

Monday, December 04, 2006

Super Type Tokens

When we added generics to Java in JDK5, I changed the class `java.lang.Class` to become a generic type. For example, the type of `String.class` is now `Class<String>`. [Gilad Bracha](#) coined the term *type tokens* for this. My intent was to enable a particular style of API, which Joshua Bloch calls the *THC, or Typesafe Heterogenous Container* pattern. For some examples of where this is used see the APIs for annotations:

```
public <A extends Annotation> A  
java.lang.Class.getAnnotation  
(Class<A> annotationClass)
```

My earliest use of this feature (like my earliest use of all recent Java language features) appears in the compiler for the Java programming language (javac), in this case as a utility called Context, and you can [find the code](#) in the open source version. It was a utility that allowed the compiler to be written as a bunch of separate classes that all refer to each other, and solved the hard problem of getting all the parts created in an order such that they can be initialized with references to each other. The utility is also used to replace pieces of the compiler, for example to make related tools like [javadoc](#) and [apt, the Annotation Processing Tool](#), and for testing. Today I would describe the utility as a simple *dependency injection* framework, but that wasn't a popular buzzword at the time.

Here is a simple but complete example of an API that uses type tokens in the THC pattern, from [Josh's 2006 JavaOne talk](#):

```
public class Favorites {  
    private Map<Class<?>, Object> favorites  
    =  
        new HashMap<Class<?>, Object>();  
    public <T> void setFavorite(Class<T>  
klass, T thing) {  
        favorites.put(klass, thing);  
    }  
    public <T> T getFavorite(Class<T>  
klass) {  
        return klass.cast(favorites.get  
(klass));  
    }  
    public static void main(String[] args) {  
        Favorites f = new Favorites();  
        f.setFavorite(String.class, "Java");  
        f.setFavorite(Integer.class,  
0xcafebabe);  
        String s = f.getFavorite  
(String.class);  
    }  
}
```

```

        int i = f.getFavorite
(Integer.class);
    }
}

```

A `Favorites` object acts as a typesafe map from type tokens to instances of the type. The main program in this snippet adds a favorite `String` and a favorite `Integer`, which are later taken out. The interesting thing about this pattern is that a single `Favorites` object can be used to hold things of many (i.e. heterogenous) types but in a typesafe way, in contrast to the usual kind of map in which the values are all of the same static type (i.e. homogenous). When you get your favorite `String`, it is of type `String` and you don't have to cast it.

There is a limitation to this pattern. [Erasure](#) rears its ugly head:

```

Favorites:15: illegal start of expression
f.setFavorite(List<String>.class,
Collections.emptyList());
               ^

```

You can't add your favorite `List<String>` to a `Favorites` because you simply can't make a type token for a generic type. This design limitation is one that a number of people have been running into lately, most recently [Ted Neward](#). "[Crazy](#)" [Bob Lee](#) also asked me how to solve a related problem in a dependency injection framework he is developing. The short answer is that you can't do it using type tokens.

On Friday I realized you can solve these problems without using type tokens at all, using a library. I wish I had realized this three years ago; perhaps there was no need to put support for type tokens directly in the language. I call the new idea *super type tokens*. In its simplest form it looks like this:

```

public abstract class TypeReference<T> {}

```

The `abstract` qualifier is intentional. It forces clients to subclass this in order to create a new instance of `TypeReference`. You make a super type token for `List<String>` like this:

```

TypeReference<List<String>> x = new
TypeReference<List<String>>() {};

```

Not quite as convenient as writing `List<String>.class`, but this isn't too bad. It turns out that you can use a super type token to do nearly everything you can do with a type token, and more. The object that is created on the right-hand-side is an anonymous class, and using reflection you can get its interface type, including generic

type parameters. Josh calls this pattern "Gafter's Gadget". [Bob Lee](#) elaborated on this idea as follows:

```
import java.lang.reflect.Constructor;
import
java.lang.reflect.InvocationTargetException;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
import java.util.ArrayList;
import java.util.List;

/**
 * References a generic type.
 *
 * @author crazybob@google.com (Bob Lee)
 */
public abstract class TypeReference<T> {

    private final Type type;
    private volatile Constructor<?>
constructor;

    protected TypeReference() {
        Type superclass = getClass
().getGenericSuperclass();
        if (superclass instanceof Class) {
            throw new RuntimeException
("Missing type parameter.");
        }
        this.type = ((ParameterizedType)
superclass).getActualTypeArguments()[0];
    }

    /**
     * Instantiates a new instance of
{@code T} using the default, no-arg
     * constructor.
     */
    @SuppressWarnings("unchecked")
    public T newInstance()
        throws NoSuchMethodException,
        IllegalAccessException,
        InvocationTargetException,
        InstantiationException {
        if (constructor == null) {
            Class<?> rawType = type
instanceof Class<?>
                ? (Class<?>) type
                : (Class<?>)
((ParameterizedType) type).getRawType();
            constructor =
rawType.getConstructor();
        }
        return (T) constructor.newInstance
();
    }
}
```

```

    /**
     * Gets the referenced type.
     */
    public Type getType() {
        return this.type;
    }

    public static void main(String[] args)
    throws Exception {
        List<String> l1 = new
    TypeReference<ArrayList<String>>()
    {}.newInstance();
        List l2 = new
    TypeReference<ArrayList>() {}.newInstance();
    }
}

```

This pattern can be used to solve [Ted Neward's](#) problem, and most problems where you would otherwise use type tokens but you need to support generic types as well as [reifiable](#) types. Although this isn't much more than a [generic factory interface](#), the automatic hook into the rich generic reflection system is more than you can get with simple class literals. With a few more bells and whistles (`toString`, `hashCode`, `equals`, etc) I think this is a worthy candidate for inclusion in the JDK.