison method to Box

```
public interface Box<T> {
    public T get();
    public void put(T element);
    public void put(Box<? extends T> box);

    boolean containsSame(Box<? extends T> other,
                         EqualityComparator<T> comparator);

    public interface EqualityComparator<T> {
        public boolean compare(T first, T second);
    }
}
```

We remembered to define the type of the other box in `containsSame()` using a wildcard, which avoids the problems we've seen before. But we've still got a similar problem; the comparator parameter must be exactly an `EqualityComparator<T>`. This means we couldn't write the code in Listing 6:

Listing 6. Failure when attempting to use the comparison method from Listing 5

```
public static EqualityComparator<Object> sameObject
    = new EqualityComparator<Object>() {
        public boolean compare(Object o1, Object o2) {
            return o1 == o2;
        }
};

...

BoxImpl<Integer> iBox = ...;
BoxImpl<Number> nBox = ...;

boolean b = nBox.containsSame(iBox, sameObject);
```

Using an `EqualityComparator<Object>` here seems a perfectly reasonable thing to want to do. Why should clients have to create a separate comparator for every possible type of `Box` when they can specify it generically? The solution is to use a lower-bounded wildcard — represented by "`? super T`". The correct version of the `Box` class, as extended with a `compareTo()` method, is shown in Listing 7:

Listing 7. More flexible version of the comparison operation from Listing 5, using bounded wildcards

```
public interface Box<T> {
    public T get();
    public void put(T element);
    public void put(Box<? extends T> box);

    boolean containsSame(Box<? extends T> other,
                         EqualityComparator<? super T> comparator);

    public interface EqualityComparator<T> {
        public boolean compare(T first, T second);
    }
}
```

By using a lower-bounded wildcard, the `containsSame()` method is saying that it requires something that can compare a T *or any of its supertypes*, allowing us to provide a comparator that knows how to compare Objects without having to wrap it with an `EqualityComparator<Number>`.

The get-put principle

There's an old joke that says "A man with one watch always knows what time it is; a man with two watches is never sure." Because the language supports both upper- and lower-bounded wildcards, how do we know which one to use, and when?

There's a simple rule, called the *get-put principle*, which tells us which kind of wildcard to use. The get-put principle, as stated in Naftalin and Wadler's fine book on generics, *Java Generics and Collections* (see Resources), says:

> Use an `extends` wildcard when you only get values out of a structure, use a `super` wildcard when you only put values into a structure, and don't use a wildcard when you do both.

The get-put principle is easiest to understand when applied to container classes like `Box` or the Collections classes, because the notion of getting or putting connects naturally with what these classes do: store things. So, if we wanted to apply the get-put principle to create a method that copies from one `Box` to another, the most general form would be as shown in Listing 8, where an upper-bounded wildcard is used for the source and a lower-bounded wildcard is used for the destination:

Listing 8. Copy method for Box using both upper-bounded and lower-bounded wildcards

```
public static<T> void copy(Box<? extends T> from, Box<? super T> to) {
    to.put(from.get());
}
```

How do we apply the get-put principle in the case of the `containsSame()` method shown earlier, where we used an upper-bounded wildcard for the box but a lower-bounded wildcard for the comparator? The first part is easy: we are getting a value from the other box, so we need to use an `extends` wildcard. But the second part is not as clear — because the comparator isn't a container, so it doesn't feel like we're either getting or putting from a data structure.

The way to think about the get-put principle when the data type is not obviously a container class like a collection is that, even though an `EqualityComparator` is not a data structure, it is still something you can "put" a value into — in the sense of passing the value to one of its methods. In the `containsSame()` method, you are using the `Box` as a producer of values (retrieving values from the `Box`) and using the comparator as a consumer of values (passing values to the comparator). So it makes sense to use an `extends` wildcard for the `Box`, but a `super` wildcard for the comparator.

We can see the get-put principle at work in the declaration for `Collections.sort()`, shown in Listing 9:

Listing 9. Another example of using lower-bounded wildcards

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

Here, we are saying we can sort a `List` that is parameterized by any type that implements `Comparable`. But rather than restricting the domain of `sort()` to lists whose elements are comparable to themselves, we can go further — we can sort lists of elements that know how to compare themselves to their supertypes, too. Because we are putting values into the comparator to determine the relative ordering of two elements, the get-put principle tells us we want to use a super wildcard here.

The seeming circular reference — where `T` extends something parameterized by `T`— is not really circular at all. It is simply expressing the constraint that to be able to sort a `List<T>`, `T` has to implement the interface `Comparable<X>`, where `X` is `T` or one of its supertypes.

The last part of the rule — don't use a wildcard when you are both getting and putting — follows from the first two parts. If you can put a `T` or any of its subtypes, and you can get a `T` or any of its supertypes, then the only thing you can both get and put is a `T` itself.

Keep bounded wildcards out of return values

It is sometimes tempting to use a bounded wildcard in the return type of a method. But this temptation is best avoided because returning bounded wildcards tends to "pollute" client code. If a method were to return a `Box<? extends T>`, then the type of the variable receiving the return value would have to be `Box<? extends T>`, which pushes the burden of dealing with bounded wildcards on your callers. Bounded wildcards work best when they are used in APIs, not in client code.

Summary

Bounded wildcards are extremely useful in making generic APIs more flexible. The biggest impediment to using bounded wildcards correctly is the perception that we don't need to use them! Some situations will call for lower-bounded wildcards, and some for upper-bounded ones, and the get-put principle can be used to determine which should be used.

Related topics

- JSR 14: Added generics to the Java programming language. The early specification was derived from GJ. Wildcards were added later.
- Generics FAQ: Angelika Langer has created a comprehensive FAQ on generics.
- "*Java theory and practice*: Going wild with generics" (Brian Goetz, developerWorks, May 2008): Deals with another tricky aspect of wildcards: wildcard capture.
- "*Java theory and practice*: Generics gotchas" (Brian Goetz, developerWorks, January 2005): Explains some of the consequences of the erasure approach to implementing generics.
- "Introduction to generic types in JDK 5.0" (Brian Goetz, developerWorks, December 2004): Introduces generic types, which let you define classes with abstract type parameters that you specify at instantiation time.
- *Java Concurrency in Practice*: The how-to manual for developing concurrent programs in Java code, including constructing and composing thread-safe classes and programs, avoiding liveness hazards, managing performance, and testing concurrent applications.