Reflection basics

Reflection is a powerful feature in Java that allows a programmer to examine or modify the structure of a class at runtime.

That means a program can inspect and manipulate its own code, making reflection a useful tool for runtime code generation, testing, and more.

Most importantly, you can **bypass encapsulation** by accessing member fields that are not exposed by its public API. It allows getting implementation details.

java.lang.reflect package

Java Reflection is implemented by the java.lang.reflect package.

java.lang.reflect package includes many interfaces, classes, and exceptions, there are only four classes that you need to know at this level. These classes are:

- **Field**: you can use it to get and modify name, value, datatype and access modifier of a variable.
- **Method**: you can use it to get and modify name, return type, parameter types, access modifier, and exception type of a method.
- **Constructor**: you can use it to get and modify name, parameter types and access modifier of a constructor.
- **Modifier**: you can use it to get information about a particular access modifier.

java.lang.Class

You can't just achieve reflection only with the Reflect package that we've mentioned above. The Reflect package can give you information about a field, method or constructor of a class, but first you have to take the field list, method list, and constructor list.

This is possible with the java.lang.Class class and its static forName() method. When you pass the name of any class to the forName() method, it returns a Class object that includes information about this class.

Another method which returns Class object is getSuperclass() instance method. This method returns a superclass of a Class instance.

The java.lang.Class also has several methods that you can use to get **attributes** (fields, methods, constructors) of the particular class you passed to forName() method.
Here are some of those methods:

- **getConstructors()**
- **getDeclaredConstructors()**

- **getFields()**
- **getDeclaredFields()**
- **getMethods()**
- **getDeclaredMethods()**

First, each of these methods returns an array of objects from java.lang.reflect classes. For example, getFields() returns an array of objects from the java.lang.reflect.Field class. After that, you can use methods from java.lang.reflect package to get further information about constructors, fields, and methods.

Second, getConstructors(), getFields() and getMethods() return only public constructors, fields and methods from the class represented by the Class object. These methods also return **inherited** public fields and methods from **superclasses**.

Similarly,getDeclaredConstructors(), getDeclaredFields(), getDeclaredMethods() return all the constructors, fields and methods from the class represented by the Class object. These methods don't return inherited fields and methods from superclasses.

Usually, you can see developers use declared methods more often than non-declared methods.

Coding examples

```
public class Student {
    public String firstName;
    public String lastName;
    public int age;
    protected String phoneNumber;
    private String accountNumber;

    Student(){
        System.out.println("This is default Constructor");
    }

    public Student(String firstName, String lastName){
        this.firstName= firstName;
        this.lastName= lastName;
        System.out.println("This is public Constructor");
    }

    private String sanitizeAccountNumber(String accountNumber){
        System.out.println("This is a private method to sanitize account number");
```

```
        //code to sanitize accountNumber goes here.
        return accountNumber;
    }

    public void setAccountNumber(String accountNumber){
        accountNumber = sanitizeAccountNumber(accountNumber);
        this.accountNumber = accountNumber;
    }
}
```

The reflection process usually has three steps:

1. Get a java.lang.Class object of the class using the forName() method. In this case, the class we want to reflect is Student.
   Class student = Class.forName("Student");

2. Get the class attributes. In this case, we are interested in superclass, fields, constructors, and methods.

- **Class superclass = student.getSuperclass();**
- **Constructor[] declaredConstructors = student.getDeclaredConstructors();**
- **Constructor[] constructors = student.getConstructors();**
- **Field[] declaredFields = student.getDeclaredFields();**
- **Field[] fields = student.getFields();**
- **Method[] declaredMethods = student.getDeclaredMethods();**
- **Method[] methods = student.getMethods();**

Get the information about class attributes and use it. In this case, we are going to retrieve the names of superclass, constructors, fields, and methods and print them.

```
System.out.println("Superclass " + superclass);

for (Constructor dc : declaredConstructors) {
    System.out.println("Declared Constructor " + dc.getName());
}
for (Constructor c : constructors) {
    System.out.println("Constructor " + c.getName());
}
for (Field df : declaredFields) {
    System.out.println("Declared Field " + df.getName());
}
for (Field f : fields) {
    System.out.println("Field " + f.getName());
}
for (Method dm : declaredMethods) {
    System.out.println("Declared Method " + dm.getName());
}
```

```
for (Method m : methods) {
   System.out.println("Method " + m.getName());
}
```

Explaining the output:

```
Superclass class java.lang.Object
Declared Constructor Student
Declared Constructor Student
Constructor Student
Declared Field firstName
Declared Field lastName
Declared Field age
Declared Field phoneNumber
Declared Field accountNumber
Field firstName
Field lastName
Field age
Declared Method sanitizeAccountNumber
Declared Method setAccountNumber
Method setAccountNumber
Method wait
Method wait
Method wait
Method equals
Method toString
Method hashCode
Method getClass
Method notify
Method notifyAll
```

You can see that getDeclaredConstructors() has returned both constructors of the Student class, while getConstructors() has returned only the public constructor. Likewise, getDeclaredFields() has returned all the fields of the Student class, while getFields() has returned only public fields.

Finally, we print the methods of the Student class. As expected, getDeclaredMethods() has returned both methods. Now the interesting part is that getMethods() has returned some methods other than setAccountNumber() we've expected. If you remember, in one of our previous topics, we mentioned that the java.lang.Object class is the superclass of all the classes we create. The Object class has **nine** public methods, and all classes we create inherit those methods.

Risks and downsides
While reflection offers powerful capabilities, it should be used with caution.

- Performance overhead: reflection can add significant overhead to your program, requiring extra processing to inspect and manipulate the code structure at runtime. This can lead to slower performance, especially in large and complex applications.
- Maintenance issues: reflection can complicate maintaining code, as the code's behavior can change dynamically at runtime. This can make it challenging to understand how the code is being used and manipulated, leading to compatibility issues and making it harder to update or change the code in the future.