Java theory and practice: Going wild with generics, Part 1

Understanding wildcard capture

Brian Goetz (brian.goetz@oracle.com)
Senior Staff Engineer
Sun Microsystems

06 May 2008

One of the most complicated aspects of generics in the Java™ language is wildcards, and in particular, the treatment and confusing error messages surrounding *wildcard capture*. In this installment of *Java theory and practice*, veteran Java developer Brian Goetz deciphers some of the weirder-looking error messages emitted by javac and offers some tricks and workarounds that can simplify using generics.

View more content in this series

Generics have been a controversial topic ever since they were added to the language in JDK 5. Some say they simplify programming by extending the reach of the type system and therefore the compiler's ability to verify type safety; others say that they add more complexity than they're worth. We've all had a few head-scratching moments with generics, but far and away the trickiest part of generics is wildcards.

Wildcard basics

Generics are a means of expressing type constraints on the behavior of a class or method in terms of unknown types, such as "whatever the types of parameters `x` and `y` of this method are, they must be the same type," "you must provide a parameter of the same type to both of these methods," or "the return value of `foo()` is the same type as the parameter of `bar()`."

Wildcards — the funky question marks where a type parameter should go — are a means of expressing a type constraint in terms of an unknown type. They were not part of the original design for generics (derived from the Generic Java (GJ) project); they were added as the design process played out over the five years between the formation of JSR 14 and its final release.

Wildcards play an important role in the type system; they provide a useful type bound for the family of types specified by a generic class. For the generic class `ArrayList`, the type `ArrayList<?>` is a supertype of `ArrayList<T>` for any (reference) type `T` (as are the raw type `ArrayList` and the root type `Object`, but these supertypes are far less useful for performing type inference).

The wildcard type `List<?>` is different from both the raw type `List` and the concrete type `List<Object>`. To say a variable `x` has type `List<?>` means that there exists some type `T` for which `x` is of type `List<T>`, that `x` is homogeneous even though we don't know what particular type its elements have. It's not that the contents can be anything, it's that we don't know what the type constraints on the contents are — but we know that there *is* a constraint. On the other hand, the raw type `List` is heterogeneous; we are not able to place any type constraints on its elements, and the concrete type `List<Object>` means that we explicitly know that it can contain any object. (Of course, the generic type system has no concept of "the contents of a list," but it is easiest to understand generics in terms of collection types like `List`.)

The utility of wildcards in the type system comes partially from the fact that generic types are not covariant. Arrays are covariant; because `Integer` is a subtype of `Number`, the array type `Integer[]` is a subtype of `Number[]`, and therefore an `Integer[]` value can be supplied wherever a value of `Number[]` is required. On the other hand, generics are not covariant; `List<Integer>` is not a subtype of `List<Number>`, and attempting to supply a `List<Integer>` where a `List<Number>` is demanded is a type error. This was not an accident — nor necessarily the error that everyone assumes it to be — but the different behavior of generics vs. arrays does cause a great deal of confusion.

I got dealt a wildcard — now what?

Listing 1 shows a simple container type, `Box`, which supports `put` and `get` operations. `Box` is parameterized by a type parameter `T`, which signifies the type of the contents of the box; a `Box<String>` can contain only elements of type `String`.

Listing 1. Simple generic Box type

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

One benefit of wildcards is that they allow you to write code that can operate on variables of generic types without knowing their exact type bound. For example, suppose you have a variable of type `Box<?>`, such as the `box` parameter in the method `unbox()` in Listing 2. What can `unbox()` do with the box that it's been handed?

Listing 2. Unbox method with wildcard parameter

```
public void unbox(Box<?> box) {
    System.out.println(box.get());
}
```

It turns out it can do quite a lot: it can call the `get()` method, and it can call any of the methods inherited from `Object` (such as `hashCode()`). The only thing it cannot do is call the `put()` method, and this is because it cannot verify the safety of such an operation without knowing the type parameter `T` for this `Box` instance. Because `box` is a `Box<?>`, and not a raw `Box`, the compiler knows that there is some `T` that serves as a type parameter for `box`, but because it doesn't know what that `T` is, it will not let you call `put()` because it cannot verify that doing so will not violate the type safety constraints for `Box`. (Actually, you can call `put()` in one special case: when you pass the `null` literal. We may not know what type `T` represents, but we know that the `null` literal is a valid value for any reference type.)

What does `unbox()` know about the return type of `box.get()`? It knows that it is `T` for some unknown `T`, so the best it can do is conclude that the return type of `get()` is the erasure of the unknown type `T`, which in the case of an unbounded wildcard is `Object`. So the expression `box.get()` in Listing 2 has type `Object`.

Wildcard capture

Listing 3 shows some code that seems like it *should* work, but doesn't. It takes a generic `Box`, extracts the value, and tries to put the value back into the same `Box`.

Listing 3. Once you take it out of the box, you can't put it back

```
public void rebox(Box<?> box) {
    box.put(box.get());
}

Rebox.java:8: put(capture#337 of ?) in Box<capture#337 of ?> cannot be applied
   to (java.lang.Object)
    box.put(box.get());
       ^
1 error
```

This code looks like it should work because the value that came out is the right type to go back in, but instead the compiler produces the (very confusing) error message about "capture#337 of ?" not being compatible with `Object`.

What on earth does "capture#337 of ?" mean? When the compiler encounters a variable with a wildcard in its type, such as the `box` parameter of `rebox()`, it knows that there must have been some `T` for which `box` is a `Box<T>`. It does not know what type `T` represents, but it can create a placeholder for that type to refer to the type that `T` must be. That placeholder is called the *capture* of that particular wildcard. In this case, the compiler has assigned the name "capture#337 of ?" to the wildcard in the type of `box`. Each occurrence of a wildcard in each variable declaration gets a different capture, so in the generic declaration `foo(Pair<?,?> x, Pair<?,?> y)`, the compiler would assign a different name to the capture of each of the four wildcards because there is no relationship between any of the unknown type parameters.

What this error message is telling us is that we can't call `put()` because it can't verify that the type of the actual parameter to `put()` is compatible with the type of its formal parameter — because the type of its formal parameter is unknown. In this case, because `?` essentially means "? extends Object," the compiler has already concluded that the type of `box.get()` is `Object`, not "capture#337 of ?," and it can't statically verify that an `Object` is an acceptable value for the type identified by the placeholder "capture#337 of ?."

Capture helpers

While it looks like the compiler has thrown away some useful information, there's a trick that we can use to get the compiler to reconstitute that information and go along with us here, which is to give a name to the unknown wildcard type. Listing 4 shows an implementation of `rebox()`, along with a generic helper method, that does the trick:

Listing 4. The "capture helper" idiom

```
public void rebox(Box<?> box) {
    reboxHelper(box);
}

private<V> void reboxHelper(Box<V> box) {
    box.put(box.get());
}
```

The helper method `reboxHelper()` is a *generic method*; generic methods introduce additional type parameters (placed in angle brackets before the return type), which are usually used to formulate type constraints between the parameters and/or return value of the method. In the case of `reboxHelper()`, however, the generic method does not use the type parameter to specify a type constraint; it allows the compiler (through type inference) to give a name to the type parameter of box's type.

The capture helper trick allows us to work around the compiler's limitations in dealing with wildcards. When `rebox()` calls `reboxHelper()`, it knows that doing so is safe because its own `box` parameter must be a `Box<T>` for some unknown `T`. Because the type parameter `V` is introduced in the method signature and not tied to any other type parameter, it can stand for any unknown type as well, so a `Box<T>` for some unknown `T` might as well be a `Box<V>` for some unknown `V`. (This is similar to the principle of alpha reduction in the lambda calculus, which allows you to rename bound variables.)Now the expression `box.get()` in `reboxHelper()` no longer has type `Object`, it has type `V`— and it is allowable to pass a `V` to `Box<V>.put()`.

We could have declared `rebox()` as a generic method in the first place, like `reboxHelper()`, but that is considered bad API design style. The governing design principle here is "don't give something a name if you're never going to refer to it by name." In the case of generic methods, if a type parameter appears only once in the method signature, then it probably should be a wildcard rather than a named type parameter. In general, APIs with wildcards are simpler than APIs with generic methods, and the proliferation of type names in a more complicated method declaration would likely make the declaration less readable. Because the name can always be resurrected with a private capture helper if needed, this approach gives you the opportunity to keep APIs clean without throwing useful information away.

Type inference

The capture-helper trick depends on several things: type inference and capture conversion. The Java compiler doesn't perform type inference in very many places, but one place it does is in inferring the type parameter for generic methods. (Other languages depend much more heavily on type inference, and we may see additional type inference features added to the Java language in the future.) You are allowed to specify the value of the type parameter if you like, but only if you can name the type — and the capture types are not denotable. So the only way this trick could work is if the compiler infers the type for you. Capture conversion is what allows the compiler to manufacture a placeholder type name for the captured wildcard, so that type inference can infer it to be that type.

The compiler will try and infer the most specific type it can for the type parameters when resolving a call to a generic method. For example, with this generic method:

```
public static<T> T identity(T arg) { return arg };
```

and this call:

```
Integer i = 3;
System.out.println(identity(i));
```

the compiler could infer that `T` is `Integer`, `Number`, Serializable, or `Object`, but it chooses `Integer` as that is the most specific type that fits the constraints.

You can use type inference to reduce some of the redundancy when constructing generic instances. For example, using our `Box` class, creating a `Box<String>` requires you to specify the type parameter `String` twice:

```
Box<String> box = new BoxImpl<String>();
```

This violation of the DRY principle (Don't Repeat Yourself) here can be irksome, even when IDEs are able to do some of the work for you. However, if the implementation class `BoxImpl` provides a generic factory method as in Listing 5 (which is a good idea anyway), you can reduce this redundancy in client code:

Listing 5. A generic factory method that allows you to avoid redundantly specifying type parameters

```
public class BoxImpl<T> implements Box<T> {

    public static<V> Box<V> make() {
        return new BoxImpl<V>();
    }

    ...
}
```

If you instantiate a `Box` using the `BoxImpl.make()` factory, you need only specify the type parameter once:

```
Box<String> myBox = BoxImpl.make();
```

The generic `make()` method returns a `Box<V>` for some type `V`, and the return value is being used in a context that requires a `Box<String>`. The compiler determines that `String` is the most specific type that `V` could take on that satisfies the type constraints, and so infers `V` to be `String` here. You still have the option of manually specifying the value of `V` as follows:

```
Box<String> myBox = BoxImpl.<String>make();
```

In addition to saving some keystrokes, the factory method technique illustrated here has other advantages over constructors: you can give them more descriptive names, they can return subtypes of the named return type, and they are not necessarily required to create a new instance for each invocation, enabling sharing of immutable instances. (See Effective Java, Item #1 in the Resources for more on the benefits of static factories.)

Conclusion

Wildcards can definitely be tricky; some of the most confusing error messages that come out of the Java compiler have to do with wildcards, and some of the most complex sections of the Java Language Specification have to do with wildcards. However, when used properly, they can be extremely powerful. The two tricks shown here — the capture helper trick and the generic factory trick — both take advantage of generic methods and type inference, which, when used properly, can hide much of the complexity.

Resources

- "Generics gotchas" (Brian Goetz, developerWorks, January 2005): Learn how to identify and avoid some of the pitfalls in learning to use generics.
- Introduction to generic types in JDK 5 (Brian Goetz, developerWorks, December 2004): Follow along with frequent developerWorks contributor and Java programming expert Brian Goetz, as he explains the motivation for adding generics to the Java language, details the syntax and semantics of generic types, and provides an introduction to using generics in your classes.
- JSR 14: Added generics to the Java programming language. The early specification was derived from GJ. Wildcards were added later.
- *Java Generics and Collections*: Offers a comprehensive treatment of generics.
- *Effective Java*: Item 1 contains further exploration of the benefits of static factory methods.
- Generics FAQ: Angelika Langer has created a comprehensive FAQ on generics.
- *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.
- Technology bookstore: Browse for books on a wide variety of technical topics.
- The Java technology zone: Hundreds of articles about every aspect of Java programming.
- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Brian Goetz

Brian Goetz has been a professional software developer for 20 years. He is a senior staff engineer at Sun Microsystems, and he serves on several JCP Expert Groups. Brian's book, *Java Concurrency In Practice*, was published in May 2006 by Addison-Wesley. See Brian's published and upcoming articles in popular industry publications.