PROG-3

UD 7

REFLEXION

Pedro J. Ponce de León

Versión 20131120

Depto. Lenguajes y Sistemas Informáticos

Universitat d'Alacant
Universidad de Alicante

Indice

Motivación. Qué es la reflexión

- Cuando nos miramos en un espejo:
  - Vemos nuestro reflejo
  - Podemos actuar según lo que vemos (ajustarnos la corbata, maquillarnos,...)
- En programación de ordenadores:
  - La reflexión es una infraestructura del lenguaje que permite a un programa conocerse y manipularse a sí mismo en tiempo de ejecución. Consiste en metadatos más operaciones que los manipulan.

(Un *metadato* es un dato que proporciona información sobre otro dato. Por ejemplo, una clase que contiene información sobre otra clase: cual es el nombre de esa clase, qué atributos o métodos tiene, etc.)

Motivación. Qué es la reflexión

- La reflexión puede usarse para
  - Construir nuevos objetos y arrays
  - Acceder y modificar atributos de instancia o de clase
  - Invocar métodos de instancia y estáticos
  - Acceder y modificar elementos de arrays
  - etc...

- La reflexión permite hacer esto sin necesidad de conocer el nombre de las clases en tiempo de compilación. Esta información puede ser proporcionada en forma de datos en tiempo de ejecución (por ejemplo, en un String).

Motivación. Qué es la reflexión

- Ejemplo
  - De un fichero de texto, leemos la cadena “Barco”, que indica que hay que crear un objeto de esa clase. Mediante reflexión, el programa actuará de la siguiente forma
    - Se preguntará ¿existe una clase 'Barco' a la que tenga acceso?
    - Si es así, la cargará en memoria.

```
Class c = Class.forName(“Barco”)
```

    - Invocará a un constructor de la clase para crear un objeto Barco.

```
Object obj = c.newInstance();
```

Fíjate que en ninguna de las dos líneas de código aparece 'Barco' como tipo.

El API de reflexión de Java

- Java proporciona dos formas de obtener información sobre los tipos en tiempo de ejecución
  - **RTTI tradicional** (upcasting, downcasting)
    - Cuando el tipo de un objeto está disponible en tiempo de compilación y ejecución
  - **Reflexión**
    - Cuando el tipo de un objeto puede no estar disponible en tiempo de compilación y/o ejecución.

El API de reflexión de Java

- El API de reflexión de Java
  - **`java.lang.Class`**
  - **`java.lang.reflect.*`**

- Representa, o refleja, las clases, interfaces y objetos de la JVM en ejecución.

- Introducida en JDK 1.1 para dar soporte a la especificación JavaBeans.

API de reflexion de Java

- **Clases del API de reflexión**

`java.lang.reflect`

- El paquete de reflexión
- Introducido en JDK 1.1

`java.lang.reflect.Array`

- Proporciona métodos estáticos para crear y acceder dinámicamente a los arrays.

`java.lang.reflect.Member`

- Interfaz que refleja información sobre un miembro de una clase (atributo, método o constructor)

API de reflexion de Java

- **Clases del API de reflexión (cont.)**

`java.lang.reflect.Constructor`

- Proporciona información y acceso sobre un método constructor

`java.lang.reflect.Field`

- Proporcionan información y acceso dinámico a un atributo de una clase.
- El atributo puede ser de clase (`static`) o de instancia.

`java.lang.reflect.Method`

- Proporciona información y acceso a un método.

API de reflexion de Java

- **Clases del API de reflexión (cont.)**

  `java.lang.Class`
  - Representa clases e interfaces

  `java.lang.Package`
  - Proporciona información sobre un paquete.

  `java.lang.ClassLoader`
  - Clase abstracta. Proporciona servicios para cargar clases dinámicamente.

1. El objeto **Class**

- Toda clase que se carga en la JVM tiene asociado un objeto de tipo **`Class`**
  - Corresponde a un fichero .class
  - El *cargador de clases* (`java.lang.ClassLoader`) es el responsable de encontrar y cargar una clase en la JVM.

- **Carga dinámica de clases**: al instanciar un objeto…
  - JVM comprueba si la clase ya ha sido cargada
  - Localiza y carga la clase en caso necesario.
  - Una vez cargada, es usada para instanciar el objeto.

1. El objeto Class

Un objeto Class contiene información sobre una clase:

- Sus métodos
- Sus campos
- Su superclase
- Los interfaces que implementa
- Si es o no un array
- ...

El objeto Class

- **`Class.forName(String)`**
  - Obtiene el objeto Class correspondiente a una clase cuyo nombre se pasa como argumento.

```
try
{
    // busca y carga la clase B,
   // si no estaba ya cargada
   Class c = Class.forName ("B");
    // c apuntará a un objeto Class
  // que representa a la clase B
}
catch (ClassNotFoundException e)
{ // B no existe, es decir,
  // no está en el CLASSPATH
   e.printStackTrace ();
}
```

El objeto Class

- En el tema anterior, hemos visto otras formas de obtener el objeto Class y/o acceder a la información que contiene:

  - **Literales de clase**

```
Circulo.class
Integer.class
int.class
```

  - **instanceOf**

```
if (x instanceof Circulo)
    ((Circulo) x).setRadio(10);
```

El objeto Class

- **Métodos en Class**

```
public String getName( );
```
- Devuelve el nombre de la clase reflejada por este objeto Class.

```
public boolean isInterface( );
```
- Devuelve cierto si la clase es un interfaz.

```
public boolean isArray( );
```
- Devuelve cierto si la clase es un array.

```
public Class getSuperclass( );
```
- Devuelve el objeto Class que representa a la clase base de la clase actual.

El objeto Class

- **Métodos en Class**

```
public Class[] getInterfaces( );
```
- Devuelve un array de objetos Class que representan los interfaces implementados por esta clase.

```
public Object newInstance( );
```
- Crea una instancia de esta clase (invocando al constructor por defecto).

El objeto Class

- **Métodos en Class**

`public Constructor[] getConstructors( );`
- Devuelve un array con todos los constructores públicos de la clase actual.

`public Method[] getDeclaredMethods( );`
- Devuelve un array de todos los métodos públicos y privados declarados en la clase actual.

`public Method[] getMethods( );`
- Devuelve un array de todos los métodos públicos en la clase actual, así como los declarados en todas las clases base o interfaces implementados por esta clase.

El objeto Class

- **Métodos en Class**

```
public Method getMethod( String methodName,
                         Class[] paramTypes );
```

- Devuelve un objeto Method que representa el método público identificado por su nombre y tipo de parámetros, declarado en esta clase o heredado de una clase base.

```
public Method getDeclaredMethod( String
methodName, Class[] paramTypes );
```

- Devuelve un objeto Method que representa el método público identificado por su nombre y tipo de parámetros, declarado en esta clase. El método puede ser privado.

- **La clase Array**

```
public class Array {

    // todo métodos static y public:
    int getLength( Object arr );
    Object newInstance( Class elements, int length );
    Object get( Object arr, int index );
    void set( Object arr, int index, Object val );

    // Varias versiones especializadas, como...
    int getInt( Object arr, int index );
    void setInt( Object arr, int index, int val );
}
```

La clase Array

- **Ejemplos usando Array**

```
Perro[] perrera = new Perro[10];
.
int n = Array.getLength( perrera );

// asigna uno de los elementos del array
Array.set( perrera, (n-1), new Perro( "Spaniel" ) );

// Obtiene un objeto del array, determina su clase y
muestra su valor:

Object obj = Array.get( perrera, (n-1) );
Class c1 = obj.getClass( );
System.out.println( c1.getName( ) + "-->"
    + obj.toString( ) );
```

La clase Array

- **Dos formas de declarar un array**

```
// 1:
Perro[] perrera = new Perro[10];

// 2:
Class c1 = Class.forName( "Perro" );
Perro[] perrera = (Perro[]) Array.newInstance( c1, 10 );
```

La clase Array

- **Ejemplo: expandir un array**

Enunciado del problema:

Escribir una función que reciba un array arbitrario, reserve memoria para dos veces el tamaño del array, copie los datos al nuevo array y lo devuelva como resultado.

La clase Array

- **Ejemplo: expandir un array**

```
// Este código no funcionará
public static Object[] doubleArrayBad( Object[] arr
{
    int newSize = arr.length * 2 + 1;

    Object[] newArray = new Object[ newSize ];

    for( int i = 0; i < arr.length; i++ )
        newArray[ i ] = arr[ i ];

    return newArray;
}
```

La clase Array

- **Ejemplo: expandir un array**

```
// Este código no funcionará
public static Object[] doubleArrayBad( Object[] arr )
{
    int newSize = arr.length * 2 + 1;

    Object[] newArray = new Object[ newSize ];

    for( int i = 0; i < arr.length; i++ )
        newArray[ i ] = arr[ i ];

    return newArray;
}
```

Este método siempre devuelve un array de Object, en lugar de un array del mismo tipo que el array que estamos copiando

La clase Array

- **Ejemplo: expandir un array**

Usemos reflexión para obtener el tipo del array:

```
public Object[] doubleArray( Object[] arr )
{
    Class c1 = arr.getClass( );
    if( !c1.isArray( ) ) return null;

    int oldSize = Array.getLength( arr );
    int newSize = oldSize * 2 + 1;

    Object[] newArray = (Object[]) Array.newInstance(
        c1.getComponentType( ), newSize );

    for( int i = 0; i < arr.length; i++ )
        newArray[ i ] = arr[ i ];

    return newArray;
}
```

Interface Member

- Representa a un miembro de una clase
- Implementado por Constructor, Method, y Field

`Class getDeclaringClass( )`

- Devuelve el objeto Class que representa a la clase donde se declara el miembro.

`int getModifiers( )`

- Devuelve un entero que representa los modificadores aplicados a este miembro.

`String getName( )`

- Devuelve el nombre simple del miembro.

Clase Method

Con un objeto Method, podemos...

- Obtener su nombre y lista de parámetros e invocarlo.
- Obtener un método a partir de una signatura, u obtener una lista de todos los métodos con esa signatura.
- Para especificar una signatura, hay que crear un array de objetos Class que representen los tipos de los parámetros del método (será de longitud cero si el método no lleva parámetros)

Clase Method

```
public class Method implements Member
{
    public Class getReturnType( );
    public Class[] getParameterTypes( );
    public String getName( );
    public int getModifiers( );
    public Class[] getExceptionTypes( );
    public Object invoke( Object obj, Object[] args);
}
```

- Los modificadores son almacenados como un patrón de bits; la clase Modifier tiene métodos para interpretar los bits.

Clase Method. Ejemplos

- Obtener el nombre de un método:

```
(Method meth;)
String name = meth.getName( );
```

- Obtener un array de tipos de los parámetros:

```
Class parms[] = meth.getParameterTypes( );
```

- Obtener el tipo de retorno de un método:

```
Class retType = meth.getReturnType( );
```

Clase Method.

- **`Method.invoke()`**

```
public Object invoke(Object obj, Object[] args)
```

  - Si los parámetros o el tipo de retorno son tipos primitivos, se usan los objetos Class de las clases 'wrapper' equivalente.
    - ejemplo: `Integer.class`
  - El primer argumento es el objeto receptor (o null para métodos estáticos)
  - El segundo argumento es la lista de parámetros.
  - Inconvenientes de usar invoke( ):
    - Se ejecuta más lento que una llamada normal
    - Hay que ocuparse de todas las posibles excepciones verificadas.
    - Se pierden muchas de las comprobaciones que se realizan en tiempo de compilación.

Clase Method. Ejemplos

- `Method.invoke()` y excepciones

- Si el método invocado lanza una excepción al ejecutarlo, invoke() lanzará InvocationTargetException
  - La excepción del método se puede obtener llamando a al método getException de InvocationTargetException
- Otras excepciones a tener en cuenta al usar invoke();
  - ¿Se cargó la clase? ClassNotFoundException
  - ¿Se encontró el método? NoSuchMethodException
  - ¿Es el método accesible? IllegalAccessException

Clase Method

- **Pasos para invocar un método mediante reflexión**

- Obtener un objeto Class, c.
- Obtener de c un objeto Method, m:
  - Crear un array de tipos de los parámetros del método a invocar.
  - Llamar a getDeclaredMethod( ), pasándole el nombre del método y el array de tipos. Devuelve m.
- Crear un array de Object que contenga los argumentos que queremos pasarle al método (segundo argumento de m.invoke()).
  - new String[ ] { "Agua", "Fuego" }
- Pasar el objeto receptor (o null si el método es estático) como primer argumento de m.invoke().
- Llamar a m.invoke( ), y capturar la excepción InvocationTargetException

Ejemplo: Invocar a main( )

Llamada: main(String[] args)

simplificado, sin control de errores:

```
Class cl = Class.forName( className );
Class[] paramTypes = new Class[] { String[].class };
Method m = cl.getDeclaredMethod( "main", paramTypes );
Object[] args = new Object[]
       { new String[] { "Agua", "Fuego" } }
m.invoke( null, args );
```

Invocar a un constructor

Circulo(Coordenada centro, float radio)

Llamar a getConstructor( ), despues a newInstance( ), capturar InstantiationException

```
Class c1 = Class.forName("Circulo");

Class[] paramTypes = new Class[] {Coordenada.class,
                                  float.class };

Constructor m = c1.getConstructor( paramTypes );


Object[] arguments = new Object[] {
   new Coordenada(10,20), 30 };


Figura2D c = (Figura2D) m.newInstance(arguments);
```

Obtener objetos Field a partir de Class

```
public Field getField( String name )
       throws NoSuchFieldException, SecurityException
```

Devuelve un objeto Field público.

```
public Field[] getFields()
       throws SecurityException
```

Devuelve un array de atributos publicos en la clase actual o sus superclases.

```
public Field[] getDeclaredFields()
       throws SecurityException
```

Devuelve un array de atributos declarados en la clase actual.

Cosas que se pueden hacer con objetos Field

- Cambiar el tipo de acceso al atributo.
- Obtener el nombre del atributo: String getName( )
- Obtener su tipo – Class getType( )
- Obtener su valor o asignarlo – Object get( )
- Asignarle un valor - void set(Object receptor, Object valor)
- Comprobar si es igual a otro – boolean equals(Object obj)
- Obtener la clase donde se declara – Class getDeclaringClass()
- Obtener sus modificadores - getModifiers( )

Otros métodos en Field

- Métodos "get" específicos:
  - boolean getBoolean(Object receptor) obtiene el valor de un atributo de tipo booleano.
  - Tambien: getByte, getChar, getDouble, getFloat, getInt, getLong, y getShort
- Métodos "set" específicos:
  - void setBoolean(Object receptor, boolean z) asigna el valor 'z' al atributo en el objeto especificado.
  - También: setByte, setChar, setDouble, setFloat, setInt, setLong, and setShort

Objetos Field

Modificar y acceder a un atributo por reflexión

Métodos get() y set() en Field

- Por ejemplo:

```
Object d = new Heroe( );
Field f = d.getClass( ).getField( "fuerza" );
System.out.println( f.get( d ) );
```

- Posibles excepciones:
  - NoSuchFieldException, IllegalAccessException

Mitos sobre la reflexión

- "La reflexión sólo es útil para trabajar con componentes JavaBeans™"
- "La reflexión es demasiado compleja para usarla en aplicaciones de propósito general"
- "La reflexión reduce el rendimiento de las aplicaciones"
- "La reflexión no puede usarse en aplicaciones certificadas con el estándar *100% Pure Java*"

Mitos sobre la reflexión

"La reflexión sólo es útil para trabajar con componentes JavaBeans™"

- Falso
- Es una técnica común en otros lenguajes orientados a objetos puros, como Smalltalk y Eiffel
- Ventajas
  - Ayuda a mantener software robusto
  - Permite a las aplicaciones ser más
    - Flexibles
    - Extensibles
    - Uso de plug-ins

Mitos sobre la reflexión

"La reflexión es demasiado compleja para usarla en aplicaciones de propósito general"

- Falso
- Para la mayoría de propósitos, el uso de reflexióin requiere únicamente saber llamar a unos pocos métodos.
- Es relativamente fácil adquirir dicha destreza
- La reflexíon puede mejorar la reusabilidad del código.

Mitos sobre la reflexión

"La reflexión reduce el rendimiento de las aplicaciones"

- Falso
- En realidad, la reflexión puede aumentar el rendimiento del código.
- Ventajas
  - Puede reducir y eliminar mucho código condicional.
  - Puede simplificar el código fuente y el diseño.
  - Puede aumentar enormemente las capacidades de una aplicación

Mitos sobre la reflexión

"La reflexión no puede usarse en aplicaciones certificadas con el estándar *100% Pure Java*"

- Falso
- Sólo hay algunas restricciones en la certificación:
  - "El programa debe limitar las llamadas a métodos a clases que sean parte del programa o del JRE"

Sun Microsystems, 100% Pure Java™ Certification Guide, version 3.1, May 2000.

Lo que no podemos hacer con reflexión

- ¿Cuales son las clases derivadas de una clase dada?
  - No es posible debido a la carga dinámica de clases en la JVM
- ¿Qué método se está ejecutando en este momento?
  - No es el propósito de la reflexión
  - Algunas librerías Java permiten saberlo.

Ejemplos de uso de la reflexión

- Navegar por la jerarquía de herencia
- Eliminar código condicional

Navegando por la jerarquía de herencia

- Encontrar un método heredado

Se trata de buscar un método hacia arriba en la jerarquía de herencia.

Funciona tanto para métodos públicos como no-públicos.

Sólo podemos ir 'hacia arriba', hacia la clase base

Navegando por la jerarquía de herencia

```
Method buscaMetodo(Class cls, String nombreMetodo,
Class[] tiposParam)
{
   Method met = null;
   while (cls != null) {
      try {
         met = cls.getDeclaredMethod(nombreMetodo,
               tiposParam);
         break;
      } catch (NoSuchMethodException ex) {
         cls = cls.getSuperclass();
      }
   }
   return met;
}
```

Ejemplo: buscaMetodo(Figura2D.class, “equals”, new Class[] {})

Eliminar código condicional

Considera este método 'factoría': Dado el nombre de una clase, crea un objeto de esa clase

```
public static Figura2D creaFigura2D (String s)
{
    Figura2D temp = null;
    if (s.equals ("Circulo"))
       temp = new Circulo ()
    else
       if (s.equals ("Cuadrado"))
          temp = new Cuadrado ();
       else
          if (s.equals ("Triangulo")
             temp = new Triangulo ();
          else
            // …
    return temp;
}
```

Eliminar código condicional

El mismo método, con reflexión:

```
public static Figura2D creaFigura2D (String s)
{
    Figura2D temp = null;
    try
    {
        temp = (Figura2D) Class.forName (s).newInstance ();
    }
    catch (Exception e)
    {
    }
    return temp;
}
```

Bibliografía

- **Introduction to Object Oriented Programming, 3rd Ed,**
  Timothy A. Budd
  Cap. 25 : Reflection and Introspection

- **Java reflection in action**
  - Ira R. Forman and Nate Forman, 2004
  http://www.manning.com/forman/

- **Piensa en Java, 4ª ed.**
  - Bruce Eckl
  Cap. 14 : Información de tipos