JAVA

COMO CREAR ANOTACIONES PROPIAS EN JAVA

Marlon Ernesto Figueroa Fuentes
marlon.f.1993@gmail.com

Tabla de contenido

Introducción

Las anotaciones en Java se incluyeron a partir de Java 5. Éstas son metadatos que se pueden asociar a clases, miembros, métodos o parámetros.

Nos dan información sobre el elemento que tiene la anotación y permiten definir cómo queremos que sea tratado por el framework de ejecución. Además, las anotaciones no afectan directamente a la semántica del programa, pero sí a la forma en que los programas son tratados por las herramientas y librerías.

Pero no sólo disponemos de las anotaciones por defecto de Java, si no que podemos crear las nuestras propias y así disponer de metadatos útiles en tiempo de ejecución.

Entorno

- Hardware: Portátil MacBook Pro 15 (2.4 GHz Intel Core I5, 8GB DDR3).
- Sistema Operativo: Mac OS Yosemite 10.10.3
- Entorno de desarrollo:
    - > Eclipse Mars
    - > Versión de Java: JDK 1.8

Creando la anotación

En Java, una anotación se define por medio de la palabra reservada @interface. Hay que tener ciertas consideraciones en cuenta a la hora de crearlas:

- Cada método dentro de la anotación es un elemento.
- Los métodos no deben tener parámetros o clausulas ***throws.***
- Los tipos de retorno están restringidos a tipos primitivos, String, enum, anotaciones y arrays de los tipos anteriores.
- Los métodos pueden tener valores por defecto.

Vamos a crear una anotación que indique cuantas veces debe ejecutarse cada elemento que se anote con ella.

MultipleInvocacion.java

```
package mipaquete.prueba;

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

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MultipleInvocation {

    int timesToInvoke() default 1;
}
```

Como se ve en el código, hemos usado algunas metaanotaciones para definir ciertos parámetros en nuestra anotación. Veamos cuáles de estas se pueden usar al crear nuestras anotaciones y qué significan:

@Target – Especifica el tipo de elemento al que se va a asociar la anotación.

- > ElementType.TYPE – se puede aplicar a cualquier elemento de la clase.
- > ElementType.FIELD – se puede aplicar a un miembro de la clase.
- > ElementType.METHOD – se puede aplicar a un método
- > ElementType.PARAMETER – se puede aplicar a parámetros de un método.
- > ElementType.CONSTRUCTOR – se puede aplicar a constructores
- > ElementType.LOCAL_VARIABLE – se puede aplicar a variables locales
- > ElementType.ANNOTATION_TYPE – indica que el tipo declarado en sí es un tipo de anotación.

@Retention – Especifica el nivel de retención de la anotación (cuándo se puede acceder a ella).

> RetentionPolicy.SOURCE — Retenida sólo a nivel de código; ignorada por el compilador.
> RetentionPolicy.CLASS — Retenida en tiempo de compilación, pero ignorada en tiempo de ejecución.
> RetentionPolicy.RUNTIME — Retenida en tiempo de ejecución, sólo se puede acceder a ella en este tiempo.

@Documented – Hará que la anotación se mencione en el javadoc.

@Inherited – Indica que la anotación será heredada automáticamente.

En nuestro caso hemos dicho que sea en tiempo de ejecución y que se aplique a métodos. También le hemos añadido un valor por defecto con la palabra ***default.***

Usando la anotación.

El funcionamiento de uso es muy sencillo; basta etiquetar con la anotación los métodos que queramos:

AutomaticWeapon.java

```
package mipaquete.prueba;

import mipaquete.prueba.MultipleInvocation;

public class AutomaticWeapon {

    private static final int BURST_FIRE_ROUNDS = 3;

    private static final int AUTO_FIRE_ROUNDS = 5;

    private int ammo;

    public AutomaticWeapon(int ammo) {
        this.ammo = ammo;
    }

    @MultipleInvocation
    public void singleFire() {
        ammo--;
        System.out.println("Single fire! Ammo left: " + ammo);
    }

    @MultipleInvocation(timesToInvoke = BURST_FIRE_ROUNDS)
    public void burstFire() {
        ammo--;
        System.out.println("Burst fire! Ammo left: " + ammo);
    }

    @MultipleInvocation(timesToInvoke = AUTO_FIRE_ROUNDS)
    public void autoFire() {
        ammo--;
        System.out.println("Auto fire! Ammo left: " + ammo);
    }
}
```

Para este ejemplo hemos creado una clase que modeliza de forma sencilla el funcionamiento de un arma automática con tres tipos de disparo:

- Único: gasta una bala por invocación.
- Ráfaga: gasta tres balas por invocación.
- Automático: gasta cinco balas por invocación.

Para modelizar los tipos de disparos nos valemos de la anotación que hemos creado, pasándole el número de disparos que debe hacer (recordemos que por defecto era 1).

Leyendo la anotación durante la ejecución

La lectura de la anotación en tiempo de ejecución se realiza mediante reflexión, pero sólo si ésta tiene un nivel de retención RUNTIME.

Vamos a acceder a la información de la anotación en tiempo de ejecución. Para ello, creamos una clase Operador que pruebe el arma automática: llamará a todos los métodos del arma y los ejecutará el número de veces que le diga la anotación (si esta existe).

Operator.java

```
package mipaquete.prueba;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

import mipaquete.prueba.MultipleInvocation;

public class Operator {

    public void operate(AutomaticWeapon weapon) {
        final String className = weapon.getClass().getName();
        try {
            final Method[] methods = Class.forName(className).getMethods();
            for (final Method method : methods) {
                invokeMethod(method, weapon);
            }
        } catch (final Exception e) {
            System.err.println("Hubo un error:" + e.getMessage());
        }
    }

    private void invokeMethod(Method method, AutomaticWeapon weapon)
            throws     IllegalAccessException,     IllegalArgumentException,
InvocationTargetException {

        final        MultipleInvocation        multipleInvocation        =
method.getAnnotation(MultipleInvocation.class);

        if (multipleInvocation != null) {
            final int timesToInvoke = multipleInvocation.timesToInvoke();

            for (int i = 0; i < timesToInvoke; i++) {
                method.invoke(weapon, (Object[])null);
            }
        }

    }
}
```

Lo que hace esta clase es muy sencillo:

1. Coge todos métodos de la clase que se le pasa (en nuestro caso, el arma).
2. Para cada método:
   2.1. Mira el número de veces a invocar.
   2.2. Invoca dicho número de veces con los argumentos necesarios (en este caso no hay).

NOTA: Soy consciente de que podría hacerse un código más «eficiente» usando de otra forma el API de Reflection de Java. Se decidió escribirlo así para mostrar mejor la forma de proceder. Esto es un ejemplo sencillo que muestra cómo tomar y usar los metadatos de las anotaciones.

Probando

Vamos a probar nuestra anotación con un sencillo programa principal que cree un operador y le asigne un arma a probar:

Operator.java

```
public class Main {

    public static void main(String[] args) {
        final AutomaticWeapon weapon = new AutomaticWeapon(30);
        final Operator operator = new Operator();
        operator.operate(weapon);
    }
}
```

Al ejecutarlo, la salida debería mostrar lo siguiente (salvo orden de métodos):

```
Auto fire! Ammo left: 29
Auto fire! Ammo left: 28
Auto fire! Ammo left: 27
Auto fire! Ammo left: 26
Auto fire! Ammo left: 25
Single fire! Ammo left: 24
Burst fire! Ammo left: 23
Burst fire! Ammo left: 22
Burst fire! Ammo left: 21
```

Vemos que efectivamente se ha ejecutado una vez el disparo único, tres veces el disparo de ráfagas y cinco el automático.

Conclusiones

Las anotaciones son un método muy útil de añadir u obtener información de los elementos que están anotados. Además, nos permiten acceder a ellos en tiempo de ejecución, con lo que los metadatos se pueden llegar a usar como variables internas para modificar el comportamiento del flujo del programa (como hemos visto).

En este tutorial sólo se ha explorado la forma de acceder en tiempo de ejecución, pero a mi parecer la forma más potente de usar estas anotaciones sería en tiempo de compilación, ya que se puede añadir código al programa para que se comporte de manera diferente, sin tener que modificar el código principal (se encargaría el compilador al crearlo). En futuros tutoriales puede que toque este tema, ya que sólo hemos arañado la superficie.