Reflection and Annotations

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.org

Have a Question?

Software University

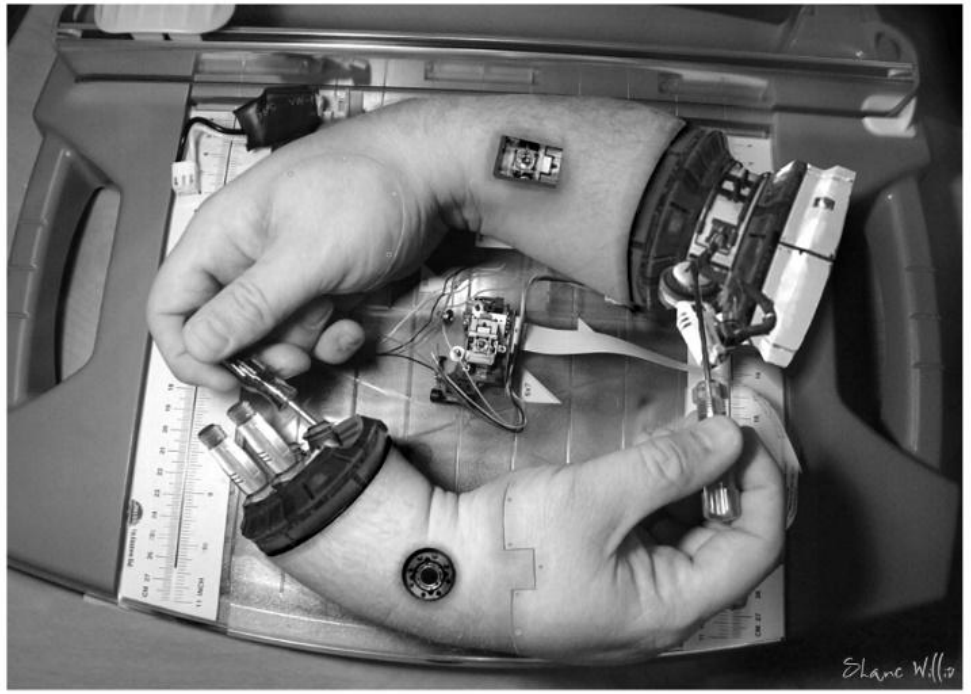
Channel for Communication

Table of Contents

Reflection

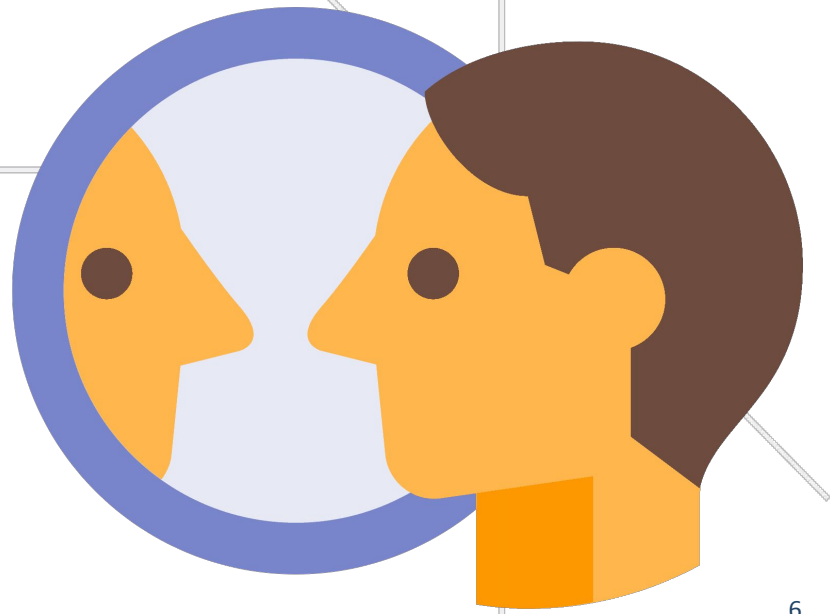
What is Metaprogramming?

- **Programming technique** in which computer programs have the ability to treat **programs as their data**
- The program can be designed to:
  - **Read**
  - **Generate**
  - **Analyze**
  - **Transform**
- **Modify itself** while **running**



What is Reflection?

- The ability of a programming language to be its **own metalanguage**
- Programs can examine information about **themselves**

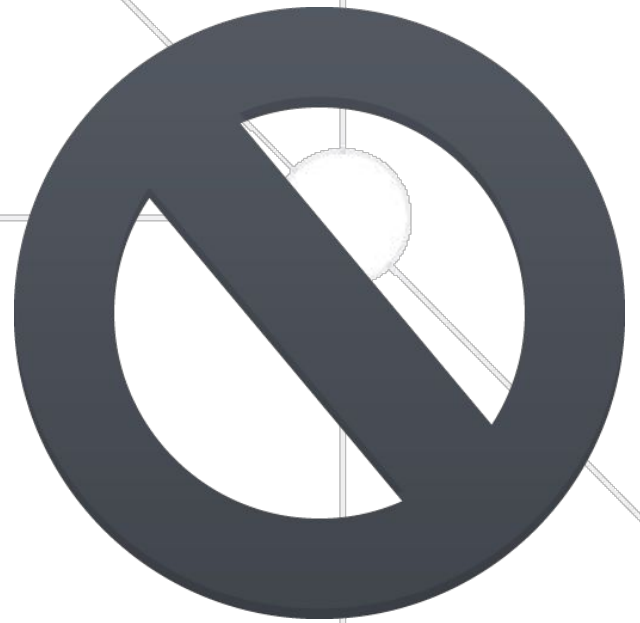


When to Use Reflection?

- Whenever we want:
  - Code to become more **extendible**
  - To **reduce code** length significantly
  - Easier **maintenance**
  - Easier **testing**

When Not to Use Reflection?

- If it is **possible** to **perform** an operation **without** using **reflection**, then it is preferable to **avoid using it**
- Cons from using Reflection
  - **Performance** overhead
  - **Security** restrictions
  - Exposure of **internal logic**



Reflection API

The Class Object

- Obtain it' `java.lang.Class` object
  - If you **know** the **name**

```
Class myObjectClass = MyObject.class;
```

  - If you **don't** know the name at **compile time**

```
Class myClass = Class.forName(className);
```

You need fully qualified class name as String

Class Name

- Obtain `Class` name
  - Fully qualified class name

```
String className = aClass.getName();
```

  - Class name without the package name

```
String simpleClassName = aClass.getSimpleName();
```

Base Class and Interfaces

Software University

- Obtain **parent class**

```
Class className = aClass.getSuperclass();
```

- Obtain **interfaces**

```
Class[] interfaces = aClass.getInterfaces();
```

  - **Interfaces** are also **represented** by `Class` objects in Java Reflection
  - Only the interfaces **specifically declared** implemented by a given class are **returned**

Problem: Reflection

- Import ReflectionClass to your **src** folder in your project
- Using **reflection** you should print:
  - This class type
  - Super class type
  - All Interfaces
  - Instantiate object using reflection and print it
- Don't change anything in class

Check your solution here: https://judge.softuni.bg/Contests/1604/Reflection-Lab

Solution: Reflection

```
Class<Reflection> aClass = Reflection.class;
System.out.println(aClass);
System.out.println(aClass.getSuperclass());
Class[] interfaces = aClass.getInterfaces();
for (Class anInterface : interfaces)
  System.out.println(anInterface);
//Reflection ref = aClass.newInstance();//Deprecated since Java 9
Reflection ref = aClass.getDeclaredConstructor().newInstance();
System.out.println(ref);
```

Create new object

Constructors, Fields and Methods

Constructors (1)

- Obtain **only public constructors**

```
Constructor[] ctors = aClass.getConstructors();
```

- Obtain **all constructors**

```
Constructor[] ctors =
              aClass.getDeclaredConstructors();
```

- Get constructor by **parameters**

```
Constructor ctor =
            aClass.getConstructor(String.class);
```

Constructors (2)

- Get **parameter types**

```
Class[] parameterTypes =
            ctor.getParameterTypes();
```

- **Instantiating objects** using constructor

```
Constructor constructor =
MyObject.class.getConstructor(String.class);
MyObject myObject = (MyObject)constructor
                .newInstance("arg1");
```

Fields Name and Type

- Obtain **public** fields

```
Field field = aClass.getField("somefield");
Field[] fields = aClass.getFields();
```

- Obtain **all** fields

```
Field[] fields = aClass.getDeclaredFields();
```

- Get field **name and type**

```
String fieldName = field.getName();
Object fieldType = field.getType();
```

Fields Set and Get

Software University

- Setting value for a field

```
Class aClass = MyObject.class;
Field field = aClass.getDeclaredField("someField");
MyObject objectInstance = new MyObject();
field.setAccessible(true);
Object value = field.get(objectInstance);

field.set(objectInstance, value);
```

Change the behavior of the AccessibleObject

The objectInstance parameter passed to the get and set method should be an instance of the class that owns the field

Methods

- Obtain **public** methods

```
Method[] methods = aClass.getMethods();
Method method =
    aClass.getMethod("doSomething",String.class);
```

- Get methods without **parameters**

```
Method method =
          aClass.getMethod("doSomething", null);
```

Method Invoke

- Obtain method **parameters** and **return type**

```
Class[] paramTypes = method.getParameterTypes();
Class returnType = method.getReturnType();
```

- Get methods with **parameters**

```
Method method = MyObject.class
        .getMethod("doSomething", String.class);
Object returnValue = method.invoke(null, "arg1");
```

null is for static methods

Problem: Getters and Setters

- Using **reflection** get all methods and print:
- **Sort** getters and setters **alphabetically**
- **Getters**:
  - A getter method have its name start with "get", take 0 parameters, and returns a value
- **Setters**:
  - A setter method have its name start with "set", and takes 1 parameter

Check your solution here: https://judge.softuni.bg/Contests/1604/Reflection-Lab

Solution: Getters

```
Method[] methods = Reflection.class.getDeclaredMethods();
Method[] getters = Arrays.stream(methods)
        .filter(m -> m.getName().startsWith("get") &&
                            m.getParameterCount() == 0)
        .sorted(Comparator.comparing(Method::getName))
        .toArray(Method[]::new);
Arrays.stream(getters).forEach(m ->
        System.out.printf("%s will return class %s%n",
            m.getName(), m.getReturnType().getName()));
```

Access Modifiers

Access Modifiers

- Obtain the **class modifiers** like this

```
int modifiers = aClass.getModifiers();
```

- Each modifier is a **flag bit** that is either set or cleared
- You can check the modifiers

```
Modifier.isPrivate(modifiers);
Modifier.isProtected(modifiers);
Modifier.isPublic(modifiers);
Modifier.isStatic(modifiers);
```

`getModifiers()` can be called on constructors, fields, methods

Arrays

Software University

- Creating arrays via Java Reflection

```
int[] intArray = (int[]) Array.newInstance(int.class, 3);
```

- Obtain parameter annotations

```
Array.set(intArray, 0, 123);
Array.set(intArray, 1, 456);
```

- Obtain fields and methods annotations

```
Class stringArrayComponentType =
                        stringArrayClass.getComponentType();
```

Problem: High Quality Mistakes

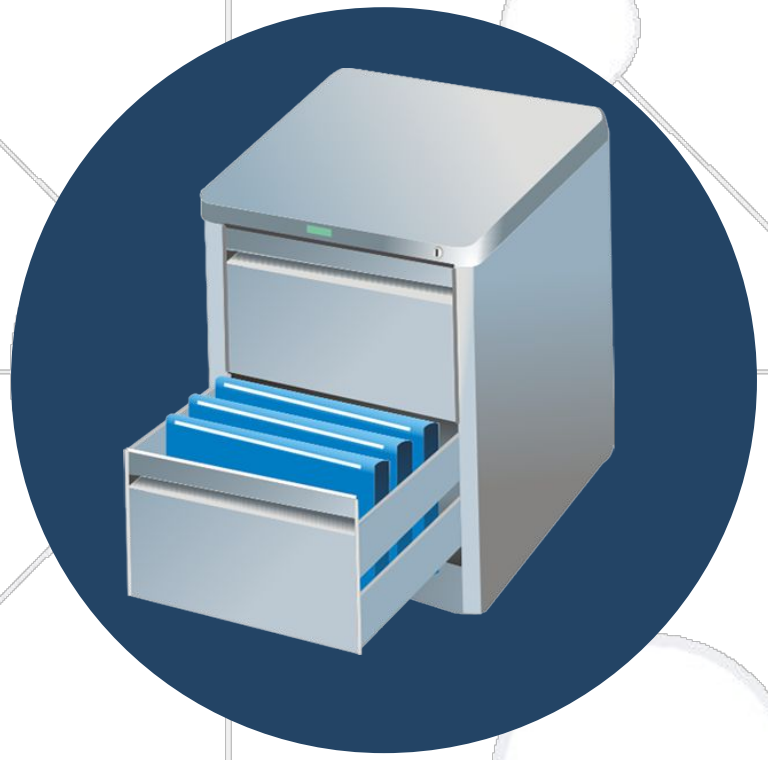
Software University

- You perfectly know how to write High Quality Code
- Check **Reflection class** and print all mistakes in **access modifiers** which you can find
- Get all fields, getters and setters and sort each category by name
- First print mistakes in **fields**
- Then print mistakes in **getters**
- Then print mistakes in **setters**

Check your solution here: https://judge.softuni.bg/Contests/1604/Reflection-Lab

Solution: High Quality Mistakes

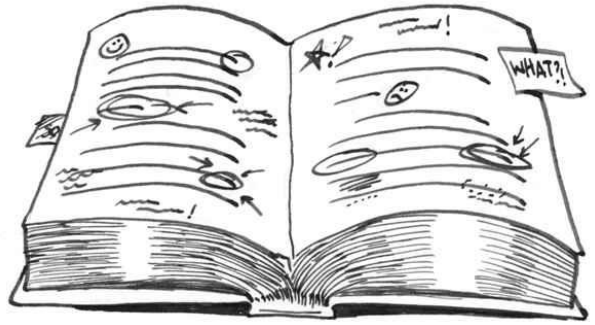
```
Field[] fields = Reflection.class.getDeclaredFields();
Arrays.stream(fields)
      .filter(f -> !Modifier.isPrivate(f.getModifiers()))
      .sorted((Comparator.comparing(Field::getName)))
      .forEach(f -> System.out
        .printf("%s must be private!%n", f.getName()));
// TODO: Do the same for getters and setters
```



Annotations

Annotation

- **Data holding** class
- **Describe** parts of your code
- Applied to: **Classes**, **Fields**, **Methods**, etc.




```
@Deprecated
public void deprecatedMethod() {
  System.out.println("Deprecated!");
}
```

Annotation Usage

- To generate **compiler messages** or **errors**

```
@SuppressWarnings("unchecked")
@Deprecated
```

- As tools
  - **Code generation** tools
  - **Documentation generation** tools
  - **Testing** Frameworks
- At runtime – **ORM**, **Serialization**, etc.

Built-In Annotations (1)

- **@Override** – generates **compile time error** if the method does not override a method in a parent class

```
@Override
public String toString() {
   return "new toString() method";
}
```

Built-In Annotations (2)

- `@SupressWarning` – turns off **compiler warnings**

```
@SuppressWarnings(value = "unchecked")
public <T> void warning(int size) {
  T[] unchecked = (T[]) new Object[size];
}
```

Annotation with value

Generates compiler warning

Built-In Annotations (3)

- **@Deprecated** – generates a **compiler warning** if the element is used

Generates compiler warning

```
@Deprecated
public void deprecatedMethod() {
  System.out.println("Deprecated!");
}
```

Creating Annotations

- `@interface` – the keyword for annotations

```
public @interface MyAnnotation {
   String myValue() default "default";
}
```

Annotation element

```
@MyAnnotation(myValue = "value")
public void annotatedMethod() {
   System.out.println("I am annotated");
}
```

Skip name if you have only one value named "value"

Annotation Elements

- Allowed types for annotation elements:
  - Primitive types (`int`, `long`, `boolean`, etc.)
  - `String`
  - `Class`
  - `Enum`
  - `Annotation`
  - `Arrays` of any of the above

Meta Annotations – @Target

- **Meta annotations** annotate annotations
- **@Target** – specifies where the annotation is applicable

```
@Target(ElementType.FIELD)
public @interface FieldAnnotation {
}
```

Used to annotate fields only

- Available element types – **CONSTRUCTOR**, **FIELD**, **LOCAL_VARIABLE**, **METHOD**, **PACKAGE**, **PARAMETER**, **TYPE**

Meta Annotations – @Retention

- `@Retention` – specifies where the annotation is available

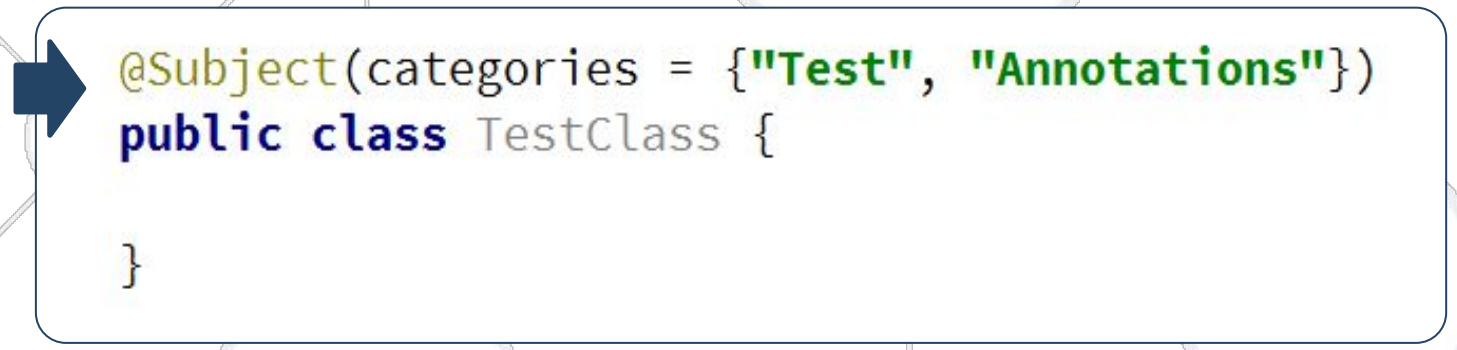
```
@Retention(RetentionPolicy.RUNTIME)
public @interface RuntimeAnnotation {
    // …
}
```

You can get info at runtime

- Available retention policies – `SOURCE`, `CLASS`, `RUNTIME`

Problem: Create Annotation

- Create annotation **Subject** with a String[] **element** "categories"
  - Should be **available at runtime**
  - Can be **placed** only **on types**



```
@Subject(categories = {"Test", "Annotations"})
public class TestClass {

}
```

Solution: Create Annotation

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Subject {
   String[] categories();
}
```

Annotations

- Obtain class annotations

```
Annotation[] annotations = aClass.getAnnotations();
Annotation annotation = aClass.getAnnotation(MyAnno.class);
```

- Obtain parameter annotations

```
Annotation[][] parameterAnnotations =
                                method.getParameterAnnotations();
```

- Obtain fields and methods annotations

```
Annotation[] fieldAnots = field.getDeclaredAnnotations();
Annotation[] methodAnot = method.getDeclaredAnnotations();
```

Accessing Annotation (1)

- Some annotations can be accessed **at runtime**

```
@Author(name = "Gosho")
public class AuthoredClass {
   public static void main(String[] args) {
      Class cl = AuthoredClass.class;
      Author author = (Author) cl.getAnnotation(Author.class);
      System.out.println(author.name());
   }
}
```

Accessing Annotation (2)

- Some annotations can be accessed **at runtime**

```
Class cl = AuthoredClass.class;
Annotation[] annotations = cl.getAnnotations();
for (Annotation annotation : annotations) {
  if (annotation.annotationType().equals(Author.class)) {
    Author author = (Author) annotation;
    System.out.println(author.name());
  }
}
```

Problem: Coding Tracker

- Create annotation **Author** with a String **element** "name"
  - Should be **available at runtime**
  - Should be **placed** only **on methods**
- Create a class **Tracker** with a method:
  - **public static void printMethodsByAuthor()**

```
@Author(name = "George")
public static void main(String[] args) {
    Tracker.printMethodsByAuthor(Tracker.class);
}

@Author(name = "Peter")
public static void printMethodsByAuthor(Class<?> cl) {...}
```

```
George: main()
Peter: printMethodsByAuthor()
```

Solution: Coding Tracker (1)

Software University

```
public class Tracker {
  public static void printMethodsByAuthor(Class<?> cl) {
    Map<String, List<String>> methodsByAuthor = new HashMap<>();
    Method[] methods = cl.getDeclaredMethods();

    for (Method method : methods) {
      Author annotation = method.getAnnotation(Author.class);
    // Continues on next slide
```

Solution: Coding Tracker (2)

Software University

```
        if (annotation != null) {
          methodsByAuthor
         .putIfAbsent(annotation.name(), new ArrayList<>());
          methodsByAuthor
         .get(annotation.name()).add(method.getName() + "()");
        }
      }
      // TODO: print the results
    }
}
```

Summary

- What is **Reflection**
- Reflection **API**
  - Reflecting Classes, Constructors, Fields, Methods
  - Access Modifiers
- **Annotations**
  - Used to describe our code
  - Provide the possibility to work with non-existing classes
  - Can be accessed through **reflection**

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

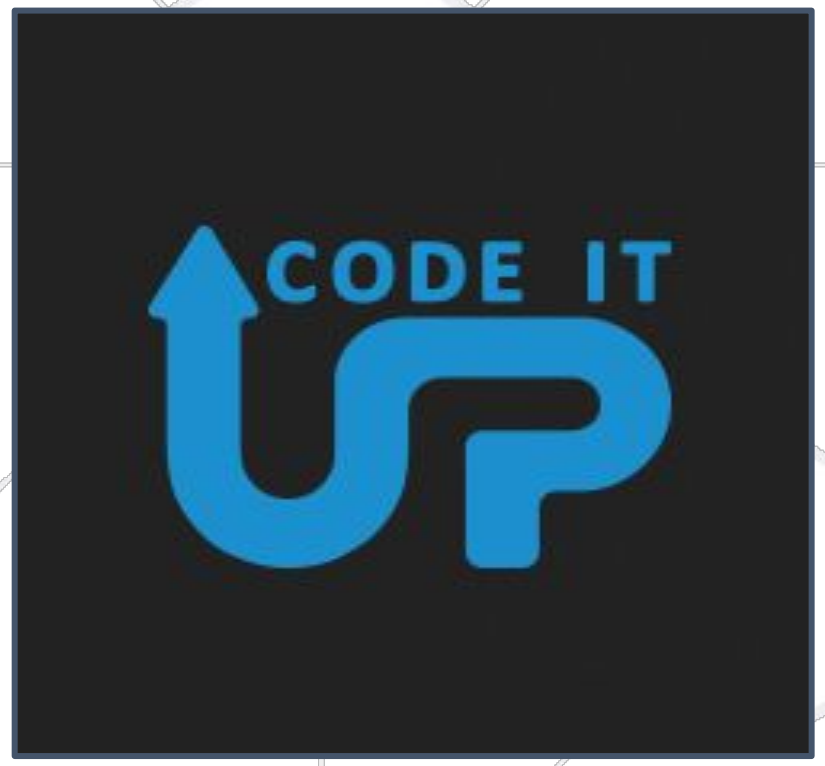
SoftUni Kids

SoftUni Diamond Partners

Software University

Educational Partners

Software University



Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- SoftUni Global
  - softuni.org

Software University

SoftUni Global

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg
- © SoftUni Global – https://softuni.org