Introducción a la Programación

Tema 7. Programación reflexiva en Java

1. Programación Reflexiva

1.1 Introducción

Como hemos visto en temas anteriores un programa en Java se compone de un conjunto de clases e interfaces agrupados en paquetes. Una de las clases debe tener un método *main* que será el punto de inicio de la ejecución del programa.

Las clases e interfaces definen tipos entre los cuales se define, debido a las relaciones de herencia e implementación, relaciones de sub-tipado. En tiempo de ejecución la información sobre los diferentes tipos (su nombre, métodos, super-tipos, etc.) se guarda en objetos especiales de tipo *Class<T>.* La programación reflexiva, en Java, está construida mediante la manipulación de objetos, que son instancias de *Class<T>*, y que representan, como hemos dicho, los diferentes tipos del programa. Como primer objetivo la programación reflexiva pretende:

- Obtener en tiempo de ejecución los objetos que representan los tipos
- Consultar las propiedades de un tipo disponiendo de un objeto que lo represente

Más adelante veremos otros usos de la programación reflexiva.

Veamos, en primer lugar, cuales son las diferentes formas de obtener objetos que representan los tipos (instancias, por lo tanto, de la *Class<T>*). Partamos de la declaración:

```
Racional r = new RacionalImpl(3,4);
```

Estamos declarando la variable r de tipo Racional. Estamos usando dos tipos que habremos declarado previamente (posiblemente en un interface y una clase): *Racional* y *RacionalImpl*.
En tiempo de ejecución por cada clase e interface existe un objeto que mantiene las propiedades compartidas por todos los objetos que son instancia del tipo (las propiedades *static*). Una de esas propiedades viene representada por el atributo *class* que es *public*, *static* y de tipo *Class<T>.* Esto nos proporciona un primer camino para obtener instancias de objetos de tipo *Class<T>:*

```
import algebra.*;
Class<Racional> tr = algebra.Racional.class;
Class<RacionalImpl> tri = algebra.RacionalImpl.class;
Class<PuntoImpl> tp = geometria.PuntoImpl.class;
```

Un objeto *Class<?>* puede representar a un tipo cualquiera. La clase *Class* posee el método:

```
public static Class<?> forName(String nT)
```

forName devuelve el representante del tipo de nombre cualificado *nT.* Donde por nombre cualificado queremos indicar una cadena compuesta del nombre del paquete donde está definido el tipo seguido de punto y el nombre del tipo.

```
Class<?> rCR = Class.forName("algebra.RacionalImpl");
```

Además todo objeto puede obtener el representante de su clase mediante el método *getClass()* que hereda de *Object* y cuya signatura es:

```
public Class<?> getClass();
```

Por lo tanto si hemos declarado la variable r como en el ejmplo de arriba podemos obtener el objeto que representa su tipo como:

```
Class<?> tr = r.getClass();
```

Tenemos por lo tanto tres vías para obtener objetos que representen tipos en tiempo de ejecución:

- Consultando el atributo *class* del objeto que contiene las propiedades estáticas del tipo en tiempo de ejecución (Racional.class, RacionalImpl.class, …)
- Invocando el método estático *forName* de *Class<T>* con un atributo que sea el nombre cualificado del tipo
- Invocando sobre un objeto el método *getClass().* Todos los objetos tienen disponible este método al heredarlo de *Object*.

1.2 La clase Class<T>

Como hemos visto la clase *Class<T>* desempeña un papel central en la programación reflexiva. Antes hemos estudiado diferentes caminos para obtener instancias de esta clase. Ahora veremos los métodos más importantes de la misma. Estos métodos son:

```
...
String getName()
Class<?> getSuperclass()
Class<?>[ ] getInterfaces()

Field[ ] getFields()
Method[ ] getMethods()
Constructor<T>[ ] getConstructors()

Field[ ] getDeclaredFields()
Method[ ] getDeclaredMethods()
Constructor<T> [ ] getDeclaredConstructors()

Constructor<T> getConstructor(Class[ ] parametrosFormales)
Method getMethod(String nombreMétodo, Class[ ] parametrosFormales)

Constructor<T> getDeclaredConstructor(Class[ ] parametrosFormales)
Method getDeclaredMethod(String nombreMétodo, Class[ ] parametrosFormales)

Object newInstance()
...
```

- *String getName():* Nombre cualificado del tipo
- *boolean isInterface():* Si es interface o no
- *Class<?> getSuperclass():* Tipo de la superclase si la hay.
- *Class<?>[ ] getInterfaces():* Interfaces implementados o heredados
- *Field[ ] getDeclaredFields():* Atributos declarados del tipo (públicos, privados, …)
- *Field[ ] getFields():* Atributos públicos del tipo ya sean declarados o heredados.
- *Method[ ] getDeclaredMethods():* Métodos declarados del tipo ya sean públicos o privados
- *Method[ ] getMethods():* Métodos públicos del tipo ya sean declarados o heredados
- *Method getDeclaredMethod(String nombreMétodo, Class[ ] parametrosFormales)* Método declarado de la clase con el nombre y los parámetros formales especificados.
- *Method getMethod(String nombreMétodo, Class[ ] parametrosFormales):* Método de la clase con el nombre y los parámetros formales especificados.
- *Constructor<T> [ ] getDeclaredConstructors():* Todos los constructores declarados por clase (públicos, privados, protected, …)
- *Constructor<T>[ ] getConstructors():* Constructores públicos de la clase

- *Constructor<T> getConstructor(Class[ ] parametrosFormales):* Constructor público de la clase con los parámetros formales especificados
- *Constructor<T> getDeclaredConstructor(Class[ ] parametrosFormales)):* Constructor declarado de la clase (público o privado o …) con los parámetros formales especificados
- *Object newInstance():* Una instancia del tipo que representa el objeto. La instancia devuelta es la que construye el constructor vacío.

Como vemos aparecen tres tipos nuevos: *Constructor<T>, Field y Method*

1.3 Clases Constructor, Field y Method

La clase *Constructor* se utiliza para representar a un constructor de una clase. La clase *Field* se utiliza para representar a un atributo de una clase. La clase *Method* se utiliza para representar a un método de un tipo. Por ejemplo tenemos:

```
Class<?> tri = Class.forName("algebra.RacionalImpl");
Field[ ] atr = tri.getDeclaredFields();
Method[ ] met = tri.getDeclaredMethods();
Constructor[ ] cons = tri.getConstructors();
```

Veamos los diferentes métodos de las tres clases.

Métodos comunes a *Field, Method y Constructor:*

> *String getName():* Nombre
> *int getModifiers():* Devuelve un entero que codifica los diferentes modificadores posibles y sus combinaciones. Para preguntar por un modificador concreto se pueden usar los métodos estáticos de la clase *Modifier*.

Métodos comunes a *Method y Constructor:*

> *Class<?>[] getParameterTypes():* Tipos de los Parámetros formales

Métodos específicos de *Field:*

> *Object get(Object obj):* Valor del atributo en el objeto *obj*
> *void set(Object obj, Object val):* Cambio del valor del atributo a val en el objeto *obj*

Método especifico de *Method:*

> *Object invoke(Object obj, Object[] pr):* Invoca el método sobre el objeto *obj* con parámetros reales *pr* y devuelve el resultado del método.

Método específico de Constructor:

Object newInstance(Object[] pr): Devuelve una instancia construida cuando al constructor se le pasan los parámetros reales *pr*.

En el trozo de código siguiente obtenemos, a partir de una clase, los interfaces que implementa, los atributos que tiene y los métodos.

```
Class<?> repC= prac8.EmpleadoImpl.class;

Class<?>[] interfaces = repC.getInterfaces();
for(Class<?> i: interfaces)
       mostrar(i.getName());

Field[] atributos = repC.getFields();
for(Field f: atributos)
       mostrar(f.getName());

Method[] metodos = repC.getMethods();
for(Method m: metodos)
       mostrar(m.getName());

interfaces = interfaces[0].getInterfaces();
for(Class<?> i: interfaces)
       mostrar(i.getName());

atributos = repC.getDeclaredFields();
for(Field f: atributos)
       mostrar(f.getName());
```

2. Usos de la Programación Reflexiva

Los ejemplos que vamos a ir presentado los implementaremos como métodos estáticos de una clase que llamaremos *Reflexion* que puede ser reutilizada.

2.1 Obtener los métodos de un tipo (clase o interfaz)

Dado el identificador de un tipo (clase o interface) obtener las signaturas de los métodos declarados en el tipo (públicos, privados o protected).

```
public static Set<String> getMetodosDeclarados(String nT) {
              Set<String> bm = Sets.newHashSet();
              try {
                     Method[] am = Class.forName(nT).getDeclaredMethods();
                     for(Method m: am) {
                            bm.add(m.toString());
                     }
              } catch (Exception e) { e.printStackTrace(); }
              return bm;
}
```

De la misma forma obtener todos los métodos públicos tipo ya sean declarados o heredados.

```
public static Set<String> getTodosLosMetodos(String nT) {
		Set<String> bm = Sets.newHashSet();
		try {
			Method[] am = Class.forName(nT).getMethods();
			for(Method m: am)
				bm.add(m.toString());
		} catch (Exception e) {	e.printStackTrace(); }
		return bm;
}
```

Obtener todos los métodos públicos de un tipo que tengan un nombre dado:

```
public static Set<String> getTodosLosMetodosConUnNombre(String nT,
    String nombre) {
		Set<String> bm = Sets.newHashSet();
		try {
			Method[] am = Class.forName(nT).getMethods();
			for(Method m: am){
				if(m.getName().equals(nombre)){
					bm.add(m.toString());
				}
			}
		} catch (Exception e) {	e.printStackTrace(); }
		return bm;
}
```

Igualmente podríamos obtener todos los métodos cuyo nombre cumple algunos requisitos (tiene un prefijo dado, …).

Obtener todos los métodos privados declarados por el tipo:

```
public static Set<String> getMetodosPrivadosDeclarados(String nT) {
		Set<String> bm = Sets.newHashSet();
		try {
			Method[] am = Class.forName(nT).getDeclaredMethods();
			for(Method m: am){
				if(Modifier.isPrivate(m.getModifiers())){
					bm.add(m.toString());
				}
			}
		} catch (Exception e) {	e.printStackTrace(); }
		return bm;
}
```

```
...
mostrar("Los métodos de PuntoMetrico son:",
	Reflexion.signaturasDeMetodosDeclarados("ejemplos.PuntoMetrico"));
...
```

2.2 Obtener los atributos de un tipo

Obtener un conjunto con la visibilidad, el tipo y el nombre de los diferentes atributos declarados.

```
public static Set<String> getAtributosDeclarados(String nC) {
		Set<String> la = Sets.newHashSet();
		try {
			Field[] aa = Class.forName(nC).getDeclaredFields();
			for(Field a: aa)
				la.add(a.toString());
		} catch(Exception e){ e.printStackTrace(); }
		return la;
}
```

Obtener un conjunto con el tipo y el nombre de los diferentes atributos públicos.

```
public static Set<String> getAtributosPublicos(String nC) {

		Set<String> la = Sets.newHashSet();
		try {
			Field[] aa = Class.forName(nC).getFields();
			for(Field a: aa)
				la.add(a.toString());
		} catch(Exception e){ e.printStackTrace(); }
		return la;
}
```

```
...
mostrar("Los atributos declarados de PuntoMetricoImpl son:",
     Reflexion.getAtributosDeclarados("ejemplos.PuntoMetricoImpl"));
...
```

2.3 Invocar un método sobre un objeto

Preguntar si existe un método con un nombre y unos parámetros formales dados

```
public static boolean existeMetodo(String nT, String nM, Class<?>[] pForms) {
		boolean r = true;
		try {
			Class.forName(nT).getMethod(nM, pForms);
		} catch (NoSuchMethodException e) {
			r = false;
		}
		catch (Exception e) {
			e.printStackTrace();
		}
		return r;
}
```

Ejecutar, si es posible, el método sin parámetros de nombre *nM* sobre el objeto *obj* y devolviendo como *Objec*t el resultado del mismo.

```
public static Object ejecutaMétodo(Object obj, String nM) throws
     IllegalArgumentException, IllegalAccessException,
     InvocationTargetException {
		Object x = null;
		Class<?>[] pf = new Class[0];

		try {
```

```
                    Method m = obj.getClass().getMethod(nM,pf);
                    x = m.invoke(obj,(Object[])null);
             } catch (SecurityException e) {
                    e.printStackTrace();
             } catch (NoSuchMethodException e) {
                    e.printStackTrace();
             }
             return x;
}
```

Ejecutar, si es posible, un método de nombre *nM* y parámetros formal especificados sobre el objeto *obj* y con unos parámetros reales dados.

```
public static Object ejecutarMétodo(
   Object obj, String nM, Class<?>[] pf, Object[] pr)
   throws IllegalArgumentException, IllegalAccessException,
   InvocationTargetException, SecurityException, NoSuchMethodException {

             Method m = obj.getClass().getMethod(nM, pf);
             return m.invoke(obj, pr);
}
```

Ejecutar, si es posible, un método de nombre *nM* sobre el objeto *obj* y con unos parámetros reales dados.

```
public static Object ejecutarMétodo(Object obj, String nM, Object[] pR) {

       Object x=null;
       try {
              Class<?>[] pF = new Class<?>[pR.length];
              for(int i=0; i < pR.length; i++){
                    pF[i]=pR.getClass();
              }
              x = ejecutarMétodo(obj,nM,pF,pR);
       } catch(Throwable e) {e.printStackTrace();}
       return x;
}
```

En el código anterior hemos de tener en cuenta que los tipos primitivos y son envolturas son diferentes aunque se puedan convertir automáticamente unos en otros. Esto implica que para que el método anterior funcione los tipos de los parámetros reales deben coincidir estrictamente con los parámetros formales de un método del tipo. Una implementación más sofisticada buscaría un método con una signatura compatible a los parámetros reales dados.

2.4 Crear objetos

Crear un objeto de un tipo mediante el constructor vacío

```
public static Object crearObjeto(String nC) throws ClassNotFoundException,
              InstantiationException, IllegalAccessException {
       return Class.forName(nC).newInstance();
}
```

Crear un objeto de un tipo mediante un constructor con unos parámetros formales dados y los correspondientes parámetros reales

```
public static Object crearObjeto(String nC,Class<?>[] pForms,Object[] pReales)
    throws Throwable {
       Class<?> ca = Class.forName(nC);
       Constructor<?> co = ca.getConstructor(pForms);
       return co.newInstance(pReales);
}
```

Crear un objeto de un tipo dados unos parámetros reales y usando un constructor con parámetros formales correspondientes a los tipos de los parámetros reales.

```
public static Object crearObjeto(String nC, Object[] pReales) {
       Object x=null;
       try {
              Class<?>[] pForms = new Class<?>[pReales.length];
              for(int i=0; i<pReales.length; i++){
                     pForms[i]=pReales.getClass();
       }
              x = crearObjeto(nC,pForms,pReales);
       } catch(Throwable e) {e.printStackTrace();}
       return x;
}
```

Como en el caso de la invocación de un método hemos de tener en cuenta que los tipos primitivos y son envolturas son diferentes aunque se puedan convertir automáticamente unos en otros. Esto implica que para que código anterior funcione los tipos de los parámetros reales deben coincidir estrictamente con los parámetros formales de un constructor. Una implementación más sofisticada buscaría un método con una signatura compatible a los parámetros reales dados.

2.5 Mostrar las propiedades de un tipo

```
public static void mostrarPropiedades(String nT) {
  try {
       Class<?> cc = Class.forName(nT);
       Test.mostrar("TIPO: ",cc.getSimpleName());
       if(cc.getSuperclass()!=null)
       Test.mostrar("\tSuperClase = ",cc.getSuperclass().getSimpleName());
       Test.mostrar("\tN° de constructores = ",cc.getConstructors().length);
       Test.mostrar("\tConjunto de Interfaces = ",interfaces(nT));
       Test.mostrar("\tConjunto de Atributos = ",getAtributosDeclarados(nT));
       Test.mostrar("\tConjunto de Métodos = ",getMetodosDeclarados(nT));
       Test.mostrar("\tClases internas = ",clasesInternas(nT));
  } catch(Exception e) {e.printStackTrace();}
}
```