Software Platforms

LM in Computer Engineering

Massimo Maresca

Annotations, Reflection and Dynamic Class Loading

- Annotations, Dynamic Class Loading and Reflection are core elements of Software Platforms, as Software Platforms require the integration of software modules/components implemented by third parties after platform deployment.
- Let us start from a short history of languages, compilation and programming models …
  - Machine Language: Numeric Instruction Codes and Memory Addresses
  - Assembly Language: Symbolic Codes associated to Instruction Codes and Memory Addresses
  - 1st Generation Compiler (e.g., Fortran): Static Variables, formula parsing and Machine Code Generation.
  - Advanced Compilation: Abstract Data Types, Type Checking, Records, Dynamic Data structures, etc.: Pascal, C
  - Object Oriented Programming: From Record to Class, Dynamic Class Loading, Reflection: C++, C#, Java, etc.

Annotations

- *Annotation is Metadata*. Metadata is data about data. So Annotations are metadata for code.
- Let us look at an example :

```
@Override
public String toString() {
return "This is String Representation of current object.";
}
```

- This code works even in absence of the @Override Annotation. The Annotation just tells the compiler that toString() is overriding another method so that if this is not the case the compiler may throw an error. So it is just an indication, i.e., metadata.

Ref.: https://dzone.com/articles/how-annotations-work-java

From Pre-processing Directives to Java Annotations

- C Language includes pre-processing directives (CPP):
  - e.g., # define, # include
- Can Annotations be considered a kind of pre-processing directives ? Yes and no. They can be both used in compilation and kept for successive utilization.
- It all depends on the Retention Policy adopted.
  - **RetentionPolicy.SOURCE** – Discard during the compile. These annotations don't make any sense after compilation has completed, so they aren't written to the bytecode. Examples @Override, @SuppressWarnings
  - **RetentionPolicy.RUNTIME** – Do not discard. The annotation should be available at runtime. Why is it useful ? Reflection is the answer.

Example of run-time Annotation

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

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
        String name();
        String value();
}

@MyAnnotation (name = "MyName", value = "MyValue")
…
<Rest of the Code>
…
```

Reflection

chi scrive codice deve fornire delle annotazioni che poi il deploiatore del codice dovre seguire per sfruttarre al massimo il codice. é particolarmente importante in applicazioni formate da piu pezzi magari mantenuta da piu persona , sfruttando i metadati si puo avere una sorta di comunicazione tra le parti della nostra applicazioni.

- Reflection refers to the ability of a program to analyze its code and/or the other programs.

REFLECTIONS=il codice deve capire cosa fa : classi metodi e annotazioni

- Reflection allows programs to analyze code elements which are available at run time, i.e., written “after” the program that perform the analysis.

tutti gli host di servizi richiedo che tu metta annotazioni

- Reflections is concerned with: Class, Method, Constructor, Field, Type, Annotation
- Rif. https://docs.oracle.com/javase/tutorial/reflect/

Reflection Example (1)

```
        Class c = Class.forName(args[0]);
// Get Declared Methods
        Method methodlist[] = c.getDeclaredMethods();
// For each method print name, types of parameters, types of exceptions
        for (int i = 0; i < methodlist.length; i++) {
          Method method = methodlist[i];
          System.out.println("Method Name = " + method.getName());
          Class parameterTypes[] = method.getParameterTypes();
          for (int j = 0; j < parameterTypes.length; j++)
                  System.out.println("Type of Parameter #" + j + " " + parameterTypes[j]);
          Class exceptionTypes[] = method.getExceptionTypes();
          for (int j = 0; j < exceptionTypes.length; j++)
                  System.out.println("Type of Exception #" + j + " " + exceptionTypes[j]);
          System.out.println("return type = " + method.getReturnType());
          System.out.println("-----");
        }
```

Reflection Example (2)

```
// Get Constructors
        Constructor constructorList[] = c.getDeclaredConstructors();
// For each constructuor print name, types of parameters, types of exceptions
        for (int i = 0; i < constructorList.length; i++) {
           Constructor constructor = constructorList[i];
           System.out.println("Constructor Name = " + constructor.getName());
           Class parameterTypes[] = constructor.getParameterTypes();
           for (int j = 0; j < parameterTypes.length; j++)
               System.out.println("Type of Parameter #" + j + " " + parameterTypes[j]);
           Class exceptionTypes[] = constructor.getExceptionTypes();
           for (int j = 0; j < exceptionTypes.length; j++)
               System.out.println("Type of Exception #" + j + " " + exceptionTypes[j]);
           System.out.println("-----");
        }
```

Reflection Example (3)

```
// Get Fields
        Field fieldList[] = c.getDeclaredFields();
// For each field print name, type, modifiers
        for (int i = 0; i < fieldList.length; i++) {
          Field field = fieldList[i];
          System.out.println("Field Name = " + field.getName());
          System.out.println("Field Type = " + field.getType());
          int fieldModifiers = field.getModifiers();
          System.out.println("Field Modifiers = " + Modifier.toString(fieldModifiers));
          System.out.println("-----");
        }
```

Dynamic Class Loading

```
public void createObject() {
// Regular object creation
        B b1 = null;
        b1 = new B();
        b1.doSomething();
}
```

Implicit class loading

```
public void getObjectFromFileSystem() {
// Object creation through class loading
        B b2 = null;
        ClassLoader cl = A.class.getClassLoader();
        try {
                Class<?> BClass = cl.loadClass("B");
                Object o = BClass.newInstance();
                b2 = (B) o;
        }
        catch (Exception e) {
                System.out.println("Error");
        }
                b2.doSomething();
        }
```

Explicit class loading

Dynamic Class Loading in a Sw Platform (1)

Interface Definition

```
interface ClassToBeLoadedInterface {
        void set_value (int value);
        int get_value();
        public void print_value(String param);
}
```

Interface used both by the SW Platform and by application developers

instanceof

```
……
    Object object = myClass.newInstance();
    System.out.println("object istanceof ClassToBeLoadedInterface:" + (object instanceof
ClassToBeLoadedInterface));
    if (object instanceof ClassToBeLoadedInterface){
        ClassToBeLoadedInterface myObject = (ClassToBeLoadedInterface) object;
     } else {
        System.out.println ("Error: Class Loaded is not an instance of ClassToBeLoadedInterface");
     }
…..
```

Dynamic Class Loading in a Software Platform (2)

```
public class ClassToBeLoaded implements ClassToBeLoadedInterface{
        private int value = 0;
        public void set_value(int value) {
                this.value=value;
        }
        public int get_value (){
                return this.value;
        }
        public void print_value(String string) {
                System.out.println (string+" "+ this.value);
        }
}
```

Example of an application to be loaded in a SW Platform

Reflection and Dynamic Class Loading: Programs

Program Examples:

- Reflection
  - DynamicClassLoading/Reflection

- Dynamic Class Loading
  - DynamicClassLoading/ClassLoader-New-vs-FileSystem
  - DynamicClassLoading/ClassLoader-ApplicationServer