developerWorks®

Java theory and practice: Generics gotchas

Identify and avoid some of the pitfalls in learning to use generics

Brian Goetz (brian@quiotix.com)
Principal Consultant
Quiotix

25 January 2005

Generic types, added in JDK 5.0, are a significant enhancement to type safety in the Java language. However, some aspects of generics may seem confusing, or even downright bizarre, to first-time users. In this month's *Java theory and practice*, Brian Goetz examines the common pitfalls that ensnare first-time users of generics.

View more content in this series

Generic types (or generics) bear a superficial resemblance to templates in C++, both in their syntax and in their expected use cases (such as container classes). But the similarity is only skin-deep -- generics in the Java language are implemented almost entirely in the compiler, which performs type checking and type inference, and then generates ordinary, non-generic bytecodes. This implementation technique, called *erasure* (where the compiler uses the generic type information to ensure type safety, but then erases it before generating the bytecode), has some surprising, and sometimes confusing, consequences. While generics are a big step forward for type safety in Java classes, learning to use generics will almost certainly provide some opportunity for head-scratching (and sometimes cursing) along the way.

Note: *This article assumes familiarity with the basics of generics in JDK 5.0.*

Generics are not covariant

While you might find it helpful to think of collections as being an abstraction of arrays, they have some special properties that collections do not. Arrays in the Java language are covariant -- which means that if `Integer` extends `Number` (which it does), then not only is an `Integer` also a `Number`, but an `Integer[]` is also a `Number[]`, and you are free to pass or assign an `Integer[]` where a `Number[]` is called for. (More formally, if `Number` is a supertype of `Integer`, then `Number[]` is a supertype of `Integer[]`.) You might think the same is true of generic types as well -- that `List<Number>` is a supertype of `List<Integer>`, and that you can pass a `List<Integer>` where a `List<Number>` is expected. Unfortunately, it doesn't work that way.

It turns out there's a good reason it doesn't work that way: It would break the type safety generics were supposed to provide. Imagine you could assign a `List<Integer>` to a `List<Number>`. Then the following code would allow you to put something that wasn't an `Integer` into a `List<Integer>`:

```
List<Integer> li = new ArrayList<Integer>();
List<Number> ln = li; // illegal
ln.add(new Float(3.1415));
```

Because `ln` is a `List<Number>`, adding a `Float` to it seems perfectly legal. But if `ln` were aliased with `li`, then it would break the type-safety promise implicit in the definition of `li` -- that it is a list of integers, which is why generic types cannot be covariant.

More covariance troubles

Another consequence of the fact that arrays are covariant but generics are not is that you cannot instantiate an array of a generic type (`new List<String>[3]` is illegal), unless the type argument is an unbounded wildcard (`new List<?>[3]` is legal). Let's see what would happen if you were allowed to declare arrays of generic types:

```
List<String>[] lsa = new List<String>[10]; // illegal
Object[] oa = lsa;  // OK because List<String> is a subtype of Object
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[0] = li;
String s = lsa[0].get(0);
```

The last line will throw a `ClassCastException`, because you've managed to cram a `List<Integer>` into what should have been a `List<String>`. Because array covariance would have allowed you to subvert the type safety of generics, instantiating arrays of generic types (except for types whose type arguments are unbounded wildcards, like `List<?>`) has been disallowed.

Construction delays

Because of erasure, `List<Integer>` and `List<String>` are the same class, and the compiler only generates one class when compiling `List<V>` (unlike in C++). As a result, the compiler doesn't know what type is represented by `V` when compiling the `List<V>` class, and so you can't do certain things with a type parameter (the `V` in `List<V>`) within the class definition of `List<V>` that you could do if you knew what class was being represented.

Because the runtime cannot tell a `List<String>` from a `List<Integer>` (at runtime, they're both just `List`s), constructing variables whose type is identified by a generic type parameter is problematic. This lack of type information at runtime poses a problem for generic container classes and for generic classes that want to make defensive copies.

Consider the generic class `Foo`:

```
class Foo<T> {
  public void doSomething(T param) { ... }
}
```

Suppose the `doSomething()` method wants to make a defensive copy of the `param` argument on entry? You don't have many options. You'd like to implement `doSomething()` like this:

```
public void doSomething(T param) {
  T copy = new T(param);  // illegal
}
```

But you cannot use a type parameter to access a constructor because, at compile time, you don't know what class is being constructed and therefore what constructors are available. There's no way to express a constraint like "`T` must have a copy constructor" (or even a no-arg constructor) using generics, so accessing constructors for classes represented by generic type parameters is out.

What about `clone()`? Let's say `Foo` was defined to make `T` extend `Cloneable`:

```
class Foo<T extends Cloneable> {
  public void doSomething(T param) {
    T copy = (T) param.clone();  // illegal
  }
}
```

Unfortunately, you still can't call `param.clone()`. Why? Because `clone()` has protected access in `Object` and, to call `clone()`, you have to call it through a reference to a class that has overridden `clone()` to be public. But `T` is not known to redeclare `clone()` as public, so cloning is also out.

Constructing wildcard references

OK, so you can't copy a reference to a type whose class is totally unknown at compile time. What about wildcard types? Suppose you want to make a defensive copy of a parameter whose type is `Set<?>`. You know that `Set` has a copy constructor. You've also been told it's better to use `Set<?>` instead of the raw type `Set` when you don't know the type of the set's contents, because that approach is likely to emit fewer unchecked conversion warnings. So you try this:

```
class Foo {
  public void doSomething(Set<?> set) {
    Set<?> copy = new HashSet<?>(set);  // illegal
  }
}
```

Unfortunately, you can't invoke a generic constructor with a wildcard type argument, even though you know such a constructor exists. However, you can do this:

```
class Foo {
  public void doSomething(Set<?> set) {
    Set<?> copy = new HashSet<Object>(set);
  }
}
```

This construction is not the most obvious, but it is type-safe and will do what you think `new HashSet<?>(set)` would do.

Constructing arrays

How would you implement `ArrayList<V>`? The `ArrayList` class is supposed to manage an array of `V`, so you might expect the constructor for `ArrayList<V>` to create an array of `V`:

```
class ArrayList<V> {
  private V[] backingArray;
  public ArrayList() {
    backingArray = new V[DEFAULT_SIZE]; // illegal
  }
}
```

But this code does not work -- you cannot instantiate an array of a type represented by a type parameter. The compiler doesn't know what type `V` really represents, so it cannot instantiate an array of `V`.

The Collections classes use an ugly trick to get around this problem, one that generates an unchecked conversion warning when the Collections classes are compiled. The constructor for the real implementation of `ArrayList` looks likes this:

```
class ArrayList<V> {
  private V[] backingArray;
  public ArrayList() {
    backingArray = (V[]) new Object[DEFAULT_SIZE];
  }
}
```

Why does this code not generate an `ArrayStoreException` when `backingArray` is accessed? After all, you can't assign an array of `Object` to an array of `String`. Well, because generics are implemented by erasure, the type of `backingArray` is actually `Object[]`, because `Object` is the erasure of `V`. This means that the class is really expecting `backingArray` to be an array of `Object` anyway, but the compiler does extra type checking to ensure that it contains only objects of type `V`. So this approach will work, but it's ugly, and not really something to emulate (even the authors of the generified Collections framework say so -- see Resources).

An alternate approach would have been to declare `backingArray` as an array of `Object`, and cast it to `V[]` everywhere it is used. You would still get unchecked conversion warnings (as you do with the previous approach), but it would have made some unstated assumptions (such as the fact that `backingArray` should not escape the implementation of `ArrayList`) more clear.

The road not taken

The best approach would be to pass a class literal (`Foo.class`) into the constructor, so that the implementation could know, at runtime, the value of `T`. The reason this approach was not taken was for backward compatibility -- then the new generified collection classes would not be compatible with previous versions of the Collections framework.

Here's how `ArrayList` would have looked under this approach:

```
public class ArrayList<V> implements List<V> {
  private V[] backingArray;
  private Class<V> elementType;

  public ArrayList(Class<V> elementType) {
    this.elementType = elementType;
    backingArray = (V[]) Array.newInstance(elementType, DEFAULT_LENGTH);
  }
}
```

But wait! There's still an ugly, unchecked cast there when calling `Array.newInstance()`. Why? Again, it's because of backward compatibility. The signature of `Array.newInstance()` is:

```
public static Object newInstance(Class<?> componentType, int length)
```

instead of the type-safe:

```
public static<T> T[] newInstance(Class<T> componentType, int length)
```

Why was `Array` generified this way? Again, the frustrating answer is to preserve backward compatibility. To create an array of a primitive type, such as `int[]`, you call `Array.newInstance()` with the `TYPE` field from the appropriate wrapper class (in the case of `int`, you would pass `Integer.TYPE` as the class literal). Generifying `Array.newInstance()` with a parameter of `Class<T>` instead of `Class<?>` would have been more type-safe for reference types, but would have made it impossible to use `Array.newInstance()` to create an instance of a primitive array. Perhaps in the future, an alternate version of `newInstance()` will be provided for reference types so you can have it both ways.

You may see a pattern beginning to emerge here -- many of the problems, or compromises, associated with generics are not issues with generics themselves, but side effects of the requirement to preserve backward compatibility with existing code.

Generifying existing classes

Converting existing library classes to work smoothly with generics was no small feat, but as is often the case, backward-compatibility does not come for free. I've already talked about two examples where backward compatibility limited the generification of the class libraries.

Another method that would probably have been generified differently had backward compatibility not been an issue is `Collections.toArray(Object[])`. The array passed into `toArray()` serves two purposes -- if the collection is small enough to fit in the array provided, the contents will simply be placed in that array. Otherwise, a new array of the same type will be created, using reflection, to receive the results. If the Collections framework were being rewritten from scratch, it is likely that the argument to `Collections.toArray()` would not be an array, but instead a class literal:

```
interface Collection<E> {
  public T[] toArray(Class<T super E> elementClass);
}
```

Because the Collections framework is widely emulated as an example of good class design, it is worth noting the areas in which its design was constrained by backwards compatibility, so that those aspects are not emulated blindly.

One element of the generifed Collections API that is often confusing at first is the signatures of `containsAll()`, `removeAll()`, and `retainAll()`. You might expect the signatures for `remove()` and `removeAll()` to be:

```
interface Collection<E> {
  public boolean remove(E e);  // not really
  public void removeAll(Collection<? extends E> c);  // not really
}
```

But it is in fact:

```
interface Collection<E> {
  public boolean remove(Object o);
  public void removeAll(Collection<?> c);
}
```

Why is this? Again, the answer lies in backward compatibility. The interface contract of `x.remove(o)` means "if `o` is contained in `x`, remove it; otherwise, do nothing." If `x` is a generic collection, `o` does not have to be type-compatible with the type parameter of `x`. If `removeAll()` were generified to only be callable if its argument was type-compatible (`Collection<? extends E>`), then certain sequences of code that were legal before generics would become illegal, like this one:

```
// a collection of Integers
Collection c = new HashSet();
// a collection of Objects
Collection r = new HashSet();
c.removeAll(r);
```

If the above fragment were generified in the obvious way (making `c` a `Collection<Integer>` and `r` a `Collection<Object>`), then the code above would not compile if the signature of `removeAll()` required its argument to be a `Collection<? extends E>`, instead of being a no-op. One of the key goals of generifying the class libraries was to not break or change the semantics of existing code, so `remove()`, `removeAll()`, `retainAll()`, and `containsAll()` had to be defined with a weaker type constraint than they might have had they been redesigned from scratch for generics.

Existing classes that were designed before generics may have semantics that resist the "obvious" generification approach. In these cases, you will probably make compromises like the ones described here, but when designing new generic classes from scratch, it is valuable to understand which idioms from the Java class libraries are the result of backward compatibility, so they are not emulated inappropriately.

Implications of erasure

Because generics are implemented almost entirely in the Java compiler, and not in the runtime, nearly all type information about generic types has been "erased" by the time the bytecode is generated. In other words, the compiler generates pretty much the same code you would have written by hand without generics, casts and all, after checking the type-safety of your program. Unlike in C++, `List<Integer>` and `List<String>` are the same class (although they are different types, both subtypes of `List<?>` -- a distinction that is more important in JDK 5.0 than in previous versions of the language).

One of the implications of erasure is that a class cannot implement both `Comparable<String>` and `Comparable<Number>`, because both of these are in fact the same interface, specifying the same `compareTo()` method. It might seem sensible to want to declare a `DecimalString` class that is comparable to both `String`s and `Number`s, but to the Java compiler, you would be trying to declare the same method twice:

```
public class DecimalString implements Comparable<Number>, Comparable<String> { ... } // nope
```

Another consequence of erasure is that using casts or `instanceof` with generic type parameters doesn't make any sense. The following code will not improve the type safety of your code at all:

```
public <T> T naiveCast(T t, Object o) { return (T) o; }
```

The compiler will simply emit an unchecked conversion warning, because it doesn't know if the cast is safe or not. The `naiveCast()` method will not in fact do any casting at all -- `T` will simply be replaced with its erasure (`Object`), and the object passed in will be cast to `Object` -- not what was intended.

Erasure is also responsible for the construction issues described above -- that you cannot create an object of generic type because the compiler doesn't know what constructor to call. If your generic class needs to be constructing objects whose type is specified by generic type parameters, its constructors should take a class literal (`Foo.class`) and store it so that instances can be created with reflection.

Summary

Generics are a big step forward in the type-safety of the Java language, but the design of the generics facility, and the generification of the class library, were not without compromises. Extending the virtual machine instruction set to support generics was deemed unacceptable, because that might make it prohibitively difficult for JVM vendors to update their JVM. Accordingly, the approach of erasure, which could be implemented entirely in the compiler, was adopted. Similarly, in generifying the Java class libraries, the desire to maintain backward compatibility placed many constraints on how the class libraries could be generified, resulting in some confusing and frustrating constructions (like `Array.newInstance()`). These are not problems with generics per se, but with the practicality of language evolution and compatibility. But they can make generics a little more confusing, and frustrating, to learn and use.

Resources

- Get a more complete introduction to generic types in Brian Goetz' "Introduction to generic types in JDK 5.0" (developerWorks, December 2004).
- Eric Allen's four-part series, "Java generics without the pain," describes the evolution of support for generic functions in the Java language (developerWorks, February 2003 thru May 2003).
- The specification for generics, including the changes to the Java Language Specification, was developed through the Java Community Process under JSR 14.
- Angelika Langer has put together a wonderful FAQ on generics.
- Gilad Bracha, the principal architect of generic type support in the Java language, has written a tutorial on generics (PDF).
- Find hundreds more Java technology resources on the developerWorks Java technology zone.
- Browse for books on these and other technical topics.

About the author

Brian Goetz

Brian Goetz has been a professional software developer for over 17 years. He is a Principal Consultant at Quiotix, a software development and consulting firm located in Los Altos, California, and he serves on several JCP Expert Groups. See Brian's published and upcoming articles in popular industry publications.