

Anotaciones

Fundamentos

➤ Permiten suministrar información al entorno de ejecución (metadatos) desde el propio código.

➤ Su sintaxis es:

```
@NombreAnotacion(atributo1="valor", atributo2="valor"..)
```

➤ Se pueden indicar delante de clases, métodos o atributos. Siempre deben ir delante del nombre del tipo:

```
@Anotacion var data; //correcto  
@Anotacion public void metodo(){...} //correcto  
String @Anotacion car; //incorrecto  
var n=@Anotacion "texto"; //incorrecto
```

➤ Java proporciona varias anotaciones predefinidas

Anotaciones personalizadas

- Podemos crear nuestras propias anotaciones personalizadas definiéndolas como una interfaz especial:

se emplea @interface
en lugar de interface

```
public @interface NuevaAnotacion{  
  
}
```

- La interfaz anterior debe estar anotada a su vez con dos anotaciones especiales, conocidas como metaanotaciones: @Target y @Retention.
- En cuanto al interior de la interfaz, ésta está formada por una serie de métodos que determinan los atributos expuestos por la anotación

Metaanotaciones

➤Target. Indica a qué tipo de elemento se aplicará la anotación:

- `ElementType.TYPE`. Se aplica a un tipo (clase, interface, enumeración).
- `ElementType.FIELD`. Se aplica a un miembro de la clase.
- `ElementType.METHOD`. Se aplica a un a un método
- `ElementType.PARAMETER`. Se aplica a parámetros de un método.
- `ElementType.CONSTRUCTOR`. Se aplica a constructores
- `ElementType.LOCAL_VARIABLE`. Se aplica a variables locales
- `ElementType.ANNOTATION_TYPE`. Indica que el tipo declarado en sí es un tipo de anotación.

➤Retention. Indica el nivel de retención de la anotación , es decir, su ámbito de acceso:

- `RetentionPolicy.SOURCE`. Retenida sólo a nivel de código, por lo que es ignorada por el compilador.
- `RetentionPolicy.CLASS`. Retenida en tiempo de compilación, pero ignorada en tiempo de ejecución.
- `RetentionPolicy.RUNTIME`. Retenida en tiempo de ejecución y sólo se puede acceder a ella en este tiempo

Ejemplo

➤ Anotación que, a través de su atributo “level”, define el nivel de detalle de un método encargado de un registro de sucesos:

Interpretada en tiempo de ejecución

Si indica valores por defecto, el atributo es opcional

```
@Retention(RUNTIME)
@Target(METHOD)
public @interface Log {
    Valores level() default Valores.UNO;
}
```

Se puede utilizar sobre métodos

Enumeración que define los posibles valores del atributo

```
public enum Valores {
    UNO,DOS,TRES
}
```

➤ Los valores de un atributo de anotación solo pueden ser primitivos, envoltorio, String o enumeraciones. También array de éstos

Revisión conceptos



Dada la siguiente anotación, indica cuales de las instrucciones indicadas son correctas

```
@Retention(RUNTIME)
@Target({METHOD, FIELD})
public @interface MyAnt{}
```

- a. `@MyAnt class Example{..}`
- b. `@MyAnt public void print(){}`
- c. `@MyAnt public static final int k=10;`
- d. `class Example{ @MyAnt public Example(){}}`

Respuesta

La anotación solo puede aplicar sobre atributos de la clase y métodos, luego b y c son correctas, mientras que a y d incorrectas

Manejo de la anotación

Interpreta la anotación

Utiliza la anotación

```
public class UsoAnotacion {  
    @Log(level = Valores.DOS)  
    public void prueba(String mensaje) {  
        System.out.println(mensaje);  
    }  
}
```

Lanza la aplicación

```
public class Inicio {  
    public static void main(String[] args) {  
        UsoAnotacion miobjeto=new UsoAnotacion();  
        Interprete interprete=new Interprete();  
        interprete.process(miobjeto);  
    }  
}
```

```
public class Interprete{  
    public void process(UsoAnotacion prueba) {  
        try{  
            Method[] methods = prueba.getClass().getMethods();  
            for (Method method : methods) {  
                procesarMetodo(method, prueba);  
            }  
        } catch (final Exception e) {  
            System.err.println("Hubo un error:" + e.getMessage());  
        }  
    }  
    private void procesarMetodo(Method method, UsoAnotacion prueba)  
        throws IllegalAccessException, IllegalArgumentException, InvocationTargetException {  
        Log log = method.getAnnotation(Log.class);  
        //si el método incluye la anotación Log, comprueba el atributo level  
        //y llama al método con el texto que corresponda a ese nivel  
        if (log != null) {  
            final Valores level = log.level();  
            switch(level) {  
                case UNO:  
                    method.invoke(prueba, "mensaje simple");  
                    break;  
                case DOS:  
                    method.invoke(prueba, "mensaje detallado");  
                    break;  
                case TRES:  
                    method.invoke(prueba, "mensaje simple a las "+LocalDate.now());  
                    break;  
            }  
        }  
    }  
}
```

Metaanotación @Repetable

- Permite que una anotación pueda aplicarse más de una vez sobre el elemento.
- Además de la anotación principal, se debe crear otra que incluya un array de objetos anotación:

```
@Repetable(Autores.class)
public @interface Autor{
    int id() default 0;
    String value();
}
```

```
public @interface Autores{
    Autor[] value();
}
```

- Para utilizarla:

```
@Autor(...)
@Autor(...)
class MiClase{
    ..
}
```

O

```
@Autores({@Autor(...), @Autor(...)})
class MiClase{
}
```


Anotación @SuppressWarnings

➤ Anotación especial para eliminar avisos en código fuente.


➤ Su definición es:

```
@Target(value={TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})  
@Retention(value=SOURCE)  
public @interface SuppressWarnings{  
    String[] value;  
}
```

➤ Entre los posibles valores de *value*:

- unchecked. Se suprimen avisos de código inseguro
- deprecation. Se suprimen avisos de código deprecado
- unused. Se suprimen avisos de elementos no utilizados

Otras anotaciones especiales

- **@Override.** Se utiliza delante de un método de instancia para indicar que dicho método está siendo sobrescrito. Es utilizada por el compilador
 - **@Deprecated.** Se utiliza para indicar que una clase, atributo o método está deprecated y no se recomienda su uso. Es utilizada en tiempo de ejecución
 - **@SafeVarargs.** Utilizada sobre métodos y constructores para afirmar que el parámetro varargs no realiza operaciones potencialmente inseguras
- 

Revisión conceptos

Indica cuál de las siguientes afirmaciones es correcta:

- a. `@Override` y `@Repeatable` son metaanotaciones
- b. `@SuppressWarnings` solo puede utilizarse con atributos y métodos
- c. La anotación `@Override` es evaluada en tiempo de ejecución
- d. El valor del atributo de una anotación no puede ser de tipo fecha

Respuesta

- a. Incorrecta. `@Override` no es metaanotación
- b. Incorrecta. Puede utilizarse con todo tipo de elementos
- c. Incorrecta. Es evaluada por el compilador
- d. Correcta. Solo se admiten primitivos, String y enumerados