Reflection en Java

La Reflection en Java (en español Reflexión) es una tecnología maravillosa, muy sencilla de aplicar; pero como todo gran poder, merece de una gran responsabilidad.

Nota para nuevos programadores Java: si estás empezando a programar con Java, no te recomiendo que utilices Reflection hasta que no domines la programación orientada a objetos. Te recomiendo que domines correctamente el resto previamente, tales como Java básico,herencia, polimorfismo, entre otros.

Nota sobre uso de Reflection: Siempre que se pueda es mejor prescindir de Reflection ya que afecta al rendimiento del programa. Es recomendable siempre que facilite el desarrollo que de no aplicarlo sería más complicado o requiera mucho más código. Además hay que tener en cuenta la seguridad, pues hay que asegurar las clases aceptadas, ya que podría ser susceptible de ejecución de código (algo parecido a los agujeros de seguridad de la función eval(), aunque si se emplea bien Reflection no habrá ningún problema de seguridad)

Nota sobre lo que vas a aprender en este artículo: Este artículo condensa decenas de tutoriales sobre Reflection, es un conjunto muy completo de información que he recopilado durante estos años.

¿Para qué sirve Reflection en Java? Te voy a responder con preguntas ¿Eres capaz de sentarte ahí y escribirme un programa que me diga de qué tipo es una variable de una clase: int, String, float, boolean, etc? Puede que se te ocurra alguna forma un poco enrevesada; vamos a otra ¿Podrías indicarme si una variable de una clase es public, privada, final o static? A simple vista parece que no es muy útil, veremos que a veces es muy necesario; otra pregunta ¿Cuántos parámetros le llegan a una función de una clase? O también ¿Me puedes hacer un programa que él solo muestre por pantalla el nombre de toda clase, función y variable que tiene dentro? O más difícil todavía ¿Podrías utilizar desde fuera una función privada de un objeto de una clase, o acceder directamente a una variable privada de otra clase?

Seguro que te me has quedado a cuadros, sobre todo con lo de acceder a funciones privadas cuando no se debería nunca ¿Nunca? Debería de tender a cero, aunque a veces **para realizar test unitarios de una función privada de una clase** no nos quedará otro remedio que utilizar Reflection 

El concepto de **Reflection** simplemente nos quiere decir que **el programa tiene la capacidad de observar y modificar su estructura de manera dinámica**. De ahí que se refleje su mismo código: podemos elegir una clase que hemos escrito previamente en un IDE y después de compilada, mientras el programa se ejecuta, poder modificarla.

Al final dejo un enlace con todo el código. Si nunca has hecho nada con Reflection te recomiendo que -con el IDE (como Eclipse) delante- vayas paso a paso programándolo conmigo y viendo cómo funciona, ejecutándolo.

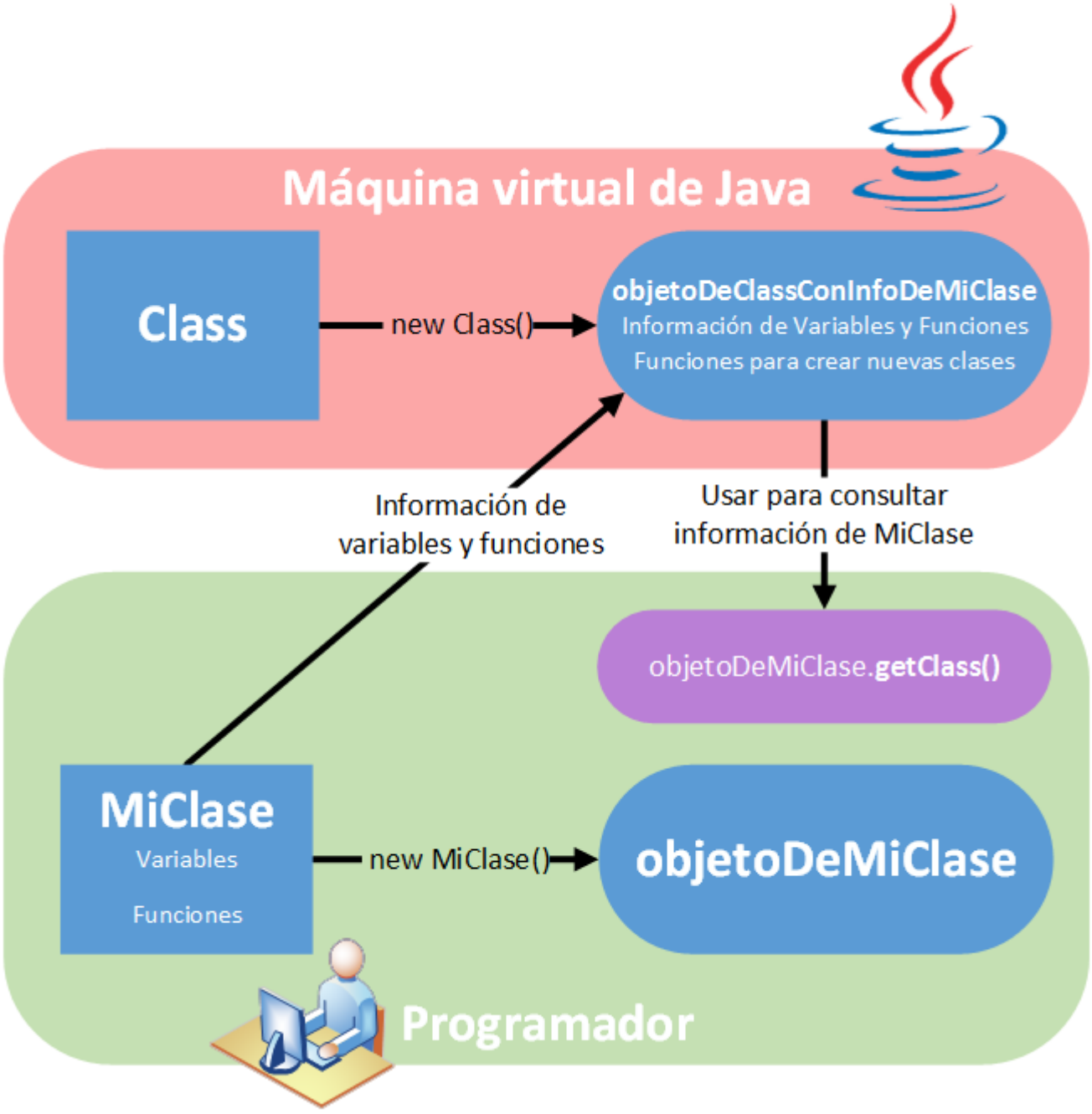
Después de leerte este artículo sobre Reflection es posible que te pique el gusanillo sobre Annotations, pues conociendo como utilizar correctamente Reflection, diseñar nuestras propias anotaciones es muy sencillo, y si no te lo crees te invito a leer este siguiente tutorial sobre las Annotations de Java.

Class

Supongamos que tenemos esta clase:

```
public class MiClase {


        public String unaVariableString = "Un Texto";

        private int unaVariableInt = 5;


        public String getUnaVariableString(String concatenar) {

                return unaVariableString + concatenar;

        }


        private int getUnaVariableInt(int suma) {

                return unaVariableInt + suma;

        }


}
```

La máquina virtual de Java crea una instancia de Class para cada tipo de objeto que incluye funciones para examinar propiedades del objeto e información de tipos, todo ello en tiempo de ejecución. La siguiente imagen muestra el proceso al crear una clase y al instanciarla (Algunas partes de la imagen están explicadas un poco más adelante):



Por ello, para acceder a la información de variables y funciones, tendremos que obtener el Class de lo que queramos (Para obtener el Class es necesario poner el <tipo genérico>; suele bastar con poner “Class<?>”, sobre todo cuando recorremos varios objetos de clases diferentes. Aunque aquí

puede haber un problema de seguridad si no definimos correctamente el tipo genérico; para evitar problemas de seguridad, le podemos decir mejor que tome el objeto que herede de mi clase <? extends MiClase>, así aseguramos que no nos puedan insertar otras cosas al no heredar de MiClase). Para ello lo extraeremos del objeto con la función getClass():

```
MiClase objetoDeMiClase = new MiClase();



Class<? extends MiClase> objetoDeClassConInfoDeMiClase = objetoDeMiClase.getClass();
```

Aunque veremos ejemplos más adelante, veo bien englobar lo más utilizados para obtener el Class aquí:

```
//Obtener el Class directamente de un objeto (Recordatorio: un objeto de tipo String válido es u
n texto entre comillas)

Class classDelTexto = "Un texto".getClass();



Integer numero = 5;

Class classDelNumero = numero.getClass();



//Para tipos primitivos tenemos que usar ".class" o con ".TYPE" (preferiblemente utilizar ".clas
s") (Recordatorio: algunos tipos primitivos son int, boolean, float, etc)

Class classDeBooleanA = boolean.class;

Class classDeBooleanB = boolean.TYPE;



//También podemos obtener una clase desde un nombre completamente cualificado (fully-quailified
name); esto es, el nombre del paquete donde está nuestra clase, seguido del nombre de la clase

Class c = Class.forName("com.paquete.MiClase");
```

Entonces podremos obtener información directamente de la clase, como por ejemplo:

```
//Obtener el nombre de la clase

String nombreDeLaClase = objetoDeClassConInfoDeMiClase.getSimpleName();
```

Esto puede aparentar como un poco confuso: tengo que utilizar un objeto de tipo Class para acceder a la información de mi clase que está por otro lado, es demasiado complicado… Para nada complicado, un truco es pensar que el objeto Class extraído de tu clase es tu propia clase sin instanciar.

Obtener Variables

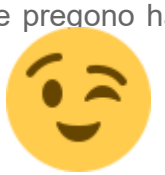
Este anterior truco que te enseño funciona, porque le puedes decir dame todas las variables globales que he declarado a mi clase.

```
//Si queremos obtener una variable pública
Field variableString = objetoDeClassConInfoDeMiClase.getField("unaVariableString");


//Si queremos obtener una variable, sea privada o no
Field variableInt = objetoDeClassConInfoDeMiClase.getDeclaredField("unaVariableInt");


//Si queremos obtener un array con todas las variables públicas de nuestra clase
Field[] todasLasVariables = objetoDeClassConInfoDeMiClase.getFields();


//Si queremos obtener todas las variables, independientemente de si son privadas o no
Field[] todasLasVariablesDeclaradas = objetoDeClassConInfoDeMiClase.getDeclaredFields();
```

Cuando obtenemos solamente una variable de tipo Field por el nombre “hardcoded” (“hardcoded” significa que está escrito directamente en el código fuente un texto cualquiera en un String; por ejemplo hemos escrito "unaVariableString" pasándosela a getField()); por prevención de errores al ser posible que la variable cuyo nombe “hardcoded”, el IDE que utilicemos nos pedirá que las pongamos unos try y catch.

Un ejemplo de cómo quedaría:

```
Field variableString = null;
try {
        variableString = objetoDeClassConInfoDeMiClase.getField("unaVariableString");
} catch (NoSuchFieldException | SecurityException e) {
        e.printStackTrace();
}
```

Nota de buena programación: Para una buena programación el error hay que tratarlo (me da igual que le hagas, notifica al usuario con un mensaje si es necesario, o que se auto-corrija al ejecutar otro código), no me dejes el “e.printStackTrace()” como he hecho yo. Ya sé que pregono haciendo lo contrario, permíteme dejar las cosas claras para que sea fácil y claro de entender 

Para los métodos que devuelven el listado Field[] no es necesario poner try y catch, pues no hay posibilidad a errores.

Obtener funciones

O bien dame todas las funciones que tiene la clase:

```
//Si queremos obtener un método público. Si tiene parámetros podremos pasar cada uno de sus tipos.class  en orden después del nombre

Method metodoGetString = objetoDeClassConInfoDeMiClase.getMethod("getUnaVariableString", String.class);



//Si queremos obtener un método, sea privado o no. Si tiene parámetros podremos pasar cada uno de sus tipos.class en orden después del nombre

Method metodoGetInt = objetoDeClassConInfoDeMiClase.getDeclaredMethod("getUnaVariableInt", int.class);



//Si queremos obtener un array con todas las variables públicas de nuestra clase

Method[] todosLosMetodos = objetoDeClassConInfoDeMiClase.getMethods();



//Si queremos obtener todos los métodos, independientemente de si son privados o no

Method[] todosLosMetodosDeclarados = objetoDeClassConInfoDeMiClase.getDeclaredMethods();
```

Al igual que pasaba con los Field, si lo obtenemos al “hardcoded” los textos, tendremos que poner los try y catch.

Además, estas dos maneras de obtener un método individual, si tiene parámetros podremos pasarlos después del nombre; habrá que poner tantos como tenga (si no tiene ninguno no se pone nada). Por ejemplo, si tuviéramos la función:

```
public void getUnaVariableString(String a, int b, float c) {

        //Contenido…

}
```

Tendríamos que escribir:

```
Method metodo = objeto.getMethod("nomVariable”, String.class, int.class, float.class);
```

Obtener constructores

Y lo mismo para los constructores, aunque en el ejemplo no tengamos:

```
//Se puede utilizar getConstructor() pasándole un array con los parámetros de entrada de tipo Cl
ass que tiene el constructor, para que devuelva el objeto de tipo Constructor, normalmente no se
utiliza y por reducir código no ponemos ejemplos (si quieres saber más tienes más información en
la bibliografía)



//Si queremos obtener un array con todos los constructores publicos de nuestra clase

Constructor[] todosLosConstructores = objetoDeClassConInfoDeMiClase.getConstructors();



//Si queremos obtener todos los métodos, independientemente de si son privados o no

Constructor[] todosLosConstructores = objetoDeClassConInfoDeMiClase.getDeclaredConstructors();
```

Examinar información básica

Veamos un ejemplo. Obtenemos todas las variables de la clase “MiClase” con:

```
Field[] todasLasVariablesDeclaradas = objetoDeClassConInfoDeMiClase.getDeclaredFields();
```

Dentro del listado “todasLasVariablesDeclaradas” estarán contenidas:

```
public String unaVariableString = "Un Texto";

private int unaVariableInt = 5;
```

El anterior listado podemos recorrerlo con:

```
for (Field variable : todasLasVariablesDeclaradas) {

        System.out.println(variable);

}
```

Esto nos muestra por consola la siguiente información de cada variable:

```
public java.lang.String main.MiClase.unaVariableString

private int main.MiClase.unaVariableInt
```

Pero no nos confundamos, la anterior información de la consola solo muestra lo que el objeto de la clase Field nos devuelve por su toString() que tiene por defecto. Nada que ver con lo que necesitamos; y que además, mañana podría cambiar con la siguiente versión, pues el texto que devuelve el toString() no nos asegura que vaya a ser igual.

Nosotros necesitamos cierta información en cómodas variables Java 

Podemos obtener el nombre de una variable (objeto tipo Field) con:

```
//Para obtener el nombre de la variable

String nombreVariable = variable.getName();
```

De este modo, si hacemos el anterior bucle con estos datos:

```
for (Field variable : todasLasVariablesDeclaradas) {

        String nombreVariable = variable.getName();

        System.out.println("\nNombre de la VARIABLE GLOBAL: " + nombreVariable);

}
```

Obtenemos la información justo como lo queremos:

```
Nombre de la VARIABLE GLOBAL: unaVariableString


Nombre de la VARIABLE GLOBAL: unaVariableInt
```

Poca información por ahora hemos obtenido. Veremos cómo obtener más, pues tenemos que entender otras cosas antes que nos van a dar mucha información extra.

Nos interesa conocer los Modificadores y los Tipos.

De semejante manera podremos recorrer métodos y constructores.

Por ejemplo para los métodos queremos saber su nombre y la cantidad de parámetros que se les pasa:

```
final Method[] metodos = objetoDeClassConInfoDeMiClase.getDeclaredMethods();

for (final Method metodo : metodos) {

        System.out.println("\nNombre del MÉTODO: " + metodo.getName());

        System.out.println("  Cantidad de parámetros: " + metodo.getParameterCount());

}
```

La consola nos dirá:

```
Nombre del MÉTODO: getUnaVariableInt

  Cantidad de parámetros: 1


Nombre del MÉTODO: getUnaVariableString

  Cantidad de parámetros: 1
```

Examinar Modificadores (Modifier)

Me queda refrescar un poco la teoría da Java al indicar que public, protected, private, abstract, static, final y otros menos utilizados, se llaman **Modificadores** (Modifier). Nombre dado al modificar el comportamiento de la variable, clase o función al que acompaña.

Pues bien, para obtenerlos de una variable (objeto tipo Field) es tan fácil como decir:

```
int modificador = variable.getModifiers();
```

Devuelve un número entero que indica que tipo de modificador es ¿Me tengo que aprender todos los códigos de los modificadores? Para nada, es tan fácil como preguntar a la clase Modifier sobre el modificador que queramos saber; algunas por ejemplo:

```
Boolean esPublic = Modifier.isPublic(modificador);

Boolean esPrivate = Modifier.isPrivate(modificador);

Boolean esFinal = Modifier.isFinal(modificador);

Boolean esStatic = Modifier.isStatic(modificador);
```

Como no quiero que te creas nada, vamos a verlo en el ejemplo:

```
for (Field variable : todasLasVariablesDeclaradas) {

        String nombreVariable = variable.getName();

        System.out.println("\nNombre de la VARIABLE GLOBAL: " + nombreVariable);



        Boolean esPublic = Modifier.isPublic(modificador);

        System.out.println("   Es public: " + esPublic);



        Boolean esPrivate = Modifier.isPrivate(modificador);

        System.out.println("   Es private: " + esPrivate);

}
```

Que devuelve por consola:

```
Nombre de la VARIABLE GLOBAL: unaVariableString

   Es public: true

   Es private: false
```

```
Nombre de la VARIABLE GLOBAL: unaVariableInt
   Es public: false
   Es private: true
```

Ya tenemos más información 

Lo mismo para los métodos y constructores. Podremos obtener sus modificadores, por ejemplo para los métodos:

```
for (final Method metodo : metodos) {
        System.out.println("Nombre del MÉTODO: " + metodo.getName());

        System.out.println("   Es public: " + Modifier.isPublic(metodo.getModifiers()));
        System.out.println("   Es private: " + Modifier.isPrivate(metodo.getModifiers()));
}
```

La consola nos muestra:

```
Nombre del MÉTODO: getUnaVariableInt
   Es public: false
   Es private: true

Nombre del MÉTODO: getUnaVariableString
   Es public: true
   Es private: false
```

Examinar Tipos (Type)

Dejo para el final los tipos, pues es un poco más especial, aunque muy parecido a Modifier. Para obtener el tipo de la variable (objeto tipo Field):

```
Type tipo = variable.getGenericType();
```

Para obtener el nombre del tipo es tan sencillo como:

```
String nombreTipoVariable = tipo.getTypeName();
```

Para los métodos y constructores es parecido.

En los métodos los tipos están ubicados en varios sitios: en el return, en los parámetros y en los throws (excepciones):

```
//Obtener el tipo del return
Type tipos = metodo.getGenericReturnType();


//Obtener todos los tipos de los parámetros
Type[] tipos = metodo.getGenericParameterTypes();


//Obtener todos los throws
Type[] tipos = metodo.getGenericExceptionTypes();
```

Y para obtener el nombre igual que antes, con getTypeName().

Te voy a poner el ejemplo anterior y aprovecho para ponerte la clase entera, así la puedes probar tú mismo:

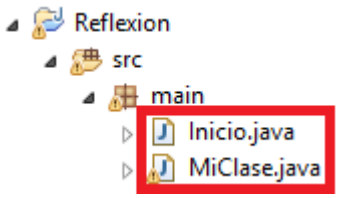
```
import java.lang.reflect.Field;
import java.lang.reflect.Modifier;
import java.lang.reflect.Type;


public class Inicio {


        public static void main(String[] args) {
```

```
                MiClase objetoDeMiClase = new MiClase();


                Class<? extends MiClase> objetoDeClassConInfoDeMiClase = objetoDeMiClase.getClas
s();


                //Recorrer todas las variables de la clase
                Field[] todasLasVariablesDeclaradas = objetoDeClassConInfoDeMiClase.getDeclaredF
ields();
                for (Field variable : todasLasVariablesDeclaradas) {
                        String nombreVariable = variable.getName();
                        System.out.println("\nNombre de la VARIABLE GLOBAL: " + nombreVariable)
;


                        Type tipo = variable.getGenericType();
                        String nombreTipoVariable = tipo.getTypeName();
                        System.out.println("  Tipo: " + nombreTipoVariable);


                        int modificador = variable.getModifiers();


                        Boolean esPublic = Modifier.isPublic(modificador);
                        System.out.println("  Es public: " + esPublic);


                        Boolean esPrivate = Modifier.isPrivate(modificador);
                        System.out.println("  Es private: " + esPrivate);
                }


                //Recorrer todos los métodos de la clase
                final Method[] metodos = objetoDeClassConInfoDeMiClase.getDeclaredMethods();
                for (final Method metodo : metodos) {
                        System.out.println("\nNombre del MÉTODO: " + metodo.getName());
```

```
                        System.out.println("  Cantidad de parámetros: " + metodo.getParameterCo
unt());



                        System.out.println("  Es public: " + Modifier.isPublic(metodo.getModifi
ers()));

                        System.out.println("  Es private: " + Modifier.isPrivate(metodo.getModi
fiers()));



                        System.out.println("  Tipo del return: "+metodo.getGenericReturnType().
getTypeName());

                        Type[] tipos = metodo.getGenericParameterTypes();



                        System.out.println("  Tipos de los parámetros:");

                        for (Type tipo : tipos) {

                                System.out.println("    "+tipo.getTypeName());

                        }

                }

        }



}
```

Tu proyecto debería quedar algo así (recuerdo que MiClase está al principio del artículo  y está el código para descargar al final del artículo):

Nota sobre la advertencia triangulo amarillo: para este ejemplo he declarado un método privado que no se usa, de ahí la advertencia que muestra el IDE (en mi caso Eclipse)



Y ya si lo ejecutas verás que nos da toda la información de cada variable por consola:

```
Nombre de la VARIABLE GLOBAL: unaVariableString
  Tipo: java.lang.String
  Es public: true
  Es private: false

Nombre de la VARIABLE GLOBAL: unaVariableInt
  Tipo: int
  Es public: false
  Es private: true

Nombre del MÉTODO: getUnaVariableInt
  Cantidad de parámetros: 1
  Es public: false
  Es private: true
  Tipo del return: int
  Tipos de los parámetros:
    int

Nombre del MÉTODO: getUnaVariableString
  Cantidad de parámetros: 1
  Es public: true
  Es private: false
  Tipo del return: class java.lang.String
  Tipos de los parámetros:
    class java.lang.String
```

Obtener el valor de la variable

Evidentemente utilizar Reflection para obtener el valor es un mal olor en el código (“bad smell” indica que probablemente lo que estamos programando se pueda hacer de una manera mucho mejor), pues si queremos obtener el valor lo mejor es obtenerlo directamente aunque sea llamándola directamente.

```
String textoObtenido = objetoDeMiClase.unaVariableString;
```

Pero si queremos obtener el valor de una variable privada, por ejemplo, no podemos acceder directamente a esta. O si estamos recorriendo un bucle con las variables tipo Field cuyos tipos son diferentes, pues no podemos acceder a estos tan directamente si queremos hacer cosas diferentes para cada tipo de variable. Para ello se nos permite obtener el valor por Reflection. Únicamente tenemos que utilizar el método get() del objeto de tipo Field que obtuvimos previamente (es decir, la variable de la clase). Es un poco peculiar este método get(), hay que pasarle la instancia de nuestra clase para poder obtener el valor que queremos; la razón es que el valor de las variables no se almacena en el Class generado, por ello hay que pasarle la instancia (esto puede confundir un poco por tener por un lado la instancia de Class, por otro la de Field y por otro la de nuestro objeto de MiClase; simplemente tenemos que pasar la instancia que tiene los valores, es decir, la instancia de MiClase que hemos llamado en el ejemplo “objetoDeMiClase”). El método get() devuelve un objeto de tipo Object, si lo queremos convertir por ejemplo a String, hay que castearlo a “(String)” como siempre se ha hecho en Java.

```
Field variableString = null;

String textoObtenido = null;

try {

        variableString = objetoDeClassConInfoDeMiClase.getField("unaVariableString");

        textoObtenido = (String) variableString.get(objetoDeMiClase);

} catch (IllegalArgumentException | IllegalAccessException | NoSuchFieldException | SecurityExce
ption e) {

        e.printStackTrace();

}

System.out.println("Valor de la variable de tipo String: " + textoObtenido);
```

Aparte de los try y catch que hay que ponerle al Field, también hay que añadir los del get() por si no obtenemos el tipo adecuado al que estamos obteniendo (imagina que nos llega un número de variable y nosotros esperábamos un String, ya tenemos el problema).

El resultado por consola será:

```
Valor de la variable de tipo String: Un Texto
```

Si es de tipo primitivo tenemos que hacer otra cosa; cosa gracias a la cual, nos ahorramos el tener que castear. Lo malo es que tenemos que llamar al getX() apropiado, donde X es un tipo primitivo como int, float, char, etc.

El listado completo para obtener los valores de tipo primitivo es:

```
byte valor = field.getByte(obj)

char valor = field.getChar(obj);

boolean valor = field.getBoolean(obj);

double valor = field.getDouble(obj);

float valor = field.getFloat(obj);

int valor = field.getInt(obj);

long valor = field.getLong(obj);

short valor = field.getShort(obj);
```

Si hacemos el ejemplo para obtener el primitivo int como hicimos en el anterior ejemplo nos devolverá un error pues no podemos acceder a los datos de la variable privada. Para ello la "hackearemos" al hacerla accesible con setAccessible(true). Quedará como sigue:

```
Field variableInt = null;

int intObtenido = -1;

try {

        variableInt = objetoDeClassConInfoDeMiClase.getDeclaredField("unaVariableInt");

        variableInt.setAccessible(true);

        intObtenido = variableInt.getInt(objetoDeMiClase);

} catch (IllegalArgumentException | IllegalAccessException | NoSuchFieldException | SecurityExce
ption e) {

        e.printStackTrace();

}

System.out.println("Valor de la variable de tipo int: " + intObtenido);
```

Y obtendremos por consola:

```
Valor de la variable de tipo int: 5
```

Nota de buena programación: La máxima en tu vida tiene que ser que las variables privadas sean privadas, no accedas a ellas al menos que sea lo último que puedas hacer. Es decir, lo más posible es que nunca vayas a tener que utilizar el método “setAccessible(true)”, que está prohibido salvo en unos pocos casos contados. En estos ejemplos que te pongo quiero que tengas toda la información, por si se diera el caso de necesidad. Algunas veces es necesario si estas creando cierto código; por ejemplo, algunas bibliotecas como Spring acceden a todas las variables y a todos sus valores directamente de las clases.

En conclusión, la mayoría de las veces nos vendrá mejor get() que nos devuelve el Object, pues o bien no sabremos qué tipo es el que nos llega o será una Clase creada por nosotros. Por ejemplo en un bucle como hicimos en Modifiers o Type.

```
for (Field variable : todasLasVariablesDeclaradas) {

        String nombreVariable = variable.getName();



        Object valorVariable = null;

        try {

                variable.setAccessible(true);

                valorVariable = variable.get(objetoDeMiClase);

        } catch (IllegalArgumentException | IllegalAccessException e) {

                e.printStackTrace();

        }

        System.out.println("\nValor de la variable " + nombreVariable + " es: " + valorVariable)
;

}
```

Devolverá por consola:

```
Valor de la variable unaVariableString es: Un Texto



Valor de la variable unaVariableInt es: 5
```

Recuerdo que de esta manera se obtiene por toString() el valor, para pintarlo por consola me sirve para este ejemplo, si quieres utilizar el int como int hay que hacer otras cosas.

Establecer el valor de las variables globales (Fields)

Hemos extraído datos, ahora vamos a meter.

Puede que no tenga mucho sentido cambiar el valor de una variable pública global de una clase, ya que lo podemos hacer directamente; pero ¿Si queremos cambiar el valor de una variable privada? Ya sé que esto suena alocado ¿Para qué cambiar el valor de variables privadas, para eso se pondría pública? Bueno, aquí las respuestas puede ser, cambio para realizar pruebas; o simplemente que no queremos que nadie las toque esas variables, pues vamos a hacer una biblioteca para los demás, pero sí que queremos tocarlas nosotros.

La razón por la que cambiar por Reflection una variable pública radica cuando se recorre un listado de variables que pueden ser públicas o privadas, cuyo nombre no conocemos, pues usamos Reflection y arreglado.

Nota sobre establecer variables globales: Reflection consume muchísimo tiempo de proceso, así que si tienes que elegir entre una asignación directa del valor y Reflection, la asignación directa no consume prácticamente nada

Para hacerlo, al objeto de tipo Field, simplemente le llamamos al **método set()** que **nos pedirá el objeto de la clase que tenga la variable que haya que cambiar, y el nuevo valor.** Si la variable es de tipo primitivo existen métodos setX(), donde X es int, float, etc.

En el siguiente ejemplo comprobarás que recorremos el listado de variables, en cada iteración del bucle se llamará a una que en nuestro caso puede ser de tipo String o int, para que no nos de error el programa y asignar un texto a un int, lo separamos con el if por los tipos. Podríamos haberlos separado por los nombres de las variables, luego en la práctica como no sabemos los nombres de las variables y además hay que “hardcoded” los nombres de las variables, resulta poco productivo.

```
for (Field variable : todasLasVariablesDeclaradas) {

        String nombreVariable = variable.getName();



        try {
```

```
                if (String.class == tipo) {

                        variable.set(objetoDeMiClase, "Texto cambiado en la variable");



                        System.out.println("  Valor cambiado de la variable " + nombreVariable
+ " es: " + objetoDeMiClase.unaVariableString);

                } else if (int.class == tipo) {

                        variable.setInt(objetoDeMiClase, 123456789);



                        variableInt.setAccessible(true);

                        intObtenido = variableInt.getInt(objetoDeMiClase);

                        System.out.println("  Valor cambiado de la variable " + nombreVariable
+ " es: " + intObtenido);

                                }



        } catch (IllegalArgumentException | IllegalAccessException e) {

                e.printStackTrace();

        }

}
```

Devuelve por consola:

```
Valor cambiado de la variable unaVariableString es: Texto cambiado en la variable



Valor cambiado de la variable unaVariableInt es: 12345
```

Invocar métodos

Sí, también se puede invocar cualquier método de una clase directamente por Reflection.

Como veremos en el ejemplo de test unitarios, es útil para ejecutar y por tanto probar métodos privados.

Para invocar un método de una clase es tan sencillo como llamar a invoke() del objeto tipo Method. Al método invoke() hay que pasarle el objeto de la clase donde esté el método a invocar, seguido de todos sus parámetros.

```
MiClase objetoDeMiClase = new MiClase();


Method metodo = objetoDeClassConInfoDeMiClase.getDeclaredMethod("getUnaVariableInt", int.class);


int valorRetorno = metodo.invoke(objetoDeMiClase, 100);
```

En un bucle hay puede haber problemas al añadir los valores de los parámetros, así que puede ser mejor pasar directamente Object como parámetros. En el siguiente ejemplo el valor a pasar lo elegimos con el nombre del método como ejemplo, y lo guardamos en un objeto de tipo Object para pasárselo a invoke(), de este modo invocamos el método y obtenemos el valor de retorno.

```
for (final Method metodo : metodos) {

        System.out.println("\nNombre del MÉTODO: " + metodo.getName());


        try {

                if (Modifier.isPrivate(metodo.getModifiers())) {

                        metodo.setAccessible(true);

                }


                Object valorPasar = null;


                if (metodo.getName() == "getUnaVariableString"){

                        valorPasar = " Añadir otra cosa";

                } else if (metodo.getName() == "getUnaVariableInt") {

                        valorPasar = 100;

                }


                Object valorRetornoMetodoInvocado = metodo.invoke(objetoDeMiClase, valorPasar);
```

```
                System.out.println("  Valor del método invocado: " + valorRetornoMetodoInvocado)
;



        } catch (IllegalAccessException | IllegalArgumentException | InvocationTargetException e
) {

                e.printStackTrace();

        }



}
```

La consola nos da:

```
Nombre del MÉTODO: getUnaVariableInt

  Valor del método invocado: 105



Nombre del MÉTODO: getUnaVariableString

  Valor del método invocado: Un Texto Añadir otra cosa
```

Crear nueva instancia de una clase

Y ya por último crear una instancia de un objeto con Reflection.

Muy útil cuando se quiere utilizar el patrón “Inyección de dependencias” (dicho rápidamente significa que no vamos a crear el objeto dentro de una clase, sino que nos lo va a pasar ya creado otra clase externa).

Muy sencillo, solo tenemos que obtener el constructor en un objeto de tipo Constructor con el que queramos instanciar el nuevo objeto. Recuerdo que si no creamos un constructor en la clase, tendrá por defecto siempre uno sin argumentos; y va a ser el vayamos a utilizar en este ejemplo, primero obtenemos el objeto de tipo Contructor con getConstructor(). Al método getConstructor() le podemos pasar los tipos de los argumentos (como por ejemplo int.class) de manera similar a como hicimos con los métodos normales; de este modo elegimos el que queramos. Desde este objeto Constructor llamamos a su método newInstance(), a la que tendremos que pasar el valor de los parámetros (por ejemplo, si a getConstructor() le pasamos int.class, aquí le podríamos pasar simplemente: 123).

Ya tenemos creado el objeto. El resto ya lo conoces: asignar valores a variables u obtenerlos, ejecutar métodos, etc.

```
try {

        Constructor constructorSinParametros = objetoDeClassConInfoDeMiClase.getConstructor();



        MiClase nuevoObjetoDeMiClase = (MiClase) constructorSinParametros.newInstance();



        Field variableString2 = nuevoObjetoDeMiClase.getClass().getField("unaVariableString");

        variableString2.set(nuevoObjetoDeMiClase, "Texto para nuevo objeto");



        System.out.println("\nNuevo objeto creado");

        System.out.println("  Valor de la variable asignada: " + nuevoObjetoDeMiClase.unaVariabl
eString);



} catch (InstantiationException | IllegalAccessException | IllegalArgumentException | Invocation
TargetException | NoSuchFieldException | SecurityException | NoSuchMethodException e) {

        e.printStackTrace();

}
```

Si quieres saber más usos, como utilizar Annotations de Java y como trabaja la "inyección de dependencias" en Frameworks profesionales como Spring, puedes seguir leyendo el siguiente artículo sobre Annotations de Java.

Test unitarios (Unit Test) de métodos privados con JUnit

Como norma, ley, obligación, moralidad, etc; ponle el nombre que quieras mientras se te quede grabado en el cerebro: lo último que se te tiene que pasar por la cabeza es usar Reflection para probar los métodos privados. Si nadie puede llamarlos desde fuera de una clase ¿Para qué vamos a probarlos

directamente? ni si quiera el IDE te sugiere métodos privados (por ejemplo en Eclipse, al escribir un objeto en otra clase, te mostrará los métodos públicos, nunca los privados de la clase de ese objeto). Primero te daré una charla de concienciación y al final te lo explicaré.

Las pruebas unitarias sirven para probar que los objetos de una clase funcionan correctamente, sin tener que abrir la clase para ver que hay dentro. Es decir, que para probar métodos privados hay que hacerlo indirectamente. Puedes pensarlo así, alguien te ha instanciado un objeto de una clase cuyo contenido de la clase no conoces, solo el objeto; por tanto conoces sus métodos públicos, nunca los privados. No tiene sentido esta prueba en muchos de los casos.

Siempre hay que probar todos los métodos “public” de cada una de las clases, eso sin duda. Si al probar estos métodos públicos todo funciona correctamente, entonces deducimos que los métodos “private” (lo mismo ocurre con los “protected”) también funcionan correctamente.

Es cierto que en algunos escasos y muy contadas excepciones. Bien sea porque el método privado haga muchas cosas por debajo con una estádistica muy elevada al fallo; o por que la probabilidad de que se ejecute dentro del funcionamiento del programa es muy pequeña; o que los métodos públicos que lo llaman generan contenido aleatorio (usan generadores aleatorios, o trabajan con datos que no conocemos a priori como la recepción de datos desde Internet); o pudiera ser que trabajáramos con hilos, por lo que apenas hay oportunidad de probar los métodos privados si no se cumplen ciertas condiciones de carrera; o bien que estemos trabajando en una biblioteca que no podemos ejecutar porque dependemos de código extra, una base de datos, etc que todavía no disponemos y no tenemos otra opción que probarlo de alguna manera. Estas son algunas de las excusas (todas discutibles según el punto de vista y las circunstancias) por las que pudiera ser justificable utilizar Reflection para probar métodos privados, por ello lo voy a explicar.

Mi recomendación: si puedes no utilices Reflection para realizar pruebas unitarias, salvo casos de necesidad absoluta.

Tenemos dos métodos en la clase “MiClase” que probar, uno público y otro privado. Como en toda prueba unitaria creamos el objeto, llamamos al método y le añadimos parámetros si tiene. El “return” del método nos devolverá un resultado.

Vamos a ver a modo de repaso y comparativa el ejemplo. Vamos a probar el método público getUnaVariableString() al que se le concatena otro String que le pasemos al método, para devolver un String formado por lo que tenía la clase más lo que le pasemos. Vamos a querer probar si lo que devuelve es “no es nulo” y si obtenemos “el texto esperado".

```
@Test
```

```
public void testGetUnaVariableString() {

        MiClase objetoDeMiClase = new MiClase();

        String valor = objetoDeMiClase.getUnaVariableString(" Cualquiera");



        assertNotNull(valor);

        assertEquals("Un Texto Cualquiera", valor);

}
```

Hasta aquí todo correcto y muy normal, un método público que probamos; efectivamente funciona como se esperaba, sin errores. ¿Y el método privado? Muy parecido, haremos lo anterior pero aplicando Reflection.

Ahora tenemos un método privado getUnaVariableInt() al que se le pasa un número entero para que se sume al que tiene dentro la clase, devuelve un “int” que será la suma. Y haremos las mismas pruebas que antes, que “no es nulo” y si obtenemos “el número esperado”.

Como hicimos antes, obtendremos el objeto tipo “Class” del que obtendremos el método y lo invocaremos para que se ejecute.

```
@Test

public void testGetUnaVariableInt() {

        MiClase objetoDeMiClase = new MiClase();



        Class<? extends MiClase> objetoDeClassConInfoDeMiClase = objetoDeMiClase.getClass();

        Method metodo;

        int valor;

        try {

                metodo = objetoDeClassConInfoDeMiClase.getDeclaredMethod("getUnaVariableInt", in
t.class);

                metodo.setAccessible(true);

                valor = (int) metodo.invoke(objetoDeMiClase, 2);



                assertNotNull(valor);

                assertEquals(7, valor);
```

```
        } catch (NoSuchMethodException | SecurityException | IllegalAccessException | IllegalArg
umentException | InvocationTargetException e) {

                e.printStackTrace();

        }

}
```

Ya está probado el método privado. Que sepas que me ha dolido escribirlo, así que evita su utilización 

La clase de test completa quedaría:

```
import static org.junit.Assert.assertEquals;

import static org.junit.Assert.assertNotNull;



import java.lang.reflect.InvocationTargetException;

import java.lang.reflect.Method;



import org.junit.Test;



public class MiClaseTest {



        @Test

        public void testGetUnaVariableString() {

                MiClase objetoDeMiClase = new MiClase();

                String valor = objetoDeMiClase.getUnaVariableString(" Cualquiera");



                assertNotNull(valor);

                assertEquals("Un Texto Cualquiera", valor);

        }



        @Test
```

```
        public void testGetUnaVariableInt() {

                MiClase objetoDeMiClase = new MiClase();



                Class<? extends MiClase> objetoDeClassConInfoDeMiClase = objetoDeMiClase.getClas
s();

                Method metodo;

                int valor;

                try {

                        metodo = objetoDeClassConInfoDeMiClase.getDeclaredMethod("getUnaVariabl
eInt", int.class);

                        metodo.setAccessible(true);

                        valor = (int) metodo.invoke(objetoDeMiClase, 2);



                        assertNotNull(valor);

                        assertEquals(7, valor);

                } catch (NoSuchMethodException | SecurityException | IllegalAccessException | Il
legalArgumentException | InvocationTargetException e) {

                        e.printStackTrace();

                }

        }



}
```

Código

Tienes todo el código en: https://github.com/jarroba/Reflection

Bibliografía

- https://docs.oracle.com/javase/tutorial/reflect/index.html
- https://en.wikipedia.org/wiki/Reflection_(computer_programming)
- https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html