Annotazioni

Le annotazioni sono una forma di metadati

in quanto forniscono informazioni su un programma.

Le annotazioni non influiscono direttamente sulla semantica del programma

ma influiscono sul modo in cui i programmi vengono trattati da **strumenti** e **librerie**, che a loro volta possono influenzare la semantica del programma in esecuzione.

Le *Java annotation* sono state introdotte nella release 5 (Tiger). A partire dalla versione Java SE 8, le annotazioni possono essere applicate non solo alle dichiarazioni (come accadeva prima) ma in ogni situazione in cui sia un tipo (si veda *Java Type Annotations*) allo scopo di realizzare uno più forte controllo di tipo (si pensi ad esempio a *@NonNull*).

Scopo delle annotazioni

Oggi le annotazioni si trovano quasi ovunque nei programmi e sono introdotte per diversi scopi:

- per aiutare a meglio comprenderne l'intento a vari elementi di una classe;
- per permettere ulteriori controlli in fase di compilazione che garantiscono il rispetto di vari vincoli;
- per dare supporto ad analisi aggiuntiva del codice tramite strumenti sensibili alle annotazioni;
- per arricchire lo spazio concettuale del linguaggio (in ottica Declarative Programming)

Le annotazioni possono essere elaborate in fase di compilazione e/o in fase di esecuzione, sfruttando in questo caso le API di *Java Reflection*, con possibile impatto sulle prestazioni, se non usate con attenzione.

Forse il più grande vantaggio delle annotazioni è dare supporto a un **paradigma di progettazione basato su configurazione esplicita**, il che permette di semplificare diversi aspetti della configurazione, con grande impatto sul processo di sviluppo.

Framework molto diffusi che sfruttano questo aspetto sono *Spring and Spring Boot* che useremo più avanti.

Per il momento ci limitiamo a illustrare i meccanismi Java che permettono di sfruttare le annotazioni nelle nostre applicazioni.

Meta-Annotation in Java

Per specificare il comportamento delle Annotation, Java introduce altre Annotation, che vengono definite **Meta-Annotation**.

<table>
<tr><td>@Target</td><td>permette di definire a quale parte del codice può essere collegata l’Annotation.<table>
<tr><td>ElementType.PACKAGE</td><td>Si applica alla definizione del package</td></tr>
<tr><td>ElementType.TYPE</td><td>Si applica alla definizione di classi, interfacce ed enumeration</td></tr>
<tr><td>ElementType.FIELD</td><td>Si applica agli attributi</td></tr>
<tr><td>ElementType.METHOD</td><td>Si applica ai metodi</td></tr>
<tr><td>ElementType.PARAMETER</td><td>Si applica ai parametri dei metodi</td></tr>
<tr><td>ElementType.CONSTRUCTOR</td><td>Si applica al costruttore</td></tr>
<tr><td>ElementType.LOCAL_VARIABLE</td><td>Si applica ad una variabile locale</td></tr>
</table></td></tr>
<tr><td>@Retention</td><td>specifica come saranno visibili le informazioni collegate all’Annotation.</td></tr>
<tr><td>@Documented</td><td>serve per includere l’Annotation nel javadoc, visto che per default sono escluse.</td></tr>
<tr><td>@Inherited</td><td>una classe che utilizza l’Annotation, la fa ereditare anche alle classi figlie.</td></tr>
</table>

Retention

- `@Retention(RetentionPolicy.SOURCE)` : Annotation non viene letta dal compilatore e non memorizzata nel bytecode (.class file). Sarà quindi ignorata dalla JVM

- @Retention(RetentionPolicy.CLASS): è il default. Annotation viene registrata nel bytecode dal compilatore, ma verrà ignorata dalla JVM quando la classe verrà caricata (a runtime)
- @Retention(RetentionPolicy.RUNTIME): Annotation viene registrata nel bytecode e potrà esser letta a runtime (mediante reflection) quando la JVM caricherà la classe.

Annotazioni: esempio

Definizione di annotazioni custom

Annotation: definizione

È possibile utilizzare l'annotazione *@Interface* per descrivere la propria definizione di annotazione.

Le annotazioni prendono la forma di una dichiarazione di interfaccia con un @ che le precede e opzionalmente marcate con una meta-annotazione.

- Ogni nuovo tipo estende automaticamente l'interfaccia *java.lang.annotation.Annotation*.
- La dichiarazione di un metodo corrisponde a un elemento dell'annotazione.
- I tipi di ritorno dei metodi devono essere: tipi primitivi oppure **String**, **Class**, **enum**, tipi di annotation o array dei tipi elencati.

Annotation: esempio di definizione

Uso di annotazioni custom

Una annotazione precede la classe, il metodo o l'attributo che si vuole annotare ed strutturata come un insieme di coppie *nome=valore*.

Come esempio, definiamo una annotazione per descrivere il modo di accedere ad una applicazione (il codice si trova in **unibo.actors23.annotations.example** del progetto **unibo.actors23**):

```
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface AccessSpec {
```

```
    enum issProtocol {UDP,TCP,HTTP,MQTT,COAP,WS} ;
    issProtocol protocol() default issProtocol.TCP;
    String url() default "unknown";
}
```

La meta-annotation **@Retention** dice che l'annotazione **@AccessSpec** è visibile a runtime.

L'annotazione permette di specificare due attributi:

- il tipo di protocollo (metodo **protocol()**), con default TCP
- un URL, come ad esempio **"http://localHost:8090/api/move"**

Annotation: esempio di uso

Introduciamo una classe che introduce l'annotazione **@AccessSpec**.

```
@AccessSpec(
    protocol   = AccessSpec.issProtocol.HTTP,
    url        = "http://localHost:8090/api/move"
    //, configFile ="myfile.txt"
)
public class AnnotationUsageDemo {
      ...
  public static void main( String[] args) {new AnnotationUsageDemo();}
}
```

Elaborazione di annotazioni custom

Le API di *Java Reflection* permettono di recuperare a le informazioni che abbiamo inserito tramite l'annotazione meta-annotata con @Retention(RetentionPolicy.RUNTIME).

```
public class AnnotationUsageDemo {

public void readProtocolAnnotation(Object element) { ...}

  public AnnotationUsageDemo() {
    readProtocolAnnotation( this );
  }

}
```

readProtocolAnnotation

Il metodo **readProtocolAnnotation** viene definito in modo da accedere dinamicamente (usando le API di *Java Reflection*) alle informazioni introdotte nell'annotazione per visualizzarle.

```
public static void readProtocolAnnotation(Object element) {
try {
  Class<?> clazz            = element.getClass();
  Annotation[] annotations  = clazz.getAnnotations();
  for (Annotation annot : annotations) {
    if (annot instanceof AccessSpec) {
      AccessSpec p  = (AccessSpec) annot;
      CommUtils.outblue("Tipo del protocollo: " + p.protocol());
      CommUtils.outblue("Url del protocollo:  " + p.url());
      CommUtils.outblue("Configuration file: "  + p.configFile());
      String v = getHostAddr( "(\\w*)://([a-zA-Z]*):(\\d*)/(\\w*)/(\\w*)", p.url());
      CommUtils.outblue("hostAddr=" + v);
    }
  }
} catch (Exception e) {... }
}
```

Il metodo **getHostAddr** estrae la parte *host:port* dall'URL, usando *Pattern matching* su espressioni regolari:

```
public static String getHostAddr(String functor, String line){
    Pattern pattern = Pattern.compile(functor);
    Matcher matcher = pattern.matcher(line);
    CommUtils.outblue("line: " + line);
    String content = null;
    if( matcher.find()) {
        for( int i = 1; i<=5; i++ ) {
            CommUtils.outblue("group " + i + ":" + matcher.group(i));
        }
        content = matcher.group(2)+":"+matcher.group(3);
     }
    return content;
}
```

L'output del programma è il seguente:

```
Tipo del protocollo: HTTP
Url del protocollo:  http://localHost:8090/api/move
Configuration file: ProtocolConfig.txt
getHostAddr | line: http://localHost:8090/api/move
getHostAddr | group 1:http
getHostAddr | group 2:localHost
getHostAddr | group 3:8090
getHostAddr | group 4:api
getHostAddr | group 5:move
hostAddr= localHost:8090
```