Tema 1 (Parte 5)
El API Reflection y anotaciones

J. Gutiérrez

Departament d'Informàtica
Universitat de València

DAW-TS (ISAW).
Curso 14-15

SOME RIGHTS RESERVED creative commons

Índice

SOME RIGHTS RESERVED creative commons

Índice

SOME RIGHTS RESERVED creative commons

Objetivos I

1. Identificar clases que pertenecen al paquete `java.lang.reflect`.
2. Utilizar *reflection* para crear instancias de un determinado tipo y llamar a métodos de las instancias construidas.
3. Interpretar qué hace un determinado código fuente que utiliza *reflection*.

SOME RIGHTS RESERVED creative commons

Índice

API Reflection

Mediante los tipos y métodos que ofrece esta API es posible obtener un objeto (en memoria) que representa a un fichero `.class`.

Una vez se tiene ese objeto es posible:

- Obtener un objeto asociado a un constructor de la clase que permite crear una instancia.
- Obtener un objeto asociado a un método para enviar un mensaje a una instancia.
- Buscar un determinado atributo y modificar su valor (esto puede ser peligroso ya que se puede violar la encapsulación).
- ...

Además sería posible realizar una aplicación que acepte un fichero .class como entrada, lo analice usando *reflection* y genere código fuente de forma dinámica .

API Reflection

El API *reflection* proporciona una serie de clases entre las que cabe destacar (el listado completo así como la descripción y sus operaciones se debe consultar en la documentación):

Tipo	Descripción de la instancia
`Class<T>`	Representa un fichero .class en memoria. Usando los métodos de esta clase se puede acceder a los atributos, métodos y constructores..
`Method`	Representa a un método de una clase
`Constructor<T>`	Representa a un constructor de una clase
`Field`	Representa a un atributo de una clase

¿Cómo se usan estas clases?

1. Obtener una instancia del tipo `Class<T>` que representa a una clase compilada `T.class`
2. Usar esa instancia para obtener instancias de objetos que representan a los atributos (`Field`), métodos (`Method`), constructores (`Constructor`), etc
3. Usar una instancia del tipo `Constructor` que permite crear una instancia del tipo `T` .
4. Usar una instancia del tipo `Method` para enviar un mensaje al objeto del tipo `T` .
5. Buscar la presencia de anotaciones en atributos o métodos. Las anotaciones se usan para simplificar el desarrollo de aplicaciones para la plataforma Java EE.

Tres formas de obtener un objeto del tipo `Class<T>`

- A partir de una cadena con el nombre de la clase:

```
Class<String> clase = Class.forName(``java.lang.String'');

Class<?> clase = Class.forName(args[0]);
```

- Usando .class:

```
Class<String> clase = java.lang.String.class;
```

- Usando el método `getClass()` declarado en la clase `Object`:

```
// Devuelve un objeto Class con la clase a la que pertenece el código
Class<?> clase = getClass();
```


Algunos métodos de Class<T> relacionados con clases

```
// Devuelve la superclase de la clase a la que se le envía el mensaje
public Class<?> getSuperClass()
```

```
// Devuelve las interfaces a las que implementa
public Class<?>[] getInterfaces()
```

```
// Devuelve las clases, interfaces que son miembros de la clase
// sin incluir los heredados (es decir, sólo los que se han declarado en ella).
public Class<?>[] getDeclaredClasses()
```

```
// Si la clase es miembro (ha sido declarada en otra clase) devuelve la
// clase donde ha sido declarada. Si no es miembro de otra clase devuelve null
public Class<?> getDeclaringClass()
```

```
// Devuelve la clase en la que está contenida (por ejemplo, si creamos una
// instancia de una clase anónima en el código de una clase A, entonces
// una llamada a getEnclosingClass() sobre la clase anónima devuelve A.
public Class<?> getEnclosingClass()
```


Ejemplo I

```
import java.io.Serializable;
import java.rmi.Remote;

public class ClassDemo extends Thread implements Serializable, Remote{
   public static class Interna{
      public void m1(){
         System.out.print("La clase Interna está declarada dentro de la clase: ");
         System.out.println(getClass().getEnclosingClass().getSimpleName());
      }
   }

   public static void main(String[] args){
      Class<ClassDemo> c = ClassDemo.class;

      Interna it = new Interna();
      it.m1();

      System.out.print("La clase ClassDemo extiende a: ");
      System.out.println(c.getSuperclass().getSimpleName());

      // Obtención de las clases declaradas dentro de esta clase
      Class<?>[] clasesInternas = c.getDeclaredClasses();

      for (int i=0;i<clasesInternas.length;i++){
         System.out.print("La clase ClassDemo declara la clase: ");
         System.out.println(clasesInternas[i].getSimpleName());
```


Ejemplo II

```
      }

      // Obtención de las interfaces a las que implementa
      Class<?>[] interfs = c.getInterfaces();

      for (int i=0;i<interfs.length;i++){
         System.out.print("La clase ClassDemo implementa a la interfaz: ");
         System.out.println(interfs[i].getSimpleName());
      }

      new Thread(new Runnable(){
         public void run(){
            System.out.print("La clase anónima implementa a la interfaz: ");
            System.out.println(getClass().getInterfaces()[0].getSimpleName());
            System.out.print("La clase anónima está declarada en: ");
            System.out.println(getClass().getEnclosingClass().getSimpleName());
         }
      }).start();
   }
}
```

Métodos de Class<T> relacionados con constructores

```
Constructor<T> getConstructor(Class<?>... parameterTypes)
//Devuelve un objeto Constructor correspondiente a un constructor público de la
//clase. Puesto que la clase puede declarar diferentes constructores,
//especificamos el que deseamos indicando los tipos de los parámetros formales
//declarados en el constructor
```

```
Constructor<?>[]  getConstructors()
//Devuelve un vector de objetos Constructor con los constructores públicos de la clase
```

```
Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)
//Devuelve un objeto Constructor correspondiente a un constructor de la clase
//ya sea público, privado, protected o con acceso de paquete.
```

```
Constructor<?>[]  getDeclaredConstructors()
//Devuelve un vector de objetos Constructor correspondientes a constructores de
//la clase ya sean públicos, privados, protected o con acceso de paquete.
```


Tipo... ref como argumento en un método

En el argumento de algunos de estos métodos aparecen puntos suspensivos tras el tipo.

Esto indica que pueden recibir un número variable de argumentos de este tipo o también un array de ese tipo. Ejemplo:

```
class Ejemplo{
   public static void metodo(String...p){
      int nDatos = p.length;
      for (int i=0;i<nDatos; i++)
         System.out.print(p[i] + " ");
      System.out.println();
   }
}
class UsaEjemplo{
   public static void main(String[] args){
      // Todas estas sentencias son validas:
      Ejemplo.metodo("Cadena 1");
      Ejemplo.metodo("Cadena 1","Cadena 2");
      Ejemplo.metodo("Cadena 1","Cadena 2","Cadena 3");
      Ejemplo.metodo(new String[]{"Cadena 1","Cadena 2","Cadena 3","Cadena 4"});
   }
}
```


Algunos métodos de Constructor

```
String getName()
//Devuelve el nombre del constructor
```

```
Class<?>[] getParameterTypes()
// Devuelve un vector con los tipos de los argumentos
```

```
Class<?>[] getExceptionTypes()
// Devuelve un vector con los tipos de las excepciones que puede lanzar
```

```
boolean isVarArgs()
// Indica si puede recibir un número variable de argumentos
```

```
T newInstance(Object... initargs)
// Crea una instancia del objeto
```


Ejemplo I

```
import java.lang.reflect.*;

class EjemploConstructor{
  EjemploConstructor(){
  }
  EjemploConstructor(double d){
  }
  EjemploConstructor(double[] d){
  }
  public EjemploConstructor(String s){
  }
  public EjemploConstructor(String s, double d){
  }

  private static void showConstructorInfo(Constructor<?> con){
    System.out.println("-------------------------------------");
    System.out.println("Constructor: " + con.getName());
    System.out.println("El constructor es: " + Modifier.toString(con.getModifiers()));
    Class<?>[] formalParameters = con.getParameterTypes();
    for (Class<?> cl: formalParameters)
    System.out.println("   Parametro formal: " + cl.getSimpleName());
  }

  public static void main(String[] args){
    Class<EjemploConstructor> c = EjemploConstructor.class;
```

Ejemplo II

```
    System.out.println("All constructors");
    Constructor<?>[] cs = c.getDeclaredConstructors();

    for (Constructor<?> con: cs)
    showConstructorInfo(con);

    System.out.println("\n\nPublic Constructors");
    cs = c.getConstructors();

    for (Constructor<?> con: cs)
    showConstructorInfo(con);

    // Obtención de constructores particulares
    try{
      Constructor<EjemploConstructor> c1 = c.getDeclaredConstructor(new Class<?>[]{});
      Constructor<EjemploConstructor> c2 = c.getDeclaredConstructor(double.class);
      Constructor<EjemploConstructor> c3 = c.getDeclaredConstructor(double[].class);

      Class<?>[] formal = new Class<?>[2];
      formal[0] = String.class;
      formal[1] = double.class;
      Constructor<EjemploConstructor> c4 = c.getConstructor(formal);

      Object[] arg = new Object[2];
      arg[0] = "Cadena";
      arg[1] = 3.5;
```


Ejemplo III

```
      EjemploConstructor ej = c4.newInstance(arg);

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


Métodos de Class<T> relacionados con atributos

```
Field getField(String nombre)
// Obtiene un objeto Field que representa a un atributo público
```

```
Field getDeclaredField(String nombre)
// Obtiene un objeto Field que representa a un atributo
```

```
Field[] getFields()
// Obtiene un vector de objetos Field con los atributos públicos
```

```
Field[] getDeclaredFields()
// Obtiene un vector de objetos Field con los atributos
```


Métodos de Field

```
String  getName()
// Devuelve el nombre del atributo
```

```
Class<?> getType()
// Devuelve el tipo del atributo
```

```
Object get(Object instancia)
// Devuelve el valor del objeto. El argumento es la instancia a la que pertenece.
```

```
void set(Object instancia, Object valor)
// Asigna un nuevo valor al atributo
```

Ejemplo I

```
import java.lang.reflect.Field;

public class AtributosDemo {
   private int a1;
   protected String a2;
   public double a3;
   long a4;

   AtributosDemo(){
      a2 = "Cadena inicial";
   }

   public String toString(){
      return a1 + ", " + a2 + ", " + a3 + ", " + a4;
   }

   public static void main(String[] args) throws Exception{

      AtributosDemo ad = new AtributosDemo();

      Class<AtributosDemo> c = AtributosDemo.class;

      Field[] atrib = c.getDeclaredFields();

      System.out.println("Valores de los atributos: ");
      for (int i=0;i<atrib.length; i++){
```


Ejemplo II

```
         if (!atrib[i].isAccessible())
            atrib[i].setAccessible(true);
         System.out.print(atrib[i].getName() + ": ");
         System.out.println(atrib[i].get(ad).toString());
      }

      atrib[2].set(ad, "Valor cambiado");

      System.out.println(ad.toString());
   }
}
```


Métodos de Class<T> relacionados con métodos

```
Method getMethod(String name, Class<?>... parameterTypes)
// Devuelve un objeto Method que representa un método público
```

```
Method getDeclaredMethod(String name, Class<?>... parameterTypes)
// Devuelve un objeto Method que representa un método
```

```
Method[] getMethods()
// Devuelve un vector de objetos Method con los métodos públicos
```

```
Method[] getDeclaredMethods()
// Devuelve un vector de objetos Method con los métodos
```


Algunos métodos de la clase `Method`

```
Class<?>[] getExceptionTypes()
// Devuelve un vector con los tipos de las excepciones que puede lanzar
```

```
Type getGenericReturnType()
// Devuelve el tipo de lo que devuelve el método
```

```
int getModifiers()
// Devuelve el modificador de acceso (public, protected, etc)
// Para decodificar el entero se puede usar la clase Modifier
```

```
public Object invoke(Object obj, Object... args)
// Permite enviar un mensaje al objeto que se pasa como primer argumento. Los
// argumentos al método se pasan en el segundo argumento de invoke.
```

Ejemplo

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

class Super{
   public void m1(int v){}
}
public class MetodosDemo {
   public void m1(){}
   protected int m2() throws Exception{
      return 0;
   }
   private void m3(String cad){}

   public static void main(String[] args){
      MetodosDemo md = new MetodosDemo();

      Class<MetodosDemo> mdc = MetodosDemo.class;

      Method[] met = mdc.getMethods();
      for (int i=0;i<met.length;i++){
         System.out.print(met[i].getName());
         System.out.println(":    " + met[i].getGenericReturnType().toString());
      }

      System.out.println("_____________________");

      Method[] metd = mdc.getDeclaredMethods();
      for (int i=0;i<metd.length;i++){
         System.out.print(metd[i].getName());
         System.out.println(":    " + metd[i].getGenericReturnType().toString());
      }
   }

}
```

SOME RIGHTS RESERVED creative commons

Índice

SOME RIGHTS RESERVED creative commons

Ejemplo 1

Supongamos que tenemos una clase cuyo código fuente es:

```
class Real{
   private double valor;

   Real(double  d){
      valor = d;
   }

   public void setValor(double d){
      valor = d;
   }

   public double getValor(){
      return valor;
   }
}
```

SOME RIGHTS RESERVED creative commons

Ejemplo 1 I

```
public class ReflectionA {
   public static void main(String[] args) throws Exception{
      //Obtencion de una instancia del tipo Class<Real>
      Class<Real> realClass = Real.class;

      // Esta es otra posible forma de obtener una instancia
      //Class<Real> realClass = Class.forName("Real");

      //Obtencion de un objeto que representa el constructor
      Constructor<Real> realConstructor = realClass.getConstructor(double.class);

      //Creacion de una instancia
      Real r = realConstructor.newInstance(new Double(3.99));

      //Obtencion de una instancia que representa al metodo getValor
      Class<?>[] voidArg = {};
      Method realGetValor = realClass.getMethod("getValor", voidArg);

      //Envio de un mensaje al Real usando la instancia del tipo Method
      Object[] noValue = {};
      double d = (Double)(realGetValor.invoke(r, noValue));

      System.out.println(d);

      //Obtencion de una instancia que representa al metodo setValor
      Method realSetValor = realClass.getMethod("setValor", double.class);
```

SOME RIGHTS RESERVED creative commons

Ejemplo 1 II

```
        //Envio de un mensaje al Real usando la instancia del tipo Method
        realSetValor.invoke(r, new Double(2.0));

        //Comprobacion de que hemos cambiado el valor
        System.out.println(r.getValor());
    }
}
```


Explicación del ejemplo 1

En el código anterior se realizan las siguientes tareas:

1. Obtención de una instancia del tipo `Class<Real>`
2. Obtención de una instancia del tipo `Constructor<Real>`
3. Creación de un `Real` usando una instancia del tipo `Constructor`. La clase `Constructor` tiene el método `newInstance(.)` que se usa para obtener una nueva instancia.
4. Obtención de una instancia del tipo `Method` del método `getValor()`
5. Envío de un mensaje al `Real` usando la instancia anterior. La clase `Method` tiene el método `invoke(.)` que se usa para enviar un mensaje al objeto.
6. Obtención de una instancia del tipo `Method` correspondiente al método `setValor(double d)`
7. Envío de un mensaje al `Real` usando la instancia anterior para cambiar el valor que almacena.

Código equivalente sin usar reflection

```
class SinReflection{
   public static void main(String[] args){
      Real r = new Real(3.99);
      double d = r.getValor();
      System.out.println(d);
      r.setValor(2.0);
      System.out.println(r.getValor());
   }
}
```

¡Vaya!, no parece que hayamos ganado mucho usando *reflection*.

Obviamente este no es un ejemplo que muestre en qué situación es apropiado el uso de *reflection*, sino que ilustra cómo se usa.

Ejemplo 2

Este ejemplo es más complejo que el anterior: en este caso no existe la clase `Real` cuando se compila el código de la clase que contiene el `main(.)`.

La clase `Real` se crea y compila al ejecutar el `main(.)`.

En el `main(.)` se realizan las siguientes tareas:

1. Escritura del código fuente de la clase `Real` en un fichero
2. Compilación del código fuente (para ello se usa la clase `Main` y es necesario incorporar al `classpath` una biblioteca que no está en el JRE pero si en el JDK: `tools.jar`)
3. El resto del código realiza las mismas tareas que el ejemplo anterior salvo que no puede aparecer ninguna referencia del tipo `Real` ya que cuando se compila este código esa clase no existe.
4. Al final se borran el fichero fuente y el compilado de `Real`.

Ejemplo 2 (cont.) I

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import java.io.*;
import com.sun.tools.javac.Main;

public class ReflectionB {

   //Este metodo genera el fichero Real.java
   public static void generaCodigoFuente(){
      try{
         PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter("Real.java"))
               ,true);
         pw.println("public class Real{");
         pw.println("   private double valor;");
         pw.println("   public Real(double  d){");
         pw.println("      valor = d;");
         pw.println("   }");
         pw.println("   public void setValor(double d){");
         pw.println("      valor = d;");
         pw.println("   }");
         pw.println("   public double getValor(){");
         pw.println("      return valor;");
         pw.println("   }");
         pw.println("}");
         pw.close();
      }catch(IOException ex){
         ex.printStackTrace();
```


Ejemplo 2 (cont.) II

```
      }
   }

   //Este metodo compila el fichero Real.java
   public static void compilaCodigoFuente() throws Exception{
      String[] fichero = {"Real.java"};
      Main.compile(fichero);
   }

   //Este metodo borra el fichero Real.java
   public static void borraReal() throws Exception{
      File f = new File("Real.java");
      f.delete();

      f = new File("Real.class");
      f.delete();
   }

   public static void main(String[] args) throws Exception{

      generaCodigoFuente();

      compilaCodigoFuente();

      Class<?> realClass = Class.forName("Real");

      // Obtencion de un objeto que representa el constructor
      Constructor<?> realConstructor = realClass.getConstructor(double.class);
```


Ejemplo 2 (cont.) III

```
      // Creacion de una instancia
      Object r = realConstructor.newInstance(new Double(3.99));

      // Obtencion de una instancia que representa al metodo getValor
      Class<?>[] voidArg = {};
      Method realGetValor = realClass.getMethod("getValor", voidArg);

      // Envio de un mensaje al Real usando la instancia del tipo Method
      Object[] noValue = {};
      double d = (Double)(realGetValor.invoke(r, noValue));

      System.out.println(d);

      // Obtencion de una instancia que representa al metodo setValor
      Method realSetValor = realClass.getMethod("setValor", double.class);

      // Envio de un mensaje al Real usando la instancia del tipo Method
      realSetValor.invoke(r, new Double(2.0));

      d = (Double)(realGetValor.invoke(r, noValue));

      System.out.println(d);

      borraReal();
   }
}
```


En este caso no podemos sustituir *reflection* por el código simple dado en la diapositiva 33.

Además, en el `main(.)` no puede haber ninguna referencia declarada del tipo `Real`.

La explicación a los dos comentarios anteriores es que en el momento en que se compila no existe el fichero `Real.class` y el compilador indicaría que no se puede usar ese tipo.

Ejemplo 3

En este ejemplo se muestra información sobre todos los métodos de una clase.

```
import java.lang.reflect.*;

public class ReflectionC {
   public static void main(String[] args) throws Exception {
      // Obtencion de una instancia del tipo Class<Real>
      Class<?> clase = Class.forName(args[0]);

      Method[] ms = clase.getMethods();

      for (int i = 0; i < ms.length; i++) {
         System.out.print("Informacion sobre el metodo " + i + " :");
         System.out.println(ms[i].getName());
         System.out.println("   Declarado en : " + ms[i].getDeclaringClass().
              getSimpleName());
         System.out.println("   Modificador: "
               + Modifier.toString(ms[i].getModifiers()));

         Class<?>[] ar = ms[i].getParameterTypes();
         for (int j = 0; j < ar.length; j++) {
            System.out.print("   Tipo argumento " + j + ": ");
            System.out.println(ar[j].getSimpleName());
         }
         System.out.println("   Return: " + ms[i].getReturnType().getName());
         System.out.println("-------------------------------");
      }
   }
}
```


Ejemplos de ejecución:

```
java ReflectionC java.awt.Button
java ReflectionC java.lang.String
java ReflectionC java.lang.System
```


Resumen

En definitiva, hemos visto que *reflection* permite

- analizar un fichero `.class`
- crear instancias
- enviar mensajes

Además, se podría analizar un fichero `.class` y generar código automáticamente para liberar al programador de escribir ese código.

Esta funcionalidad se usa en RMI, en EJBs y en Servicios Web para generar código de forma automática.

Índice

Las anotaciones

Una parte esencial de las nuevas versiones de la plataforma Java EE es la definición y uso de anotaciones para facilitar el desarrollo de aplicaciones empresariales.

Las anotaciones se utilizan en los siguientes contextos:

- Para indicar que una clase es un determinado tipo.
- Para indicar que se debe realizar la inyección de una instancia a un atributo.
- Para indicar que un método realiza una tarea particular.

- Se trata de un modo de programación declarativo: se indica qué hacer con un elemento en lugar de dar los pasos (el procedimiento) para hacerlo.
- Las anotaciones simplifican la programación: se puede conseguir lo mismo sin usarlas pero hay que escribir más código.
- Las anotaciones son detectadas mediante *reflection* para realizar la tarea que indican.
- En el caso de Java EE las anotaciones las procesa el contenedor que gestiona el ciclo de vida del componente.

API Reflection y anotaciones

Las clases `Class`, `Constructor`, `Method` y `Field` declaran los métodos:

```
// Devuelve un array de referencias del tipo \texttt{Annotation} con
// las anotaciones presentes en ese elemento
Annotation[] getDeclaredAnnotations()

// Devuelve la anotación del tipo solicitado o null si no está presente
T getAnnotation(Class<T> annotationClass)
```

La interface `AnnotatedElement` define el método

```
// Devuelve true si el elemento está anotado con la anotación proporcionada
boolean  isAnnotationPresent(Class<? extends Annotation> annotationClass)
```


A continuación se va a mostrar un ejemplo completo en el que:

- Se creará una anotación
- Se usará la anotación
- Se procesará el código que usa la anotación y se hará una tarea.

Con este ejemplo no afirmo que el contenedor Java EE trabaje así. Se trata de un ejemplo que muestra todo el proceso.

En primer lugar vamos a crear una nueva anotación:

```
import java.lang.annotation.*;

@Documented //La anotacion aparecera en la documentacion del elemento anotado
@Retention(RetentionPolicy.RUNTIME) //La anotacion estara en tiempo de ejecucion
@Target({ElementType.FIELD}) //Anota a atributos
public @interface Componente {
  String tipo() default "javax.swing.JTextField";
  String texto() default "";
}
```


Ahora vamos a crear código que usa la anotación anterior:

```
import java.awt.*;
import javax.swing.*;
public class Aplicacion extends JFrame{
   private static final long serialVersionUID = 1L;

   // Notese que en este código no se asigna una
   // instancia a c1 ni a c2, quien gestione el
   // objeto Aplicacion debe proporcionar (inyectar) una instancia
   @Componente(tipo="javax.swing.JButton",texto="Enviar")
   private Component c1;

   @Componente(tipo="javax.swing.JTextArea",texto="Texto de prueba en el area de texto")
   private Component c2;

   public void inicializa(){
      JPanel p = new JPanel();
      p.setLayout(new BorderLayout());
      p.add(c1,BorderLayout.SOUTH);
      p.add(c2,BorderLayout.CENTER);
      this.add(p);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(400,400);
      setVisible(true);
   }
}
```

Si ejecutamos esta aplicación obtendremos una excepción del tipo `NullPointerException` .

Finalmente necesitamos una aplicación que lo gestione creando una instancia del tipo `Aplicacion` e inyectando instancias a los atributos anotados:

```
import java.lang.reflect.*;

public class EjemploInjection {
   public static void main(String[] args) throws Exception{

      Class<Aplicacion> c = Aplicacion.class;

      Field[] f = c.getDeclaredFields();

      Constructor<Aplicacion> constr = c.getConstructor(new Class[] {});

      // Creacion de una instancia a partir del constructor
      Object[] params = new Object[] {};
      Aplicacion instancia = constr.newInstance(params);

       for (int i=0;i<f.length;i++){

         if (((AnnotatedElement)f[i]).isAnnotationPresent(Componente.class)){
            Componente cont = ((AnnotatedElement)f[i]).getAnnotation(Componente.class);

            // Hacemos accesible el atributo (aunque sea privado)
```



```
            f[i].setAccessible(true);


            // Obtenemos la clase correspondiente al tipo
            // que se especifique en la anotación
            Class<?> com = Class.forName(cont.tipo());

            // Obtenemos el constructor que admite una cadena
            Constructor<?> cc = com.getConstructor(new Class[] {String.class});

            // Al atributo anotado se inyectamos la instancia
            // indicada en la anotación
            f[i].set(instancia,cc.newInstance(new Object[] {cont.texto()}));
         }
      }
      instancia.inicializa();
   }

}
```