Java
Reflection

Programação III

José Luis Oliveira; Carlos Costa

What is reflection?

- When you look in a mirror:
  - You can see your reflection
  - You can act on what you see, for example, straighten your tie
- In computer programming:
  - Reflection is infrastructure enabling a program can see and manipulate itself
  - It consists of metadata plus operations to manipulate the metadata
- Meta means self-referential
  - So metadata is data (information) about oneself

What is reflection?

Reflection

- "Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution." [Demers and Malenfant]

Java Tutorials

- "Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine."

- "… advanced feature … a powerful technique … can enable applications to perform operations which would otherwise be impossible."

Java looking at Java

- Reflection permite que um programa se examine a si mesmo.

Podemos:

- Determinar a classe de um objecto
- Descobrir toda a informação associada a determinada classe:
  - access modifiers, superclass, fields, constructors, and methods
- Obter informação relativa ao conteúdo de uma interface.
- Mesmo sem saber o nome (classes, métodos,…) podemos:
  - Criar uma instância de uma classe
  - ler/modificar variáveis
  - Invocar métodos
  - Criar e manipular vectores de objectos

Utilização de Java Reflection

- JavaBeans (component architectures)
- Database applications
- Serialization
- Scripting applications
- Runtime Debugging/Inspection Tools
- etc

Acesso a metadados

- Java armazena metadados em classes
  - Metadata for a class: `java.lang.Class`
  - Metadata for a constructor: `java.lang.reflect.Constructor`
  - Metadata for a field: `java.lang.reflect.Field`
  - Metadata for a method: `java.lang.reflect.Method`
- Podemos aceder à Class de um objecto de duas formas:

```
Class<?> cl1 = Class.forName("java.util.Properties");
```

ou

```
Object obj = ... // e.g. new StringBuffer("Teste");
Class<?> cl2 = obj.getClass();
```

- As classes do package Reflection são inter-dependentes
  - Exemplos a seguir…

Metadata de tipos primitivos e vectores

- Java associa uma instância de Class a cada tipo primitivo:

```
Class<?> c1 = int.class;
Class<?> c2 = boolean.class;
Class<?> c3 = void.class;
```

- Podemos usar Class.forName() para aceder à classe de um vector

```
Class<?> c4 = byte.class;              // byte
Class<?> c5 = Class.forName("[B");  // byte[]
Class<?> c6 = Class.forName("[[B"); // byte[][]
```

- Encoding scheme utilizado por Class.forName()

  B → byte; C → char; D → double; F → float; I → int; J → long;
  Lclass-name → class-name[]; S → short; Z → boolean

  Use as many “[”s as there are dimensions in the array

Reflection API - Class

```
public final class Class<T>
extends Object
implements Serializable, GenericDeclaration,
       Type, AnnotatedElement

       static Class<?> forName(String className);
       T newInstance();
       Field[] getFields();
       Method[] getMethods();
       boolean isInstance(Object obj);
       String getName();

       getInterfaces(), getSuperclass(),
       getModifiers(), getField(), getMethod(),...
```

```
void printClassName(Object obj) {
        System.out.println("The class of " + obj +
                           " is " + obj.getClass().getName());
    }
```

Reflection API - Field

```
public final class Field
extends AccessibleObject
implements Member
```

```
Object get(Object obj);
void   set(Object obj, Object val);

getType(), getDeclaringClass(),

setDouble(…), setInt(…), .....
```

```
Field[] flds = someObject.getClass().getFields();
for (Field f: flds)
        System.out.println(f.getName());
```

Reflection API - Method

```
public final class Method
extends AccessibleObject
implements GenericDeclaration, Member
```

```
Object invoke(Object obj, Object... args);

Class<?> getReturnType();

Class<?>[] getParameterTypes(),

getExceptionTypes(), getDeclaringClass(),..
```

```
Method methods[] = someClass.getMethods();
for (Method m: methods)
        System.out.println(m);
```

Reflection API - others...

```
class Constructor<T>
class AccessibleObject
        ← Constructor,
        ← Field,
        ← Method
class Array;
class Proxy;
Class Modifier
```

java.lang.reflect

Interfaces

AnnotatedElement
GenericArrayType
GenericDeclaration
InvocationHandler
Member
ParameterizedType
Type
TypeVariable
WildcardType

Classes

AccessibleObject
Array
Constructor
Field
Method
Modifier
Proxy
ReflectPermission

Exceptions

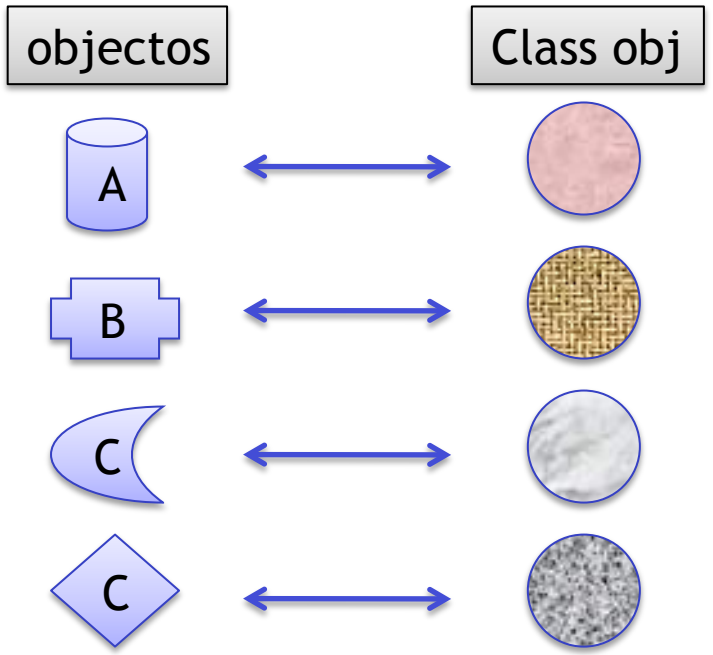
InvocationTargetException
MalformedParameterizedTypeException
UndeclaredThrowableException

Errors

GenericSignatureFormatError

A Classe Class

- Para cada objecto carregado pela JVM, existe um objecto do tipo *Class associado*.
  - *Os tipos primitivos também são representados por objectos Class.*
- As instâncias do tipo *Class* armazenam informações sobre a classe:
  - Nome da classe
  - Herança
  - Interfaces Implementadas
  - Métodos
  - Atributos
- *Permite invocar métodos e referenciar atributos*



Métodos de java.lang.Class - 1

- public static Class<?> forName(String className)
  - returns a Class object that represents the class with the given name
- public String getName()
  - returns the full name of the Class object, such as "java.lang.String".
- public int getModifiers()
  - returns an integer that describes the class modifier: public, final or abstract
- public T newInstance()
  - creates an instance of this class at runtime

Exemplo - newInstance

```
public class ReflectionNew {
    public static void main(String[] args) throws Exception {
          Class<?> sc = Class.forName("aula5_1.Circulo");

          System.out.println("Name = " + sc.getName());
          System.out.println("SimpleName = " + sc.getSimpleName());
          System.out.println("CanonicalName = " + sc.getCanonicalName());

          Class<?>[] paramTypes = { Double.TYPE, Double.TYPE, Double.TYPE };
          Constructor<?> cons = sc.getConstructor(paramTypes);
          Object ar[] = { 2, 4, 10 };
          Object theObject = cons.newInstance(ar);
          System.out.println("New object: " + theObject);

          Constructor<?> cs = sc.getConstructor(new Class<?>[]{Double.TYPE});
          System.out.println("New object: " + cs.newInstance(new Object[]{20}));
    }
}
```

```
Name = aula5_1.Circulo
SimpleName = Circulo
CanonicalName = aula5_1.Circulo
New object: Circulo de Centro (2.0,4.0) e de raio 10.0
New object: Circulo de Centro (0.0,0.0) e de raio 20.0
```

Exemplo - Modifiers

```
public class SampleModifier {
    public static void main(String[] args) {
          printModifiers(new String());
          printModifiers(new SampleModifier());
    }
    public static void printModifiers(Object o) {
          Class<?> c = o.getClass(); // returns the Class object of o
          System.out.print("***** Class " + c.getName()+" : ");

          int m = c.getModifiers(); // return the class modifiers
          if (Modifier.isPublic(m)) // checks if is public
                    System.out.print("public ");
          if (Modifier.isAbstract(m)) // checks if it is abstract
                    System.out.print("abstract ");
          if (Modifier.isFinal(m)) // checks if it is final
                    System.out.print("final "); System.out.println();
    }
}
```

```
***** Class java.lang.String : public final
***** Class reflection.SampleModifier : public
```

Métodos de java.lang.Class - 2

- public Class[] getClasses()
  - returns an array of all inner classes of this class
- public Constructor getConstructor(Class[] params)
  - returns all public constructors of this class whose formal parameter types match those specified by params
- public Constructor[] getConstructors()
  - returns all public constructors of this class

Exemplo - Construtores

```
public class Reflection2 {
    public static void main(String[] args) throws InstantiationException,
                                IllegalAccessException {
        String s="Mar";
        Class<?> sc = s.getClass();
        System.out.println("\n******** Construtores **********************\n");
        Constructor<?> contrs[] = sc.getConstructors();
        for (Constructor<?> c: contrs)
                System.out.println(c);
    }
}
```

```
******** Construtores **********************
public java.lang.String()
public java.lang.String(java.lang.String)
public java.lang.String(char[])
public java.lang.String(char[],int,int)
public java.lang.String(int[],int,int)
public java.lang.String(byte[],int,int,int)
public java.lang.String(byte[],int)
public java.lang.String(byte[],int,int,java.lang.String) throws java.io.UnsupportedEncodingException
public java.lang.String(byte[],int,int,java.nio.charset.Charset)
public java.lang.String(byte[],java.lang.String) throws java.io.UnsupportedEncodingException
public java.lang.String(byte[],java.nio.charset.Charset)
public java.lang.String(byte[],int,int)
public java.lang.String(byte[])
public java.lang.String(java.lang.StringBuffer)
public java.lang.String(java.lang.StringBuilder)
```

Métodos de java.lang.Class - 3

- public Field getField(String name)
  - returns an object of the class Field that corresponds to the instance variable of the class that is called name
- public Field[] getFields()
  - returns all accessible public instance variables of the class
- public Field[] getDeclaredFields()
  - returns all declared fields (instance variables) of the class

Exemplo - Fields

```
public class Reflection2 {
   public static void main(String[] args) throws InstantiationException,
                         IllegalAccessException {
        String s="Mar";
        Class<?> sc = s.getClass();
        System.out.println("\n******** Fields      **********************\n");
        Field fields[] = sc.getFields();
        for (Field f: fields)
                System.out.println(f);
   }
```

```
public static final java.util.Comparator java.lang.String.CASE_INSENSITIVE_ORDER
```

Exemplo - Fields

```
public static void main(String[] args) throws Exception {
   Class<?> sc = Class.forName("aula5_1.Circulo");
   System.out.println("\n******** Fields      **********************\n");
   Field fields[] = sc.getFields();
   for (Field f: fields)
       System.out.println(f);
   System.out.println("\n******** Declared Fields ******************\n");
   Field dfields[] = sc.getDeclaredFields();
   for (Field f: dfields)
       System.out.println(f);
   System.out.println("\n******** raio Field      ********************\n");
   Field field = sc.getField("raio");  // deve usar-se getDeclaredField
   System.out.println(field);
}
```

```
******** Fields      **********************

******** Declared Fields     *************
private double aula5_1.Circulo.raio
******** raio Field      ******************
      Exception in thread "main" java.lang.NoSuchFieldException: aula5_1.Circulo.raio
      at java.lang.Class.getField(Class.java:1520)
      at reflection.Reflection2.main(Reflection2.java:39)
```

Ler atributos

```
class SampleGet {
    public static void main(String[] args) {
        Rectangle r = new Rectangle(100, 325);
        printHeight(r);
    }
    static void printHeight(Rectangle r) {
        Field heightField; //declares a field
        Integer heightValue;
        Class<?> c = r.getClass(); //get the Class object
        try {
            heightField = c.getField("height"); //get the field object
            heightValue = (Integer)heightField.get(r); //get the value of the field
            System.out.println("Height: " + heightValue.toString());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
Height: 325
```

Modificar atributos

```
class SampleSet {
   public static void main(String[] args) {
      Rectangle r = new Rectangle(100, 20);
      System.out.println("original: " + r.toString());
      modifyWidth(r, new Integer(300));
      System.out.println("modified: " + r.toString());
   }
   static void modifyWidth(Rectangle r, Integer widthParam ) {
      Field widthField; //declare a field
      Integer widthValue;
      Class<?> c = r.getClass(); //get the Class object
      try {
          widthField = c.getField("width"); //get the field object
          widthField.set(r, widthParam); //set the field to widthParam =300
      } catch (Exception e ) {
          // . . .
      }
   }
}
```

```
original: java.awt.Rectangle[x=0,y=0,width=100,height=20]
modified: java.awt.Rectangle[x=0,y=0,width=300,height=20]
```

Métodos de java.lang.Class - 4

- public Method getMethod(String name, Class[] params)
  - returns an object Method that corresponds to the method called name with a set of parameters params
- public Method[] getMethods()
  - returns all accessible public methods of the class
- public Method[] getDeclaredMethods()
  - returns all declared methods of the class.

- public Package getPackage()
  - returns the package that contains the class
- public Class getSuperClass()
  - returns the superclass of the class

Exemplo - Métodos

```
public class Reflection2 {
   public static void main(String[] args) throws InstantiationException,
                                IllegalAccessException {
         String s="Mar";
         Class<?> sc = s.getClass();
         System.out.println("\n******** Métodos      **********************\n");
         Method methods[] = sc.getMethods();
         for (Method m: methods)
                  System.out.println(m);
   }
}
```

```
******** Métodos      **********************

public boolean java.lang.String.equals(java.lang.Object)
public java.lang.String java.lang.String.toString()
public int java.lang.String.hashCode()

.....
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()
```

Manipulação de vectores

```
public static void main(final String[] args) {
   try {
      String[] z = new String[] { "Jim", "John", "Joe" };
      final Class<?> type = z.getClass();
      if (!type.isArray()) {
         throw new IllegalArgumentException();
      } else {
         System.out.println("Name = "+ type.getName() +
               "\nType = "+type.getComponentType());
      }
   } catch (final Exception ex) {
      ex.printStackTrace();
   }
}
```

```
Name = [Ljava.lang.String;
Type = class java.lang.String
```

Manipulação de vectores

```
public class ArrayNew {
    public static void main(String[] args) throws ClassNotFoundException {
          System.out.println(createNativeArray("int", 12).getClass());
          System.out.println(createNativeArray("boolean", 10, 10).getClass());
          System.out.println(createNativeArray("double", 5, 5, 5).getClass());
    }
    public static Object createNativeArray(String typeName, int... dim)
    throws ClassNotFoundException {
          Class<?> clazz = null;
          if ("int".equals(typeName)) {
                    clazz = Integer.TYPE;
          } else if ("boolean".equals(typeName)) {
                    clazz = Boolean.TYPE;
          } else if ("double".equals(typeName)) {
                    clazz = Double.class;
                    // All other native types: short, long, float ......
          } else {
                    throw new ClassNotFoundException(typeName);
          }
          return Array.newInstance(clazz, dim);
    }
}
```

```
class [I
class [[Z
class [[[Ljava.lang.Double;
```

Utilização de Plugins

IPlugin.java

```
public interface IPlugin {
    public void metodo();
}
```

Plugin1.java

```
public class Plugin1 implements IPlugin {
    public void metodo() {
          System.out.println("Plugin1: metodo invocado");
    }
}
```

Plugin2.java

```
public class Plugin2 implements IPlugin {
    public void metodo() {
          System.out.println("Plugin2: metodo invocado");
    }
}
```

Plugin3.java

```
public class Plugin3 implements IPlugin {
    public void metodo() {
          System.out.println("Plugin3: metodo invocado");
    }
}
```

Utilização de Plugins

Plugin.java

```
package reflection;
import java.io.File;

abstract class PluginManager {
    public static IPlugin load(String name) throws Exception {
      Class<?> c = Class.forName(name);
      return (IPlugin) c.newInstance();
    }
}

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

      File proxyList = new File("reflection/plugins");
      for (String f: proxyList.list()) {
         try {
            IPlugin obj = PluginManager.load("reflection."+f.substring(0,f.lastIndexOf('.')));
            obj.metodo();
         }
         catch (Exception e) {
            e.printStackTrace();
         }
      }
    }
}
```

```
Plugin1: metodo invocado
Plugin2: metodo invocado
Plugin3: metodo invocado
```

Padrões: Fábrica sem reflection

```
class Viveiro {
   public static Arvore factory(String pedido) {
     if (pedido.equalsIgnoreCase("Figueira"))
      { return new Figueira(); }
     if (pedido.equalsIgnoreCase("Pessegueiro"))
      { return new Pessegueiro(); }
     if (pedido.equalsIgnoreCase("Nespereira"))
      { return new Nespereira(); }
     else
      throw new IllegalArgumentException(”Árvore não
     existente!");
}
```

Padrões: Fábrica com reflection

```
class Viveiro {
   public static Arvore factory(String pedido){
     Arvore arv = null;
     try {
        arv =
   (Arvore) Class.forName("patterns."+pedido).newInstance();
     }
     catch(Exception e) {
        throw new IllegalArgumentException(Árvore nao
   existente!");
     }
        return arv;
   }
}
```

java.lang.reflect.Proxy

- Proxy provides static methods for creating dynamic proxy classes and instances, and it is also the superclass of all dynamic proxy classes created by those methods.
- To create a proxy for some interface Foo:

```
InvocationHandler handler = new MyInvocationHandler(...);
Class proxyClass = Proxy.getProxyClass(
    Foo.class.getClassLoader(), new Class[] { Foo.class });
Foo f = (Foo) proxyClass.
    getConstructor(new Class[] { InvocationHandler.class }).
    newInstance(new Object[] { handler });
```

- or more simply:

```
Foo f = (Foo) Proxy.newProxyInstance(Foo.class.getClassLoader(),
                                     new Class[] { Foo.class },
                                     handler);
```

Criação de proxies

- Podemos criar dinamicamente um proxy usando o método Proxy.newProxyInstance(). Este métodos aceita 3 parâmetros:
  1. O ClassLoader que "carrega" dinamicamente a class proxy
  2. Um vector das interfaces implementadas
  3. Um InvocationHandler para reencaminhar as chamadas aos métodos
- Exemplo:

```
InvocationHandler handler = new MyInvocationHandler();
MyInterface proxy = (MyInterface) Proxy.newProxyInstance(
                                MyInterface.class.getClassLoader(),
                                new Class[] { MyInterface.class },
                                handler);
```

  - A variável proxy passa a referenciar uma implementação dinâmica da interface MyInterface.
  - Todas as invocações ao proxy serão passadas à implementação do handler (do tipo InvocationHandler)

Proxy - utilização

- Database Connection and Transaction Management
- Dynamic Mock Objects for Unit Testing
- Adaptation of DI Container to Custom Factory Interfaces
- AOP-like Method Interception
- ..

Dynamic Proxy Classes

```
package reflection;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

interface MyInterface {
    void method();
}

class MyInterfaceImpl implements MyInterface {
    public void method() {
          System.out.println("method");
    }
}

class MyInterfaceImpl2 implements MyInterface {
    public void method() {
          System.out.println(”outro método");
    }
}
```

Dynamic Proxy Classes

```
class ProxyClass implements InvocationHandler {
    Object obj;

    public ProxyClass(Object o) {
        obj = o;
    }

    public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {
        Object result = null;
        try {
                System.out.println("before the method is called ");
                result = m.invoke(obj, args);
        } catch (Exception eBj) {
        } finally {
                System.out.println("after the method is called");
        }
        return result;
    }
}
```

Dynamic Proxy Classes

```
public class ProxySample {
    public static void main(String[] argv) throws Exception {

        MyInterface myintf =
        (MyInterface) Proxy.newProxyInstance(MyInterface.class.getClassLoader(),
        new Class[] { MyInterface.class },
        new ProxyClass(new MyInterfaceImpl()));
        myintf.method();

        myintf =
        (MyInterface) Proxy.newProxyInstance(MyInterface.class.getClassLoader(),
        new Class[] { MyInterface.class },
        new ProxyClass(new MyInterfaceImpl2()));
        myintf.method();
    }
}
```

```
before the method is called
method
after the method is called
before the method is called
outro método
after the method is called
```