Onu Eduard Gabiel

Gr 244

Referat java reflection si aplicatii

1. Introducere

Uneori aplicatiile complexe necesita un control absolout asupra fiecarei componente ce o alcatuieste, deseori fiind nevoie sa se modifice comportamentul acestora la runtime. In limabjele de programare low-level nu exista o astfel de metoda, insa situatia sta cu totul altfel cand vorbim de limbajele de programare de nivel inalt.

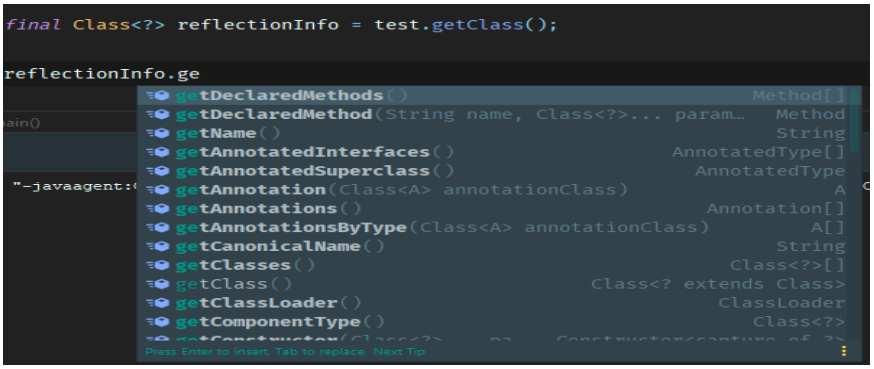
Reflection (reflectia) este o metoda ce este utilizata pentru examina sau modifica comportamentului claselor, metodelor sau interfetelor la runtime (momentul rularii). Aceast API poate fi folosit si pentru a usura procesul de dezvoltare al aplicatiilor, prin completarea automata a codului cu alte parti deja auto-generate sau prin usurarea modului in care unele componente sunt create, folosite, serializate sau stocate (Dependency injection, Proxy, JSON, XML, JDBC).

In Java acest feature a fost introdus inca de la inceputuri (**java.lang.reflect**), dar incepand cu versiunea a 8 a libajului de programare aceasta a fost radical imbunatatita, fiindu-i adaugate noi functionalitati.

2. Exemplificarea tehnologiei

In Java, asa cum bine stim, ficare obiect mostenste din clasa Object si datorita acestui fapt acesta are by default o serie de metode ce pot fi folosite, printre care si metoda **getClass().**

Aceasta metoda este punctul de start al reflectiei din Java, doarece in urma apelului acestei metode este intors un object de tipul Class<?>. Acest obiect, asa cum este ilustrat in imaginea urmatoare, are o serie de metode, ce sunt folosite pentru a “reflecta” continutul clasei la runtime.



Mai jos este defita clasa pe care vom dori sa o reflectam:

```
class TestClass {

    final private String field1;
    final private String field2;

    public TestClass(final String field1, final String field2) {
        this.field1 = field1;
        this.field2 = field2;
    }

    public void methodA() {
        System.out.println(field1);
    }

    public void methodB() {
        System.out.println(field2);
    }

    public void methodC(final List<Integer> arg0) {
        arg0.forEach(System.out::println);
    }

    private void hiddenMethod() {
        System.out.println("I am a private method");
    }
}
```

a) Reflectia metodelor

- Putem afla informatii despre metodele dintr-o clasa si le putem apela pe un anumit obiect sau putem sa le modificam mofificatorul de acces (o metoda private sau protected putand fi modificata astfel incat sa o putem accesa).

```
var test = new TestClass("field1", "filed2");

final Class<?> reflectionInfo = test.getClass();
final Method[] methods = reflectionInfo.getDeclaredMethods();
for(var method : methods) {
    System.out.println(method.getName() + " " + method.canAccess(test));
}
```

Executia acestei parti de cod va afisa pe consola numele metodelor declarate in clasa de mai sus si daca acestea pot fi apelate din afara clasei (adica daca nu sunt fie private fie protected).

- Asa cum am spus o metoda poate fi modificata, asfel incat sa poata fi apelata din afara clasei. Mai jos este o secventa de cod care modifica toate metodele private astfel incat sa poata fi apelate pe un anumit obiect la runtime.

```
var test = new TestClass("field1", "filed2");

final Class<?> reflectionInfo = test.getClass();
final Method[] methods = reflectionInfo.getDeclaredMethods();

for (var method : methods) {

    //if the method is accessible ignore it
    if(method.canAccess(test)) {
        continue;
    }

    //modify at runtime the method accessibility
    method.setAccessible(true);
    System.out.println(method.getName());

    //call the method on a specific object
    method.invoke(test);
}
```

Cu ajutorul metodei setAccessible(true) metoda privata acum este “transformata” in publica putand astfel sa fie apelata pe obiectul test.

Apelul method.invoke(test) este echivalent cu test.hiddenMethod() si are acelasi efect: se apeleaza metoda hiddenMethod(). Daca apelul setAccessible(true) nu ar fi fost efectuat inainte, rezultatul ar fi fost o exceptie.

- Putem afla informatii despre parametrii precum numele si tipul

```
final Method[] methods = reflectionInfo.getDeclaredMethods();
for (var method : methods) {
    final Parameter[] parameters = method.getParameters();
    for (var param : parameters) {
        System.out.println(param.getName() + " " + param.getType().getName());
    }
}
```

b) Reflectia constuctorilor

```
final Constructor<?>[] constructors = reflectionInfo.getConstructors();
for(var constructor : constructors) {
    System.out.println(constructor.getName());
    Arrays
            .asList(constructor.getParameters())
            .forEach(x -> System.out.println(x.getName() + " " + x.getType()));
```

c) Crearea de obiecte la runtime

Se alege constructorul a carei signatura respecta tipul parametrilor dati, dupa care se creeaza o noua instanta a clasei.

In exemplul urmator se alege si metoda ce sa se apeleze si se apeleaza pe obiectul creat

```
var cls = TestClass.class;
var test = cls
        .getConstructor(String.class, String.class)
        .newInstance( ...initargs: "field10", "filed20");

cls.getDeclaredMethod( name: "methodA").invoke(test);
cls.getDeclaredMethod( name: "methodB").invoke(test);
```

d) Reflectia fieldurilor

Se vor printa numele field-urilor impreuna cu valoare acestora

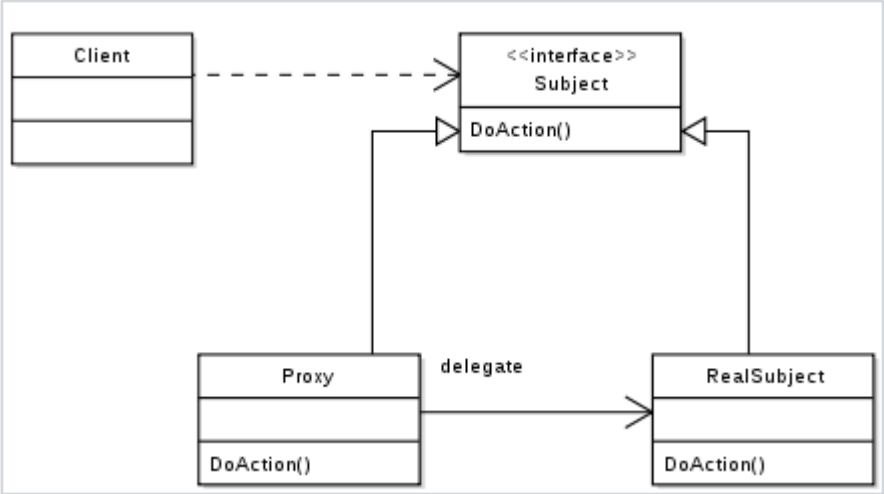
```
var cls = TestClass.class;
var test = cls
        .getConstructor(String.class, String.class)
        .newInstance( ...initargs: "field10", "filed20");

final Field[] fields = test.getClass().getDeclaredFields();
for (var field : fields) {
    field.setAccessible(true);
    System.out.println(field.getName() + " " + field.get(test));
    field.setAccessible(false);
}
```

3. Aplicatii

Reflectia este utilizata “behind the scenes” in mare parte dintre aplicatiile pe care le folosim sau scriem zilnic. De aceea am decis sa prezint 2 dintre cele mai inatlnite situatii in care ne confruntam cu reflectia.

a) Sablonul de proiectare Proxy

Un proxy este o clasa care functioneaza ca o interfata spre altceva. Acesta poate interfata spre orice (network connection, un fisier, un obiect foarte mare in memorie ce ar necesita timp pana cand s-ar incarca). Principiul dupa care functioneaza acest sablon este urmatorul: “in loc sa primesti obiectul in sine, primesti ceva ce se comporta ca obiectul dorit si prin intermediul acestuia se deleaga actiuni pe acel obiect”.



Multe dintre tehnologii folosesc acest proxy pentru a face diferite actiuni: Hibernate foloseste pentru persistenta, proxy (atunci cand se foloseste lasy fetch), in networking Java Rmi, Java Beans folosesc stubs + skeleton care defapt sunt proxy-uri.

Acest sablon de programare necesita un limbaj in care reflexia poate fi aplicata, deoarece este nevoie sa se creeze, la runtime, (implementari ale unor interfete) intr-un mod dinamic pe un anumit obiect.

In java pentru a putea crea un proxy putem folosi o clasa deja implementata pentru noi, si anume clasa Proxy (metodele statice pe aceasta clasa).

```java
var instance = Foo.class.getConstructor().newInstance();
var proxyInstance = (IFoo) Proxy.newProxyInstance(
        Thread.currentThread().getContextClassLoader(),
        instance.getClass().getInterfaces(),
        (proxy, method, methodArguments) -> {
            System.out.println("Proxy method");
            return method.invoke(instance, methodArguments);
        }
);

proxyInstance.doStuff();
```

```java
interface IFoo {
    void doStuff();
}


class Foo implements IFoo {

    public Foo() {
    }

    @Override
    public void doStuff() {
        System.out.println("I am doing stuff...");
    }
}
```

In imaginile anterioare am prezentat un mod prin care se pot creea proxy catre o anumita clasa, ce implementeaza o anumita interfata la runtime.

b) Serializare si deserializare de JSON si XML

a. JSON

Fara ajutorul reflectiei ar fi foarte greu ca de fiecare data sa construim noi din obiecte, JSON-uri (sub forma de string-uri) pentru a putea comunica intre client si server.Cu ajutorul reflectiei un serializator de obiecte sub forma de json, poate accesa datele dintr-un obiect si poate sa creeze-l serializeze si deserializeze in string(JSON) / repectiv din string-uri.

In java obiectul trebuie marcat ca serializabil, prin implementarea interfetei Serializable iar mai apoi putem specifica numele field-urilor din json(cu ajutorul adnotarii JsonProperty()) sau putem exclude anumite filed-uri din Json (JsonIgnore()). In java by default nu exista un serializator de json, trebuind sa fie importate librarii (jackson, gson, etc.). In figura urmatoare este

exemplificat un mod in care se poate serializa un obiect, cu ajutorul lui Jackson.

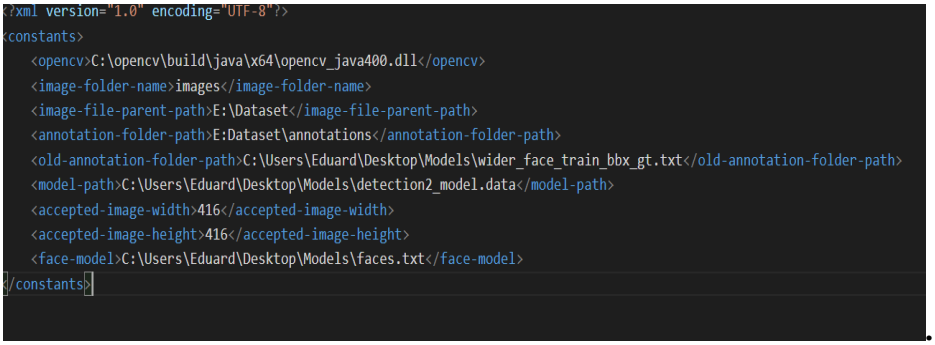
```java
public class User implements Serializable {

    @Id
    @JsonIgnore
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int userId;

    @Column
    @JsonIgnore
    private String password;

    @Column
    @JsonProperty("usern")
    private String username;

    public User() {
    }

}
```

b. XML

Cu ajutorul refectiei putem serializa/deserializa foarte usor XML-uri. Cu ajutorul unor librarii externe acest proces este foarte usor.

```java
@XmlRootElement
public class Constants {
```

```java
@XmlElement(name = "image-folder-name")
public void setImageFolderName(String imageFolderName) {
    this.imageFolderName = imageFolderName;
}

@XmlElement(
        name="image-file-parent-path"
)
public void setImageFileParentPath(String imageFileParentPath) {
    this.imageFileParentPath = imageFileParentPath;
}

@XmlElement(
        name = "annotation-folder-path"
)
public void setAnnotationFolderPath(String annotationFolderPath) {
    this.annotationFolderPath = annotationFolderPath;
}
```



4. Penalizarea de performanta

Cu toate ca reflectia ne “usureaza viata de programator”, asa cum bine stim, exista si dezavantaje ale acestei tehnologii, cea mai mare dintre acestea fiind penalizarea de perfomanta.

Reflectia adauga un nou strat de abstractizare in aplicatiile pe care noi le scriem, iar folosirea acesteia duce la incetinirea acesora, de aceea ne este recomandat sa folosim reflectia numai cand este nevoie si numai daca nu exista o alta alternativa de a obtine acelasi comportament/rezultat.

5. Implemetarea de cache generic pentru servicii folosind java reflection + proxy(demo)

6. Concluzii

Reflectia din Java este utilizata foarte des in programe ce au nevoie de abilitatea de a examina sau modifica comportamentul din momentul rularii al apllicatiilor ce sunt scrise in acest limbaj. Acesta tehnologie este destul de complicata si ar trebui folosita numai de programatori ce stiu foarte bine bazele limbajului de programare.

Aceasta este o tehnica foarte puternica care ofera capacitatea aplicatiilor sa efectueaze operatii ce altfel nu ar putea fi executate.