Annotation Hammer

Venkat Subramaniam
venkats@agiledeveloper.com
(Also published at http://www.infoq.com)

Abstract

Annotations in Java 5 provide a very powerful metadata mechanism. Yet, like anything else, we need to figure out where it makes sense to use it. In this article we will take a look at why Annotations matter and discuss cases for their use and misuse.

Expressing Metadata

Let's start with what we're familiar with as good old Java programmers. We want to express that an instance of a class may be serialized. We can say this by implementing the Serializable interface as shown here:

```
public class MyClass implements java.io.Serializable
{
}
```

How many methods does the Serializable interface have? The answer, of course, is zero. Now, why would we have an interface with zero methods and inherit from it? I call it the use of the *inheritance hammer*. We didn't use inheritance in this case to derive any behavior, or express a specific contract, but to provide our consent that it is OK to serialize an object of this class if a user of the class so desires. We call the Serializable interface a tagging interface. Other interfaces like Cloneable fall into this tradition of tagging interfaces. Now, what if I have a field within my object and I don't want it to be serialized? Java uses the `transient` keyword to express this as shown below:

```
public class MyClass implements java.io.Serializable
{
    private int val1;
    private transient int val2;
}
```

So, we needed an interface (Serializable) and a keyword (transient) to get our job done in the above example.

Let's proceed further with this example. Assume that I have a framework that provides some service. You may send objects of your class to my framework. However, I need to know if your objects are thread-safe, after all if it's not thread-safe you wouldn't want me to use it concurrently from multiple threads. Continuing from what we've learned from the above example, I can define a tagging interface (let me call it ThreadSafe). If you implement this interface, then I can figure out that your class is thread-safe.

```
public class MyClass
      implements java.io.Serializable, VenkatsFramework.ThreadSafe
{
    private int val1;
    private transient int val2;
}
```
See that was simple! Now, let’s assume you have a method of the class which, for whatever reason, should not be called from multiple threads. How do we do that? No problem. We can kindly request a new keyword to be introduced into the Java language, so we can mark our method using the new keyword (or you can argue that we can use the `synchronized` keyword, but you can see the limitation of this approach of using a keyword).

```
public class MyClass
      implements java.io.Serializable, VenkatsFramework.ThreadSafe
{
    private int val1;
    private transient int val2;

    public our_new_fangled_keyword void foo() // Not valid Java code
    {
    //...
    }
}
```

As you can see, we lack the expressive power to extend the metadata. What we want to do is, so to say, color the classes, methods, and fields to say that it is Serializable, thread-safe, or whatever that we wish to express based on our need.

More Power to You

Enter Annotations. Annotations provide a way for us to extend the Java language with new metadata. It provides great expressive power. Let’s see how we can express the concept we described in the previous example, elegantly using annotations.

```
//ThreadSafe.java
package VenkatsFramework;

public @interface ThreadSafe
{

}

//NotThreadSafe.java
package VenkatsFramework;

public @interface NotThreadSafe
{
}


//MyClass.java
package com.agiledeveloper;

import VenkatsFramework.ThreadSafe;
import VenkatsFramework.NotThreadSafe;

@ThreadSafe
public class MyClass implements java.io.Serializable
```

```
{
    private int val1;
    private transient int val2;

    @NotThreadSafe public void foo()
    {
    //...
    }
}
```

The annotation `ThreadSafe` is written as if it is an interface (more about this later). In `MyClass`, I have used the annotation `ThreadSafe` (for the class) and `NotThreadSafe` (for the method). When using frameworks, we would often use annotations rather than defining them. However, it is interesting to learn how to define annotations.

Defining Annotations

Let's define an annotation called `AuthorInfo` which can be used to express who wrote a piece of code. Here is the code:

```
package com.agiledeveloper;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@Inherited
public @interface AuthorInfo
{
    public String name();
    public String email() default "";
    public String comment() default "";
}
```

The `AuthorInfo` annotation has three elements: `name`, `email`, and `comment`. These are defined using a method declaration like syntax. The `email` and `comment` have default values, so these two may be left out when `@AuthorInfo` annotation is used. If an annotation has only a single value, you can specify that value without the member name.

Where (method, class, field, etc.) can you use the `@AuthorInfo` annotation? This is defined using the meta-annotation `@Target` (Nice: annotations eating their own dog food). The possible `ElementType` values for the `@Target` annotation are: `ANNOTATION_TYPE`, `CONSTRUCTOR`, `FIELD`, `LOCAL_VARIABLE`, `METHOD`, `PACKAGE`, `PARAMETER`, and `TYPE`. The `@Inherited` meta-annotation indicates that the annotation not only affects the class that declares the annotation, but also any class derived from the declaring class (In the case of `AuthorInfo` annotation, it really doesn't make sense to use `Inherited` as the author of a class may be different from the author of its base class).

Finally, the `@Retention` meta-annotation tells us how far the annotation will be. If the value is `RetentionPolicy.SOURCE` then the annotation is seen in the source code, but discarded by the compiler. The value of `RetentionPolicy.CLASS` means the annotation information will be retained in the class file, but not loaded into the virtual machine. The value of `RetentionPolicy.RUNTIME` means the value is retained at runtime and you can use reflection to explore the annotation details. One more meta-annotation not shown here is `@Documented` which indicates that the use of annotation may be documented in javadoc.

Using the Annotations

Here is an example of using the `AuthorInfo` annotation:

```
package com.agiledeveloper;

@com.agiledeveloper.AuthorInfo(name = "Venkat Subramaniam")
public class SomeClass
{
    @com.agiledeveloper.AuthorInfo(
            name = "Venkat Subramaniam", comment = "bug free")
    public void foo()
    {

    }
}
```

`SomeClass` specifies the `@AuthorInfo` annotation with a value for the `name` element. Similarly the `foo()` method has the `@AuthorInfo` annotation.

The following example is not valid:

```
package com.agiledeveloper;

@com.agiledeveloper.AuthorInfo(name = "Venkat Subramaniam")
public class SomeClass2
{
    // ERROR 'com.agiledeveloper.AuthorInfo' not applicable to fields
    @com.agiledeveloper.AuthorInfo(name = "Venkat Subramaniam")
            // Not valid
    private int val1;
}
```

The compiler gives an error message since the `@AuthorInfo` is useable only on classes and methods (as defined by the `@Target` meta-annotation).

Exploring Annotations

As I mentioned earlier, most of the time we will use annotations rather than define them. However, if you are curious how a framework would use the annotation, here it is. Using reflection you can explore the details of annotation on your class, method, field, etc. An example code that explores the `@AuthorInfo` on the class is shown below:

```
package com.agiledeveloper;
```

```
import java.lang.reflect.Method;

public class Example
{
    public static void report(Class theClass)
    {
        if (theClass.isAnnotationPresent(AuthorInfo.class))
        {
            AuthorInfo authInfo =
                 (AuthorInfo) theClass.getAnnotation(AuthorInfo.class);

            System.out.println(theClass.getName() +
                        " has @AuthorInfo annotation:");
            System.out.printf("name = %s, email = %s, comment = %s\n",
                  authInfo.name(),
                  authInfo.email(), authInfo.comment());
        }
        else
        {
            System.out.println(theClass.getName()
            + " doesn't have @AuthorInfo annotation");
        }

        System.out.println("-----------------");
    }

    public static void main(String[] args)
    {
        report(SomeClass.class);
        report(String.class);
    }
}
```

The output from the above program is shown below:

```
com.agiledeveloper.SomeClass has @AuthorInfo annotation:
name = Venkat Subramaniam, email = , comment =
-----------------
java.lang.String doesn't have @AuthorInfo annotation
-----------------
```

The `isAnnotationPresent()` method of `Class` tells us if the class has the expected Annotation. You can fetch the Annotation details using the `getAnnotation()` method.

An Example of Annotation

Let's take a look at an annotation built into Java.

```
package com.agiledeveloper;

public class POJOClass
{
    /**
     * @deprecated Replaced by someOtherFoo()…
     */
    public static void foo()
    {
    }

    @Deprecated public static void foo1()
    {
    }
}
```

The `foo()` method uses the traditional way to declare a method as `deprecated`. This approach lacked the expressiveness and, even though Sun compilers typically provide a warning if you use a method declared as deprecated, there is no guarantee that all compilers will issue a warning for this. A more standardized and portable version of this is to use `@Deprecated` annotation (though it lacks the power to provide the description for deprecation that the older approach allowed–so you typically would use it in combination with the older approach) as in:

```
    /**
     * @deprecated Replaced by someOtherFoo1()...
     */
    @Deprecated public static void foo1()
    {
    }
```

Annotation and the Hammer

“If the only tool you have is a hammer, then everything looks like a nail,” goes a saying. While annotations are a good tool, not every situation warrants their use. Most of us have come to dislike XML configuration. However, suddenly, everything in the XML configuration shouldn’t become annotation.

Use annotations for what you want to express intrinsically in code. For example, the `@Persist` annotation in Tapestry is a good example. You want to declare a property of a bean as persistent and Tapestry will take care of storing it (in the session for instance). I would much rather define this as annotation than using a verbose configuration to say the same. The chances are, if I decide not to make the property persistent, much is going to change in the code anyways.

To consider another good example, in defining a web service, how do you describe which methods of your class need to be exposed as a web service method, so its description can appear in the WSDL? We have come across solutions that use configuration files for this. One problem with configuration file approach is, if you modify the code (like change method name) you also have to modify the configuration file. Furthermore, you rarely really configure a method as a service method back and forth. Marking a method as web service method using annotation may make good sense.

The expressive power of annotations and their ability to extend the metadata of the language allow code generation tools to create code for you based on the characteristics you’ve expressed. Annotations can also help us express some aspects.

Now consider using annotation to configure security settings for a method. That would be a stretch. It's likely that you will modify the security settings during deployment. You may want to be able to alter it without having to modify the code and recompiling. Annotation is not the best choice to express things that are somewhat extrinsic and better expressed outside of the code. Does that mean we need to use extensive XML configuration for these? Not necessarily, as we discuss in the next section.

Convention over configuration

Certain things are better suited to be configured and expressed using Annotations. Certain things are better suited to be expressed external and separated from the code, may be in XML, YAML, etc. However, not every thing should be configured. Configuration provides flexibility. Just like in real life, too much of anything can be bad. Certain things may be easily figured out in the application based on convention rather than configuration.

For instance, in pre JUnit 4.0 versions, you would indicate that a method is a test method by prefixing it with test. In JUnit 4.0, you instead mark a method using the @Test annotation. Not looking at other features and benefits of JUnit 4.0, is this better? You may point out that the benefit you get from the use of annotation is that you don't have to extend your test class from the TestCase class. I agree, but, that's at the class level. At the method level, is it less noise, less clutter to simply write a method as:

```
public void testMethod1() {}
```

or

```
@Test public void method1() {}
```

Why not, by default, consider all methods in a test class as test methods? Then may be you can specify (may be using annotation) that a method is not a test method. Kind of like how in Java a method is considered virtual (polymorphic) unless you declare it final. Then you are making less noise to communicate your intent, isn't it?

The point I am making is that there is nothing wrong in using convention where it makes sense, improves the signal-to-noise ratio in the code, less clutter, and less typing.

To Annotate or Not To

When to use annotations is an interesting question to ask. Both the answers "Always" and "Never" are incorrect. There are places where annotation is suitable and even elegant. Then there are places where it may not be the best choice.

Use Annotation if:

- Metadata is intrinsic
  - If you would have used a keyword, if available in the language, then this may be a candidate for annotation (`transient` for example).
- Is simpler to express and easier to work with as annotation than otherwise

    - For instance, it is easier to mark a method as a web method in a web service instead of writing an xml configuration file to say the same.
- Class based, not object specific
    - Represents metadata that is at the class level, irrespective of any specific object of the class

Don’t use Annotation if:

- Just because it’s currently in xml config does not mean it should become an annotation now
    - Don’t blindly convert your xml configuration information to annotations
- You already have an elegant way to specify this
    - If the way you are representing this information is already elegant and sufficient, don’t fall into the pressure of using annotation.
- Metadata is something you will want to change
    - You want this to be modified or configurable, for example security settings for a method of a bean–you may want to modify this at any time. These may be better left in a configuration file.
- Your application can figure the details out from convention instead of configuration
    - You can configure (using annotation or otherwise) only things that are different from the obvious or the default.

Conclusion

Annotations are an interesting addition to the language. They provide a powerful construct for extending the Java language with new metadata. However, we need to take the time to evaluate its merit on a case by case basis. “The right use of annotations” is a design concern that deserves due consideration in application development–neither blind acceptance nor rejection of it is a good idea.