Java Annotations

By Bhaskar Swaminathan
03/22/2009

Annotations were introduced in Java since the JDK 5 release. Annotations are meta-data tags that can be attached to various Java code elements such as, class declarations, interface declarations, field declarations, constructor definitions, method definitions, etc and can later be accessed either at compile-time or at run-time. Annotations provide additional information about code elements, which can be used by tools and/or frameworks for further processing in interesting ways, such as, controlling the behavior of the compiler, configuring the application at run-time, or profiling an application at run-time, etc.

Consider the following simple Java program:

```
/*
 * Name: Bhaskar S
 *
 * Date: 03/22/2009
 */

package com.polarsparc.annotations.samples;

import java.util.*;

public class Suppress {
	public static void main(String[] args) {
		Map map = new HashMap();
		for (String s : args) {
			map.put(s, s);
		}
	}
}
```

The above program will compile just fine in pre-Java 5 compiler. What about the Java 6 compiler ? Give it a try and you will see an output similar to what is shown below:

```
$ javac com/polarsparc/annotations/samples/Suppress.java
Note: com/polarsparc/annotations/samples/Suppress.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Interesting behavior in Java 6 !!! Neither the above Java program is incorrect nor is the behavior of Java 6 compiler. Java 6 has a much stronger type checking and as a result generates the above warning.

Let us modify the simple Java program by adding one of the Standard Java annotations as shown below:

```
/*
 * Name: Bhaskar S
 *
 * Date: 03/22/2009
 */

package com.polarsparc.annotations.samples;

import java.util.*;
```

```
public class Suppress {
	@SuppressWarnings("unchecked")
	public static void main(String[] args) {
		Map map = new HashMap();
		for (String s : args) {
			map.put(s, s);
		}
	}
}
```

The @SuppressWarnings("unchecked") annotation provides information to the compiler to suppress warnings related to type checking. It does not alter the behavior or logic of the code; its just an additional information to the compiler to change certain behavior.

In addition to **@SuppressWarnings**, the current Java 6 language specification predefines two additional standard annotations: **@Override** and **@Deprecated**.

The **@Override** annotation indicates to the compiler that the following method overrides the method declared in the superclass.

The **@Deprecated** annotation indicates to the compiler that the following method may not be supported in the future and hence should be avoided.

As we can see the standard annotations **@SuppressWarnings**, **@Override**, and **@Deprecated** provide additional meta information about the source elements, which is used by the Java compiler.

Look at the extensive use of Annotations in EJB3. Annotations are used to configure EJB(s) at deployment time without the need for any external XML Deployment Descriptor.

Similarly, we can define our own custom annotations, which we could use in our applications. Hang in there and we will do just that !!!

To define a Custom Annotation, we need the following three ingredients:

Ingredient	Description
Annotation Type	Defines the type for the Custom Annotation. It is nothing more than a special type of Java ***interface*** – the only difference being the use of the new keyword ***@interface*** instead of the regular keyword ***interface***
Annotation Member(s)	Defines one or more field(s) for associating named content with the Custom Annotation. They are defined as methods (similar to Java ***interface***) which can only return Java primitive types, have no method parameters and have no ***throws*** keyword. Definition of the field also allows one to specify a default value. It is done using the ***default*** keyword followed by the value
Meta-Annotation(s)	Defines meta-data that can only be applied to Custom Annotations at the time of definition. Think of these as Annotations for Annotations.

There are three flavors of Annotation Types – **Marker** Annotations, **Single-value** Annotations and **Multi-value** Annotations.

A **Marker** Annotation Type only has an Annotation name. The following is an example of Marker Annotation Type:

```
/*
 * Name: Bhaskar S
 *
 * Date: 03/22/2009
 */

package com.polarsparc.annotations.samples;

public @interface MyMarkerType {
}
```

MyMarkerType is the name of the Marker Annotation Type.

The standard Java Annotations **@Override** and **@Deprecated** are examples of Marker Annotation Types.

A **Single-value** Annotation Type has an Annotation name and one data field. The following is an example of Single-value Annotation Type:

```
/*
 * Name: Bhaskar S
 *
 * Date: 03/22/2009
 */

package com.polarsparc.annotations.samples;

public @interface MySingleValueType {
        String value() default "abc";
}
```

MySingleValueType is the name of the Single-value Annotation Type. Here value is the name of the data field. Look at the way the data variable is defined – ***String value()***. This is consistent with the way methods are defined in an ***interface***. Also, the data field has defined a default of "abc". If no content is associated with value, then it defaults to "abc".

To use this annotation, we would type: @MySingleValueType(value="xyz") OR the short form @MySingleValueType("xyz") since there is only one data variable. We can also use the Annotation without specifying any data, such as @MySingleValueType. In this case the field value defaults to the string "abc".

The standard Java Annotation **@SuppressWarnings** is an example of Single-value Annotation Type.

A **Multi-value** Annotation Type has an Annotation name and more than one data fields. The following is an example of Multi-value Annotation Type:

```
/*
 * Name: Bhaskar S
```

```
 *
 * Date: 03/22/2009
 */

package com.polarsparc.annotations.samples;

public @interface MyMultiValueType {
        int value1();
        String value2();
        String value3();
}
```

MyMultiValueType is the name of the Multi-value Annotation Type. Here value1, value2, and value3 are the names of the data fields.

To use this annotation, we would type: @MyMultiValueType(value1=7, value2=”abc”, value3=”xyz”).

Java defines 4 standard Meta-Annotations: @**Target**, @**Retention**, @**Documented**, and @**Inherited**.

The @**Target** Meta-Annotation specifies to which Java code element(s) the Custom Annotation applies. The various Java code elements are defined in the ***enum*** java.lang.annotation.ElementType which are as follows:

Element Type	Description
ElementType.PACKAGE	Applies to ***package***
ElementType.TYPE	Applies to ***class***, ***interface***, or ***enum***
ElementType.FIELD	Applies to the fields
ElementType.METHOD	Applies to the methods
ElementType.PARAMETER	Applies to the parameters of the method
ElementType.CONSTRUCTOR	Applies to the constructors
ElementType.LOCAL_VARIABLE	Applies to the local variables

The following is an example of Marker Annotation Type that can be applied at the class, field, or method level:

```
/*
 * Name: Bhaskar S
 *
 * Date: 03/22/2009
 */

package com.polarsparc.annotations.samples;

import java.lang.annotation.*;

@Target({ElementType.TYPE,
         ElementType.FIELD,
         ElementType.METHOD})
public @interface MyMarkerType {
}
```

The @Retention Meta-Annotation indicates where the Custom Annotation details are retained. It can be

retained at the Java source level, or at the Java class level, or at the Java runtime level. The various retention codes are defined in the ***enum*** java.lang.annotation.RetentionPolicy which are as follows:

Retention Type	Description
RetentionPolicy.SOURCE	Retained only in the Java source code and read by the Java compiler but will be discarded from the Java class file
RetentionPolicy.CLASS	Retained in the Java class file by the Java compiler and will be not be accessible through reflection API at runtime. This is the default retention policy
RetentionPolicy.RUNTIME	Retained in the Java class file by the Java compiler and will be accessible through reflection API at runtime

The following is an example of Single-value Annotation Type that is retained in the class and is accessible through Java reflection API:

```
/*
 * Name: Bhaskar S
 *
 * Date: 03/22/2009
 */

package com.polarsparc.annotations.samples;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
public @interface MySingleValueType {
        String value();
}
```

The **@Documented** Meta-Annotation indicates that the Custom Annotation should be processed by the **javadoc** tool and included in the generated documentation. By default, the **javadoc** tool does not document Annotations.

The **@Inherited** Meta-Annotation makes the Custom Annotation applied to a super-class to be inherited by a derived-class. By default Annotations are not inherited.

Now that we are familiar with the intricacies of Annotations, we will go ahead and build our own custom Annotation. Many a times we specify a version number for our application. We usually achieve this by initializing a ***static final String*** called VERSION as shown in the following:

```
/*
 * Name: Bhaskar S
 *
 * Date: 03/22/2009
 */

package com.polarsparc.annotations.samples;

public class StaticVersion {
        static final String VERSION = "V1.5R3";

        public static void main(String[] args) {
                System.out.println("Version: " + VERSION);
```

```
                // Application logic starts here
        }
}
```

We will define a custom Annotation for specifying a version number. The following Java code defines the custom Annotation called Version:

```
/*
 * Name: Bhaskar S
 *
 * Date: 03/22/2009
 */

package com.polarsparc.annotations.samples;

import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Version {
        int major() default 1;
        int minor() default 0;
        String release() default "";
}
```

The Version Annotation can be targeted only at a ***class***, ***interface***, or ***enum*** level. Also, it is retained in the Java class file and can be queried using Java reflection API.

The following Java class uses the **@Version** Annotation:

```
/*
 * Name: Bhaskar S
 *
 * Date: 03/22/2009
 */

package com.polarsparc.annotations.samples;

@Version(minor=5,release="3289")
public class VersionedClass {
        // ---- class definition does here ---
}
```

We have only specified the fields minor and release and assigned it values 5 and “3289” respectively. The field major is not specified and hence defaults to 1.

The following Java program accesses the **@Version** Annotation using the Java reflection API:

```
/*
 * Name: Bhaskar S
 *
 * Date: 03/22/2009
 */
```

```
package com.polarsparc.annotations.samples;

import java.lang.annotation.Annotation;

public class VersionTest {
	public static void main(String[] args) {
		try {
			Class<?> clazz =
			    Class.forName("com.polarsparc.annotations.samples.VersionedClass");

			Annotation[] array = clazz.getAnnotations();
			for (Annotation an : array) {
				if (an instanceof Version) {
					Version v = (Version) an;

					System.out.println("Version: V" + v.major() + "." + v.minor() +
"R" + v.release());
				}
			}
		}
		catch (Throwable t) {
			t.printStackTrace(System.err);
		}
	}
}
```

When we execute this Java program, we will see the output: **V1.5R3289**

With this we conclude this tutorial on Java Annotations. You can try more interesting use-cases.