Programare Orientată pe Obiecte

Reflection

- Responsabil: Cosmin Boacă
- Data publicării: 05.01.2015
- Data ultimei modificări: 05.01.2015

Obiective

Scopul acestui laborator este familiarizarea studenţilor cu noţiunnile de **Reflection**, **Java Native Interface** și **Annotations**.

Java Native Interface (JNI)

Programele scrise in java **NU** rulează direct pe mașina fizică ci rulează într-o mașină virtuala cunoscută sub numele de **JVM (Java Virtual Machine)**. Din acest motiv codul java nu se compilează în cod mașină ci se compilează intr-un cod intermediar numit **bytecode** care este interpretat de mașina virtuală, aceasta fiind cea care este responsabilă de execuția codului. Acest lucru are o serie de avantaje și dezavantaje. Printre avantaje se numără :

- Portabilitatea (Codul compilat sub formă de bytecode poate fi rulat pe orice mașină pe care este este prezent un JVM, indiferent de arhitectura acesteia).
- Managementul erorilor (Operatiile ilegale făcute de program sunt semnalate prin aruncarea de excepții nu prin trimiterea de semnale de către sistemul de operare)

Principalul dezavantaj al rulării programulu intr-o mașină virtuală este performanța. Bytecode-ul trebuie interpretat/compilat (în realitate este o combinație) de către mașina virtuală care la rândul ei să genereze cod mașină, sau cod **nativ**, care se poate rula direct pe mașina fizică.

Din considerente de performanță și flexibilitate java permite executarea de **cod nativ** compilat în biblioteci partajate (shared libraries). Modalitatea de interfațare dintre codul java și codul nativ este numită **Java Native Interface** sau, pe scurt, **JNI**. Nu vom intra în detalii despre funcționarea internă a acestui mecanism, vom prezenta doar un exemplu de declarare de funcție nativă in java.

NativeMethodDeclaration.java

```
package laborator;

class NativeTest {
     private int nativeAdd(int num1, int num2);
}
```

Functia in C/C++ trebuie sa aiba urmatoarea definite

```
extern "C" {
Java_laborator_NativeTest_nativeAdd(JNIEnv *env, JObject obj, Jint a, Jint b) {
     return a + b;
}
}
```

Numele metodei in C derivă din numele fully qualified al metodei din Java. (nume_pachet.nume_clasa.nume_metoda). Pentru metoda de mai sus, acest nume este laborator.NativeTest.nativeAdd. Numele metodei din C se formeaza prin concatenarea Java_

la numele metodei în care . este inlocuit cu _ . Metoda va fi compilată într-o bibliotecă partajată (shared_library) cu linking de C (extern “C”), acest lucru fiind necesar pentru a putea fi găsită corect din Java.

Reflection

Reflection reprezintă un mecanism ce permite unui program să-și examineze și să-și modifice structura in timpul rulării. La baza API-ului pentru **Reflection** stau următoarele clase :

- `Class`
- `Constructor`
- `Method`
- `Field`
- `Annotation`

Fiecare dintre aceste clase abstractizează conceptul corespunzător numelui clasei. Implementarea internă a acestui mecanism este diferită de la un JDK la altul, dar în mare este bazată pe **metode native**.

```
private native Field[]       getDeclaredFields();
private native Method[]      getDeclaredMethods();
private native Constructor<T>[] getDeclaredConstructors();
private native Class<?>[]   getDeclaredClasses();
private native Annotation[] getDeclaredAnnotations();
```

Acestea sunt câteva dintre metodele ce vor fi folosite in laborator. Există numeroase moduri de a utiliza acest mecanism. Majoritatea framework-urilor scrise in Java sunt implementate folosind reflection. Un exemplu este chiar framework-ul Junit, care folosește reflection să se uite ce metode au adnotările `@Test`, `@Before`, `@After` in clasele definite de utilizator. Un alt exemplu de folosire al reflection este configurarea dinamica a unei aplicații folosind un fișier de configurare. În general se folosesc fisiere de configurare XML pentru că sunt ușor de scris și de citit. Să considerăm următoarele definiții de clase

```
class Server {
    private ITransporter transporter;
    private IStorage storage;
}

abstract class Transporter {
    private ISerializer serializer;
}

class Message {
    //Visitor
    String serialize(ISerializer serializer) {
        return serializer.serialize(this);
    }
}

class TCPTransporter extends Transporter;
class UDPTransporter extends Transporter;
class HTTPSTransporter extends Transporter;

class XMLSerializer implements ISerializer;
class JSONSerializer implements ISerializer;

class FileStorage implements IStorage;
class EncryptedFileStorage implements IStorage;
class ClouldStorage implements IStorage;
class EncryptedCloudStorage implements IStorage;
```

Aceste clase caracterizeaza un server rudimentar care admite diverse tipuri de conexiuni, de formate de mesaje si de stocare a mesajelor primite. Să zicem că vrem să livram acest server unui client pentru a îl folosi. Se poate observa destul de ușor că pentru a instanția un server trebuie instanțiate clase adiționale pentru Transporter, Serializer, Storage fiecare cu parametri specifici ceea ce implică mult cod scris care trebuie modificat oricând vrem să facem o schimbare. Acest lucru poate face un

framework de genul acesta să fie foarte greu de folosit. Soluția ar fi să folosim un fișier de configurare care să ascundă detaliile de implementare. Un fișier de configurare in format XML ar putea arăta cam așa :

```
<server>
    <component name="transporter" type="HTTPSTransporter">
        <component name="serializer" type="JSONSerializer" />
    </component>
    <component name="storage" type="CloudStorage">
    ...
</server>
```

Acest fișier va fi parsat primit ca parametru de o clasă care îl va parsa și va returna un server. Avantajul este că instanțierea unui server este foarte simplă chiar dacă parserul fisierului de XML este destul de greu de scris.

Adnotări (Annotations)

Adnotările sunt o formă de a încorpora **metadata** în codul Java. Ele pot fi folosite la compilare sau pot persista și la runtime. În cadrul acestui laborator vom folosi doar adnotări care persistă la runtime și care pot fi aplicate metodelor unei clase. O astfel de adnotare se declară în felul următor :

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

@Target(value = ElementType.METHOD)
@Retention(value = RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
}
```

Pentru mai multe detalii despre adnotări și despre analiza lor folosind **reflection** consultați tutorialul următor : http://www.vogella.com/tutorials/JavaAnnotations/article.html [http://www.vogella.com/tutorials/JavaAnnotations/article.html]

Exemple

AppTest.java

```
public class AppTest{

        private int counter;

        public void printIt(){
                System.out.println("printIt() no param");
        }

        public void printItString(String temp){
                System.out.println("printIt() with param String : " + temp);
        }

        public void printItInt(int temp){
                System.out.println("printIt() with param int : " + temp);
        }

        public void setCounter(int counter){
                this.counter = counter;
                System.out.println("setCounter() set counter to : " + counter);
        }

        public void printCounter(){
                System.out.println("printCounter() : " + this.counter);
        }
}
```

ReflectApp.java

```
import java.lang.reflect.Method;

public class ReflectApp{

        public static void main(String[] args) {

        //no paramater
        Class noparams[] = {};

        //String parameter
        Class[] paramString = new Class[1];
        paramString[0] = String.class;

        //int parameter
        Class[] paramInt = new Class[1];
        paramInt[0] = Integer.TYPE;

        try{
                //load the AppTest at runtime
                Class cls = Class.forName("AppTest");
                Object obj = cls.newInstance();

                //call the printIt method
                Method method = cls.getDeclaredMethod("printIt", noparams);
                method.invoke(obj, null);

                //call the printItString method, pass a String param
                method = cls.getDeclaredMethod("printItString", paramString);
                method.invoke(obj, new String("test"));

                //call the printItInt method, pass a int param
                method = cls.getDeclaredMethod("printItInt", paramInt);
                method.invoke(obj, 123);

                //call the setCounter method, pass a int param
                method = cls.getDeclaredMethod("setCounter", paramInt);
                method.invoke(obj, 999);

                //call the printCounter method
                method = cls.getDeclaredMethod("printCounter", noparams);
                method.invoke(obj, null);

        }catch(Exception ex){
                ex.printStackTrace();
        }
   }
}
```

Exerciții

1. **(2p)** Creati clasa `ReflectionDummy`, o clasă care va conține 3 membri de tipul int, unul private, unul protected și unul public. Creați un obiect din această clasă și setați valorile membrilor folosind API-ul pentru Reflection. Folosiți metoda `getDeclaredFields` din clasa `Class`, și metodele `setAccessible` respectiv `set` din clasa `Field`.
2. **(10p)** Acest task va urmări crearea unei clase ce poate rula teste având o funcționalitate de bază asemănătoare cu aceea a framework-ului Junit. Se vor folosi metodela `Modifier.isPublic` a clasei `Modifier`, cât și metodele `getGenericReturnType`, `getGenericParameterTypes`, `getAnnotations` ale clasei `Method`. Puteți folosi orice alte metode considerați din API.
    - **(2p)** Definirea adnotărilor `@Test`, `@Before`, `@After`.
    - **(2p)** Identificarea metodelor cu adnotarea `@Test`. Toate aceste metode trebuie să aibă tipul de retur `boolean` și să nu primească parametri. (Vor returna true dacă testul trece sau false daca pică). Programul trebuie să arunce excepție dacă există vreo metodă cu adnotarea `@Test` care nu respectă semnătura și trebuie să ia în considerare doar metodele publice.
    - **(2p)** Identificarea metodelor cu adnotarea `@Before` înainte de fiecare test. Metodele trebuie să returneze void și să nu primească parametri. Programul va verifica acest lucru și va lua în considerare doar metodele publice.

- **(2p)** Identificarea metodelor cu adnotarea `@After` după fiecare test. Aceleași mențiuni ca si metodele cu adnotarea `@Before`.
- **(2p)** Implementarea unei clase `TestRunner` care conține o metodă statică run care primeste ca parametru un String reprezentând numele unei clase ce conține teste. Această clasă va rula testele din clasa primită ca parametru precum și metodele de setup respectiv tear down și va afișa o statistică sub forma :

```
numeMetoda1 failed/passed/error
numeMetoda2 failed/passed/error
...
numeMetodaN failed/passed/error
passed : numărTesteTrecute
failed : numărTesteEșuate
errors : numărTesteEronate
```
