

Java Annotation

Annotation

- Compiler instructions
- Build-time instructions
- Runtime instructions

At build-time:

- generating source code, (see e.g. <http://projectlombok.org/>)
- compiling the source,
- generating XML files (e.g. deployment descriptors),
- packaging the compiled code and files into a JAR file etc.

Build tools may scan your Java code for specific annotations and generate source code or other files based on these annotations.

Example: lombok

```
import lombok.AccessLevel;
import lombok.Setter;
import lombok.Data;
import lombok.ToString;

public @Data class Mountain{
    private final String name;
    private double altitude, longitude;
    private String country;
}
```

autogenerates:

- setters
- getters
- hashCode()
- equals(Object)
- toString()

You have to include lombok.jar as library – see . <http://projectlombok.org/>

Predefined Annotation

- `@Deprecated`
- `@Override`
- `@SuppressWarnings`

Creating your own annotation

```
@interface MyAnnotation {  
    String    value() default "123";  
    String    surname();  
    int       age();  
    String[]  names();  
}
```

```
@MyAnnotation(  
    surname="Depippis",  
    age=33,  
    names={"Pluto", "Goofie"}  
)  
public class MyClass {  
    ...  
}
```

Default name for single argument is "value"

Annotation lifetime

```
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;
```

```
@Retention(RetentionPolicy.RUNTIME)  
@interface MyAnnotation {  
    String    value() default "123";  
    String    surname();  
    int       age();  
    String[]  names();  
}
```

CLASS

recorded in the class file by the compiler but need not be retained by the VM at run time.

RUNTIME

recorded in the class file by the compiler and retained by the VM at run time, so they may be read reflectively.

SOURCE

discarded by the compiler.

Annotation scope

```
import java.lang.annotation.ElementType;  
import java.lang.annotation.Target;
```

```
@Target({ElementType.METHOD})
```

```
@interface MyAnnotation {  
    String    value() default "123";  
    String    surname();  
    int       age();  
    String[]  names();  
}
```

The `@Inherited` annotation signals that a custom annotation used in a class should be inherited by subclasses inheriting from that class.

- **ANNOTATION_TYPE**
 - Annotation type declaration
- **CONSTRUCTOR**
 - Constructor declaration
- **FIELD**
 - Field declaration (includes enum constants)
- **LOCAL_VARIABLE**
 - Local variable declaration
- **METHOD**
 - Method declaration
- **PACKAGE**
 - Package declaration
- **PARAMETER**
 - Parameter declaration
- **TYPE**
 - Class, interface (including annotation type), or enum declaration

Java Reflection

discovering the code...

The Class class

represents a class.

```
Class c=Class.forName("javax.swing.JFrame");
```

```
Class c=JFrame.class;
```

```
Class c=int.class;
```

What can we do with it?

```
String mysteryName=JOptionPane.showInputDialog(  
    "Give me the qualified name of a class",null);
```

```
Class mystery=Class.forName("javax.swing.JFrame");
```

```
Object o=mystery.getInstance();
```

Methods of the Class class

- `String getName()`,
- `getCanonicalName()`,
- `getSimpleName()`

the **name** is the name that you'd use to dynamically load the class with, e.g. in `Class.forName`

the **canonical name** is the name that'd be used in an import statement and uniquely identifies the class.

the **simple name** loosely identifies the class

name and canonical name are different for inner classes.

```
java.lang.String  
java.lang.String  
String
```

```
java.util.AbstractMap$SimpleEntry  
java.util.AbstractMap.SimpleEntry  
SimpleEntry
```

see

<http://stackoverflow.com/questions/15202997/what-is-the-difference-between-canonical-name-simple-name-and-class-name-in-jav>

Methods of the Class class

- boolean isAnnotation(), isArray(), isAnonymousClass(), isPrimitive()...
- boolean isInstance(Object a)
- toString()

Methods of the Class class

- **Method** `getMethod(String s, Class[] ptypes);` - also `getDeclaredMethod`
- `Method[] getMethods();` - also `getDeclaredMethods`
- **Constructor** `getConstructor(Class[] ptypes);` - `getConstructors()`
- **Field** `getField(String s);` – `getFields();`
- `Class getClass(String s);` – `getClasses()` //inner classes
- **Annotation** `getAnnotation(Class a);` – `getAnnotations()`
- `Class[] getInterfaces()`
- `Class getSuperclass()`
- **Package** `getPackage()`

note: the "Declared" version is there also for `Constructor(s)`, `Field(s)`, `Annotation(s)`

Package, Method, Constructor, Field, Annotation are in the `java.reflect` package

Class: checking for inheritance

instanceof works on instances, i.e. on Objects. Sometimes you want to work directly with classes. In this case you can use the `subClass` method of the Class class.

```
Class o=Object.class;
```

```
Class c=Class.forName("javax.swing.JFrame").asSubclass(o);
```

this will go through smoothly because JFrame is subclass of Object.

c will contain a Class object representing the JFrame class.

```
Class o=JButton.class;
```

```
Class c=Class.forName("javax.swing.JFrame").asSubclass(o);
```

this will launch a java.lang.ClassCastException because JFrame is NOT subclass of JButton.

c will not be initialized.

```
Class o=Serializable.class;
```

```
Class c=Class.forName("javax.swing.JFrame").asSubclass(o);
```

this will go through smoothly because JFrame implements the java.io.Serializable interface.

c will contain a Class object representing the JFrame class.

The Field Class

extends `AccessibleObject`, implements `AnnotatedElement`

- `getName()`
- `int getModifiers()`
- accessory methods
 - `getInt/setInt`, `getBoolean/setBoolean`...
- `isAccessible/setAccessible`

Modifier class

- static int ABSTRACT
- static int FINAL
- static int INTERFACE
- static int NATIVE
- static int PRIVATE
- static int PROTECTED
- static int PUBLIC
- static int STATIC
- static int STRICT
- static int SYNCHRONIZED
- static int TRANSIENT
- static int VOLATILE

The sets of modifiers are represented as integers with distinct bit positions representing different modifiers

static boolean methods `isStatic()`, `isPublic()`...

The Method Class

- `getName()`
- `Class[] getParameterTypes()`
- `int getModifiers()`
- `Class getReturnType()`
- `Class getExceptionTypes()`
- `Class getDeclaringClass()`
- `Annotation getAnnotation(Class annotation)`
- `Annotation[] getDeclaredAnnotations()`
- `invoke(Object, Object...)`

Method invocation from reflection

```
Student s=new Student();  
s.enrol(4,"Course");
```

```
Class c=Class.forName("reflect.Student");  
Object o=c.newInstance();  
Methods[] ms=c.getMethods();  
// show the user the available methods, and ask him to  
select one  
Method m=c.getMethod("enrol",int.class,  
    Class.forName("java.lang.String"));  
m.invoke(o,3,"Course");
```

Costructors

Class **Constructor**

and methods of the Class class

- **getConstructor(Class[] parameterTypes)**
- **getConstructors()**
- **getDeclaredConstructors()**
- **getDeclaredConstructor(Class[] parameterTypes)**

work as for methods and has a subset of similar methods

Java Reflection

an example

Our annotations

```
package reflect;
import ..;
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface ClassNote {
    String value();
}
```

```
package reflect;
import ..;
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface FieldNote {
    String value();
}
```

```
package reflect;
import ..;
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface MethodNote {
    String value();
    int par2();
}
```

Our classes

```
package reflect;
import ..;
public class Person {
    protected String name;
    public int age;
    @FieldNote("comment")
    public static String company;
    @MethodNote(value="getter",par2=1)
    public String getName() {
        return name;
    }
}
```

```
package reflect;
import ..;
@ClassNote("demo")
public class Student extends Person{
    @FieldNote("comment")
    private int matricola;

    @MethodNote(value="someText",par2=27)
    public int enrol(int course_id,String s) {
        return course_id*10+1;
    }
}
```

The inspector

```
package reflectInspector;
import ..;
public class Reflect {
    void p(String s) { System.out.println(s); }
    public static void main(String[] args) {
        String mysteryName=JOptionPane.showInputDialog(
            "Give me the qualified name of a class", null); //"reflect.Student"
        Class mystery = null;
        try {
            mystery = Class.forName(mysteryName); // get the class
            //Object misteriousObj=mystery.newInstance(); // this would instantiate the class
        } catch (ClassNotFoundException ex) {
            ex.printStackTrace();    System.exit(1);
        }
        new Reflect().discover(mystery);
    }
}
```

Discover info about the class:

Class methods

```
void discover(Class mystery) {  
    p("Class name: " + mystery.getSimpleName() +  
      " in package: " + mystery.getPackage().getName());  
    p("inherits from: " +  
      mystery.getSuperclass().getName());  
    if (Modifier.isPublic(mystery.getModifiers())) {  
        p("The class is public");  
    }  
    getAnnotationInfo(mystery,  
        mystery.getSimpleName());  
    p("=====");  
}
```

...

Output:

Class name: Student in package: reflect
inherits from: reflect.Person

The class is public

==> Annotations:

annotation reflect.ClassNote for Class Student

-> name: value value: demo

Instance variables: Field

```
p("==> Instance variables:");
Field[] vars1 = mystery.getFields(); // get public fields
// get non public fields
Field[] vars2 = mystery.getDeclaredFields();
ArrayList<Field> vars=
    new ArrayList<>(Arrays.asList(vars2));
vars.addAll(Arrays.asList(vars1));
for (Field var : vars) {
    p(var.getType().getName() + " " + var.getName());
    if (Modifier.isPublic(var.getModifiers())) {
        p("the field is public");
    } else var.setAccessible(true);
    if (Modifier.isStatic(var.getModifiers())) {
        p("the field is static");
    }
    getAnnotationInfo(var, var.getName());
    p("-----");
}
p("=====");
```

OUTPUT

==> Instance variables:

int matricola

==> Annotations:

annotation reflect.FieldNote
for Field matricola

-> name: value value: comment

int age

the field is public

java.lang.String company

the field is public

the field is static

==> Annotations:

annotation reflect.FieldNote
for Field company

-> name: value value: comment

Methods

```
p("===> Methods:");
Method[] ms =
    mystery.getMethods();
    for (Method m : ms) {
        Class retType =
            m.getReturnType();
        Class[] parTypes =
            m.getParameterTypes();
        StringBuilder b =
            new StringBuilder("");
        if (Modifier.isPublic
            (m.getModifiers())) {
            b.append("public ");
        }
        b.append(retType.getName());
        b.append(" ");
        b.append(m.getName());
        b.append("(");
        boolean first = true;
        for (Class t : parTypes) {
            if (!first) {
                b.append(",");
            }
            b.append(
                t.getName());
            first = false;
        }
        b.append(")");
        p(b.toString());
        getAnnotationInfo(m,
            m.getName());
        p("-----");
    }
}
```

===> Methods:

```
public int enrol(int,java.lang.String)
```

===> Annotations:

```
annotation reflect.MethodNote
    for Method enrol
```

```
-> name: value value: someText
```

```
-> name: par2 value: 27
```

```
public java.lang.String getName()
```

===> Annotations:

```
annotation reflect.MethodNote
    for Method getName
```

```
-> name: value value: getter
```

```
-> name: par2 value: 1
```

```
public void setName(java.lang.String)
```

```
public void wait(long,int)
```

```
public void wait(long)
```

```
public void wait()
```

```
public boolean equals(java.lang.Object)
```

```
public java.lang.String toString()
```

```
public int hashCode()
```

```
public java.lang.Class getClass()
```

```
public void notify()
```

```
public void notifyAll()
```

How did we get the annotations?

==> Annotations:

annotation reflect.ClassNote for **Class** Student

-> name: value value: demo

...

==> Annotations:

annotation reflect.FieldNote for **Field** matricola

-> name: value value: comment

...

==> Annotations:

annotation reflect.MethodNote for **Method** getName

-> name: value value: getter

-> name: par2 value: 1

What do **Class**, **Field** and **Method** have in common?

they all implement **AnnotatedElement**

Exploring the annotations

```
void getAnnotationInfo(AnnotatedElement o, String annotatedElementName) {  
    Annotation[] notes = o.getAnnotations();  
    if (notes.length != 0) {  
        p("==> Annotations:");  
        for (Annotation note : notes) {  
            p("annotation " + note.annotationType().getName() + " for " +  
              o.getClass().getSimpleName() + " " + annotatedElementName);  
            Method[] ms = note.annotationType().getMethods();  
            for (Method m : ms) {  
                String methodName = m.getName();  
                switch (methodName) {  
                    case "toString":  
                    case "hashCode":  
                    case "annotationType":  
                        continue;  
                }  
            }  
        }  
    }  
}
```

==> Annotations:
annotation reflect.ClassNote for Class Student
-> name: value value: demo

==> Annotations:
annotation reflect.MethodNote for Method getName
-> name: value value: getter
-> name: par2 value: 1

Exploring the annotations

```
try {  
    p("-> name: " + methodName + " value: " + (m.invoke(note)));  
} catch (    SecurityException |  
            IllegalAccessException |  
            IllegalArgumentException |  
            InvocationTargetException ex) {  
    //ex.printStackTrace();  
}  
}  
p("-----");  
}  
}
```

==> Annotations:

annotation reflect.MethodNote for Method getName

-> name: value value: getter

-> name: par2 value: 1