Example10

Table of contents

Dynamic Proxies

Nell'esempio Example09 abbiamo visto l'invocazione dinamica dei metodi, la possibilita' di usare il package `java.lang.Reflect` per, non solo descrivere le classi di cui conosciamo i descrittori, ma anche per invocare codice.

Se abbiamo a disposizione un insieme d'interfacce, che definiscono un insieme di metodi da poter invocare, non e' necessario avere a disposizione una classe che implementi queste, per avere oggetti.

Dato un array di descrittori a, un *dynamic proxy* e' un oggetto che implementa l'interfaccia in a e invoca il codice utente se necessario. Viene costruito un oggetto nell'interfaccia ma non nell'implementazione.

`it.unipr.informatica.examples.model`

`Student.java`

L'interfaccia dello studente ci dice che tutto quello che risponde alle caratteristiche di uno studente, e' in grado di ricevere 3 messaggi:

- `getID()`;
- `getName()`;
- `getSurname()`.

Nessuno di questi ha argomenti, non viene previsto che chi invia messaggio, si aspetti di ricevere argomenti validi. Tramite l'interfaccia, possiamo costruire oggetti chiamati *proxy* (programmazione distribuita) che ricevono i messaggi elencati.

```
package it.unipr.informatica.examples.model;

public interface Student {
        public int getID();
        public String getName();
        public String getSurname();
}
```

`it.unipr.informatica.examples`

`Example10.java`

Il nostro esempio costruisce un oggetto che risponde ai messaggi dell'interfaccia studente dando sempre le stesse risposte. E' un esempio di base ma se non altro vediamo di capire come funzionano le cose.

Viene costruito un oggetto il cui riferimento si chiama `student`.
Siccome e' d'interfaccia `Student`, quindi risponde ai messaggi elencati nell'interfaccia. Notare che l'interfaccia e' il tipo di dato.
Una volta che abbiamo l'oggetto, stampiamo il suo nome, il suo cognome, l'ID.

`newPoxyInstance()` permette di costruire un proxy che andra' a implementare le interfacce che andremo a elencare e utilizzera' un metodo di *dispatch* che riceve i messaggi e decide cosa fare per produrre risposte:

- `getClass().getClassLoader()` ci permette, partendo dal descrittore di classe, di ottenere l'identificatore univoco di questa;
- `Class<?>[]` sarebbe l'array contenente i descrittori delle interfacce che vogliamo vengano implementate da questo oggetto, che sarebbe `Student.class`, descrittore della classe `Student`;
- passiamo istanza dell'*invokation handler* tramite l'unico metodo che la implementa, `this::handler` si occupa di ricezione del messaggio.

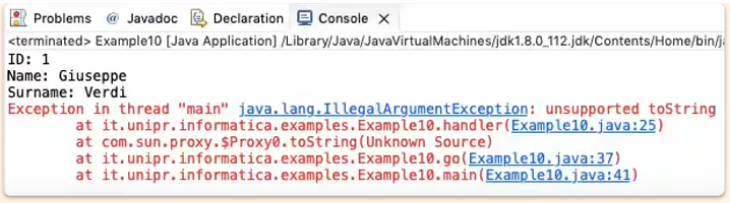
`handler` viene utilizzato tutte le volte che mandiamo un messaggio.
Tutte le volte che inviamo un messaggio riferito allo `student`, succede che il messaggio passa all'invokation handler.

```
        // ...
        private void go() {
                Student student = (Student) Proxy.newProxyInstance
                        (getClass().getClassLoader(),
                        new Class<?>[] { Student.class }, this::handler);
                System.out.println("ID: " + student.getID());
                System.out.println("Name: " + student.getName());
                System.out.println("Surname: " + student.getSurname());
                System.out.println();
        }
        // ...
```

Il `main()` fa quello che fa sempre, crea `Example10` ed esegue la sua `go()`.

```
        // ...
        public static void main(String[] args) {
                new Example10().go();
        }
}
```

> ⚠ `student` **non e' un oggetto fatto particolarmente bene**
>
> `System.out.println(student.toString())` non possiamo farlo, siccome sta a noi se implementare o meno i metodi di `Object`.




```java
package it.unipr.informatica.examples;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import it.unipr.informatica.examples.model.Student;

public class Example10 {
        private Object handler
                (Object proxy, Method method, Object[] arguments)
                        throws Throwable {
                String methodName = method.getName();
                switch (methodName) {
                case "getID":
                        return 1;
                case "getSurname":
                        return "Verdi";
                case "getName":
                        return "Giuseppe";
                default:
                        throw new IllegalArgumentException
                                ("unsupported " + method.getName());
                }
        }
        // ...
```

Siccome adesso non ho piu' la classe, dove va messo lo stato dell'oggetto?
Possiamo metterlo dove vogliamo, ad esempio in qualche oggetto, da qualche parte nello stack (non permanente), basta ci sia un posto.
Noi useremo l'invokation handler come posto per lo stato, che non e' una soluzione bellissima siccome tutti quelli che lo hanno, avranno stato uguale.