Report

No .3

By Amir Lotfi

Contents

Java Reflection

زبانهای برنامه نویسی مدرن، داده های زیادی را بهمراه برنامه در حافظه ذخیره میکنند. این داده ها اغلب Meta Data نامگذاری میشوند. مثلاً در زبان سی، رشته ها به یک کاراکتر خاص ختم میشدند و برنامه نویس باید به واسطه حقله ها و شمارش اعضا، به اندازه کل رشته دسترسی پیدا کند، برعکس در زبان جاوا رشته ها بهمراه طول خود در حافظه ذخیره میشوند. در اینجا طول حکم یک Meta Data را دارد. مثال بعدی، اعلانهای دسترسی به متغیرها و توابع سطح شی است که به سه دسته public, private & protected تقسیم میشوند؛ این سه اعلان نیز در حافظه بهمراه کلاس ذخیره میشود. اینها نیز جزوی از Meta Data هستند.

در Reflection، در زمان اجرا، به این داده ها دسترسی پیدا میکنیم.

```
@Deprecated
public class Student
{
    private Integer id;
    private String  name;
    private String  surname;
    private String function(String arg1, String arg2, Integer arg3)
    {
        // TODO some stuff here!
        return "I am kinda private, so you cannot access me!";
    }

    public Integer getId()
    {
        return id;
    }

    public void setId(Integer id)
    {
        this.id = id;
    }

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public String getSurname()
    {
        return surname;
    }

    public void setSurname(String surname)
    {
        this.surname = surname;
    }
}
```

کد زیر، تست کلاس Student است که یک سری از خصوصیتهای Reflection را نمایش میدهد.

```
public class JavaTestStudentReflection
{
    private static Student                  student;
    private static Class<? extends Student> studentClass;

    @BeforeAll
    public static void initialize()
    {
        student = new Student();

        student.setId(0);
        student.setName("student#name");

        studentClass = student.getClass();
    }


    @Test
    public void testAnnotations()
    {
        Annotation[] annotations = studentClass.getAnnotations();
        Assertions.assertEquals(annotations.length, 1);
        Assertions.assertEquals(annotations[0].annotationType().getName(), Deprecated.class.getName());
    }

    @Test
    public void testFields() throws Exception
    {
        Field[] declaredFields = studentClass.getDeclaredFields();

        Assertions.assertEquals(declaredFields.length, 2);

        Assertions.assertEquals(studentClass.getDeclaredField("id").getGenericType().getTypeName(), Integer.class.getName());
        Assertions.assertEquals(studentClass.getDeclaredField("name").getGenericType().getTypeName(), String.class.getName());

        Arrays.stream(declaredFields).forEach(field -> Assertions.assertEquals(field.getModifiers(), Modifier.PRIVATE));
    }

    @Test
    public void testMethods() throws Exception
    {
        Method[] declaredMethods = studentClass.getDeclaredMethods();
        Assertions.assertEquals(declaredMethods.length, 5);

        Method method = studentClass.getDeclaredMethod("getName");
        String name = (String) method.invoke(student);
        Assertions.assertEquals(name, "student#name");

        method = studentClass.getDeclaredMethod("function", String.class, String.class, Integer.class);
        Assertions.assertEquals(method.getModifiers(), Modifier.PRIVATE);
        Assertions.assertEquals(method.isAccessible(), Boolean.FALSE);

        method.setAccessible(Boolean.TRUE);
        String privateFunction = (String) method.invoke(student, null, null, null);
        Assertions.assertEquals(privateFunction, "I am kinda private, so you cannot access me!");

        method = studentClass.getDeclaredMethod("setId", Integer.class);
        Assertions.assertEquals(student.getId(), 0);
        method.invoke(student, 1);
        Assertions.assertEquals(student.getId(), 1);

    }
}
```

Generic Service with CRUD Operation

با استفاده از Generic در جاوا میتوانیم کدهای تکراری عملیاتهای اساسی دیتابیس یعنی CRUD را یکبار بنویسیم و آنرا به واسطه ارثبری به سرویسهای سیستم اضافه کنیم.

```
class BaseEntity

class BaseRepository[E <: BaseEntity, K]
  extends Serializable

abstract class BaseService[E <: BaseEntity, R <: BaseRepository[E, K], K](var repository: R)
  extends Serializable
{
  def create(e: E): Unit= ???
  def read(k: K): E = ???
  def update(e: E): Unit = ???
  def delete(k: K): Unit = ???
}
```

طراحی بالا وابستگی شدیدی به بانک اطلاعاتی دارد؛ در صورتی که از کلیدهای جانشین استفاده کنیم، مقدار K اگر از روی همین کلیدها مقداردهی بشود عملاً بیمعنی خواهد بود و حتی منطق را Controller نیز تغییر میدهد. غیر از این روش بالا به خاطر تعیین سقف Repository ما را محدود به استفاده تنها از بانک اطلاعاتی رابطه ای میکند. فرضاً بخواهیم یک بانک جدید مبتنی بر NOSQL قرار بدهیم باید طراحی را از اول پیاده سازی کنیم. طراحی بالا شباهتی با الگوی Template Method دارد. روش دیگر استفاده از الگوی Strategy است.

```
class BaseEntity(var key : String)

trait CrudOperations[E <: BaseEntity, K]
{
  def create(e: E): E = ???
  def read(k: K): E = ???
  def update(e: E): E = ???
  def delete(k: K): E = ???
}

class JpaCrudOperations
  extends CrudOperations[BaseEntity, String]

class RedisCacheCrudOperations
  extends CrudOperations[BaseEntity, String]

class BaseRepository[E <: BaseEntity, K]
  extends Serializable

trait CrudService[E <: BaseEntity, K]
{
  def createEntity(e: E):E = ???
  def readEntity(k: K): E = ???
  def updateEntity(e: E): E = ???
  def deleteEntity(k: K): E = ???
}

abstract class BaseService[E <: BaseEntity, R <: BaseRepository[E, K], K]
(var repository: R, var operations: CrudOperations[E, K])
  extends CrudService[E, K] with Serializable
{

}
```

Invoking Setter/Getter using Reflection

فراخوانی متدها به دو روش صورت میگیرد، فرض میکنیم که کلاس زیر را داریم.

```
public class StudentBean
        implements Serializable
{
    private Integer                id;
    private String                 name;

    public StudentBean()
    {

    }

    public StudentBean(Integer id, String name)
    {
        this.id = id;
        this.name = name;
    }

    public Integer getId()
    {
        return id;
    }

    public void setId(Integer id)
    {
        this.id = id;
    }

    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }

}
```

با استفاده از پکیج Reflection نتیجه زیر را داریم

```
public class StudentBeanReflection
{

    StudentBean studentBean = new StudentBean(0, "student#name");

    @Test
    public void invokeGetter() throws Exception
    {
        Method method = studentBean.getClass().getDeclaredMethod("getName");
        String name = (String) method.invoke(studentBean);
        Assertions.assertEquals(name, studentBean.getName());

        method = studentBean.getClass().getDeclaredMethod("getId");
        Integer id = (Integer) method.invoke(studentBean);
        Assertions.assertEquals(id, 0);
    }

    @Test
    public void invokeSetter() throws Exception
    {
        Method method = studentBean.getClass().getDeclaredMethod("setName", String.class);
        method.invoke(studentBean , "student#newName") ;
        Assertions.assertEquals(studentBean.getName() , "student#newName");

        method = studentBean.getClass().getDeclaredMethod("setId", Integer.class);
        method.invoke(studentBean , 1) ;
        Assertions.assertEquals(studentBean.getId() , 1);
    }
}
```

با پکیج java.beans.* نتیجه زیر را داریم

```
public class StudentBeanReflection
{

    StudentBean studentBean = new StudentBean(0, "student#name");

    @Test
    public void invokeGetter() throws Exception
    {
        PropertyDescriptor descriptor = new PropertyDescriptor("name", StudentBean.class);
        String name = (String) descriptor.getReadMethod().invoke(studentBean);
        Assertions.assertEquals(name, studentBean.getName());

        descriptor = new PropertyDescriptor("id", StudentBean.class);
        Integer id = (Integer) descriptor.getReadMethod().invoke(studentBean);
        Assertions.assertEquals(id, studentBean.getId());
    }

    @Test
    public void invokeSetter() throws Exception
    {
        PropertyDescriptor descriptor = new PropertyDescriptor("name", StudentBean.class);
        descriptor.getWriteMethod().invoke(studentBean, "student#newName");
        Assertions.assertEquals("student#newName", studentBean.getName());

        descriptor = new PropertyDescriptor("id", StudentBean.class);
        descriptor.getWriteMethod().invoke(studentBean, 1);
        Assertions.assertEquals(1 , studentBean.getId());
    }
}
```

SOLID Principles

۵ اصل که با رعایت آنها کدی طلایی داریم؛ قابلیت حمل بالا، تست، تغییر سریع و آسان از خصوصیتهای این کد طلایی هستند.

Single Responsibility

هر قطعه کد(کلاس) باید یک وظیفه را انجام دهد یا یک دلیل برای تغییر قطعه کد وجود دارد.

```
// not so good
class Authenticator
{
  def authenticate(): Unit = ???

  def report(): Unit = ???
}

// better
class Authenticator
{
  def authenticate(): Unit = ???
}

class Reporter
{
  def report(): Unit = ???
}
```

Open-Close

هر قطعه کد اعم از کلاس، ماژول، توابع و ... مجاز به گسترش و نه تغییر هستند.

```
trait CrudOperation
{
  def create(): Unit
  def read(): Unit
  def update(): Unit
  def delete(): Unit
}
// just need to extends the crud
trait BaseService
  extends CrudOperation
{
  def buyProduct() : Unit
}
```

Liskov Substitution

هر شی از کلاسی در یک سلسله مراتب با ارجاع والدین خود جابجا میشود و باید کار کند.

```
object Application
{
  def main(arg: Array[String]): Unit =
  {
    val l : List          = new List()
    val il: ImmutableList = new ImmutableList()

    var collection: Collection = l // seem work!
    collection = il //well! not now. we cannot invoke add in this, throw exception
  }
}

class Collection
{
  def add(): Unit =
  {

  }
}

class List
  extends Collection
{
  override def add(): Unit = super.add()
}

class ImmutableList
  extends Collection
{
  override def add(): Unit = throw new RuntimeException()
}
```

کد زیر اصل بالا را رعایت میکند.

```
object Application
{
  def main(arg: Array[String]): Unit =
  {
    val list                  = new MutableList
    val immutableList         = new ImmutableList
    //u can invoke size now and it work
    var collection: Collection = list
    collection = immutableList
  }
}

class Collection
{
  def size(): Int = ???
}

class MutableList
  extends Collection
{
  override def size(): Int = super.size()

  def add(): Unit = ???
}

class ImmutableList
  extends Collection
{
  override def size(): Int = super.size()

  def add(): Unit = throw new RuntimeException()
}
```

Interface Segregation

واسطها یا interfaces باید تنها نیازمندهایی را حمل کنند که کلاینت به آنها نیاز دارد، در صورتی که این نیازمندهای پیاده بشوند اما مورد استفاده قرار نگیرند، باید آنها را به واسطهای دیگر انتقال داد یا حذف کرد. به تعبیر دیگر، هر واسط باید اعمالی که نیازدارد را شبیه سازی کند و نه بیشتر از آن.

Dependency Inversion

هر قطعه از برنامه به قطعه های دیگر نیازمند است؛ در این صورت باید به انتزاع وابسته باشد و نه پیاده سازی. مثلاً در Java Spring یک سرویس باید به JpaRepository وابسته باشد که یک انتزاع از دیتابیس است و نه کلاسی خامِ وابسته به دیتابیس.