CODEGYM

Java developer in 12 months

Mentor-supported training program from CodeGym

MODULE 2. JAVA CORE

Лекция #17 Reflection API

Lesson plan

- Reflection API
- Getting data using reflection
- Creating objects using reflection
- Changing the internal state of an object using reflection
- Dynamic Proxy

Reflection API

Reflection in Java is a mechanism that allows a developer to make changes and obtain information about classes, interfaces, fields, and methods at runtime without knowing their names.

The Reflection API also helps you create new class instances, call methods, and get or set field values.

Reflection is used in almost all modern Java technologies and underlies most modern Java / Java EE frameworks and libraries, for example, in:

- **Spring** - frameworks for building web applications
- **JUnit** - testing framework

Possibilities of using reflection

- Find out/determine the class of an object.
- Get information about class modifiers, fields, methods, constants, constructors and superclasses.
- Find out which methods belong to the implemented interface/interfaces.
- Create an instance of a class when the class name is not known until the time when the program is executed.
- Get and set the value of an object field by name.
- Call an object's method by name.

Reflection disadvantages

- Application security breaches. With reflection, we can access a piece of code that we shouldn't (violation of encapsulation).
- Security restrictions. Reflection requires runtime permissions that are not available to systems running a security manager.
- Low performance. Reflection in Java determines types dynamically by scanning the classpath to find the class to load. This reduces the performance of the program.
- Support difficulties. Code written with reflection is difficult to read and debug. It becomes less flexible and harder to maintain.

CODEGYM

Getting data using reflection

Main methods of the java.lang.reflect.Field class:

getType() – returns a class object that identifies the declared type for the field represented by this Field object.

getAnnotatedType() - returns an annonatedType object representing the use of the type to specify the declared type of the field.

getGenericType() - returns a Type object representing the declared type for the field represented by this Field object.

getName() – returns the name of the field represented by the Field object.

getModifiers() - Returns the modifiers for the field represented by this Field object, in a form of an integer. The Modifier class should be used to decode modifiers.

getAnnotation(Class<T> annotationClass) - returns the annotation of the element for the specified type, if such an annotation is present, otherwise null.

getAnnotationsByType(Class<T> annotationClass) - returns the annotations associated with the element. If no annotations are associated with this element, the return value is a zero-length array.

CODEGYM

Main methods of the java.lang.reflect.Method class:

getName() - returns the name of the method represented by the Method object as a string.

getModifiers() - returns the modifiers of the method represented by the Method object.

getReturnType() - returns a Class object representing the formal return type of the method represented by the Method object.

getGenericReturnType() - returns a Type object representing the formal return type of the method represented by the Method object.

getParameterTypes() - returns an array of Class objects that represent the types of formal parameters in the declaration of the method, represented by the Method object. Returns a zero-length array if the method takes no parameters.

getGenericParameterTypes() - returns an array of Type objects that represent the types of formal parameters in the method declaration, represented by the Method object.

getExceptionTypes() - returns an array of Class objects representing the types of exceptions declared to be thrown by the method, represented by the Method object. Returns a zero-length array if the method does not declare exceptions in its throws.

getGenericExceptionTypes() - returns an array of Type objects representing the exceptions declared to be thrown by the Method object.

getAnnotations() - returns the annotations present in this method. If there are no annotations on the method, the return value is a zero-length array.

getDeclaredAnnotations() - returns the annotations directly present in this method.

CODEGYM

Creating objects using reflection

In Java, we usually create objects using the new keyword. But in the Reflection API, there are two methods that we can use to create objects:

1. **Class.newInstance() from the java.lang package**
2. **Constructor.newInstance() from the java.lang.reflect package**

In order to use Class.newInstance(), we first need to get an instance of the class we want to create objects for. Example:

```
Employee employee = Employee.class.newInstance();
```

To use the Constructor.newInstance() method, we first need to get a constructor object for that class.

CODEGYM

getConstructors and getDeclaredConstructors methods

getDeclaredConstructors() - returns an array of Constructor objects representing all constructors declared by the class, represented by this class object. These are public, protected, default (package) and private constructors.

getConstructors() - returns an array containing Constructor objects reflecting only the public constructors of the class represented by this class object.

The java.lang.reflect.Constructor class and its main methods

getName() - returns the string name of the constructor represented by the Constructor object.

getModifiers() – returns the constructor modifiers represented by the Constructor object.

getExceptionTypes() - returns an array of Class objects, representing the exception types declared for the constructor represented by the Constructor object. Returns a zero-length array if the constructor does not declare exceptions in its throws.

getParameters() - returns an array of Parameter objects that represent all the parameters of the constructor represented by the Constructor object. Returns a zero-length array if the constructor has no parameters.

getParameterTypes() - returns an array of Class objects that represent the formal parameter types in the order of declaration in the constructor represented by the Constructor object. Returns a zero-length array if the constructor takes no parameters. Note that the constructors of some inner classes may have an implicitly declared parameter in addition to the explicitly declared ones.

getGenericParameterTypes() - returns an array of Type objects that represent the formal parameter types in the order of declaration in the constructor represented by the Constructor object.

Creating an object using Constructor.newInstance()

In order to use the **Constructor.newInstance()** method, we first get the constructor object for this class, and then call **newInstance():**

```
Constructor<Employee> constructor = Employee.class.getConstructor();
Employee employee = constructor.newInstance();
```

Difference between Class.newInstance() and Constructor.newInstance()

Both methods have the same names, but there are differences between them:

1. Class.newInstance() can only call a constructor with no arguments. Constructor.newInstance() can call any constructor, regardless of the number of parameters.

2. Class.newInstance() requires the constructor to be visible. Constructor.newInstance() can also call private constructors under certain circumstances.

3. Class.newInstance() throws any exception (checked or unchecked) thrown by the constructor. Constructor.newInstance() always wraps the thrown exception in an InvocationTargetException.

For the above reasons, **use of Constructor.newInstance() is preferred**, which is why it is used by various frameworks and APIs such as Spring, Guava, Zookeeper, Jackson, Servlet, etc.

Changing the internal state of an object using reflection

How do we change the value of a variable using reflection?
To do this, we have **setByte()**, **setShort()**, **setInt()**, **setLong()**, **setFloat()**, **setDouble()**, **setChar()**, **setBoolean()** and **set()** methods which accept reference data types.

```
Cat cat = new Cat("Tom");
Class<? extends Cat> catClass = cat.getClass();
Field nameField = catClass.getField("name");
nameField.set("Jerry");
```

Changing the value of a private variable

As in the case of getting the value of a private variable or changing the final variable, before using one of the set methods, you must call the setAccessible(true) method, if this is not done, then when trying to change such a variable, we will receive an IllegalAccessException.

```
Cat cat = new Cat("Tom");
Class<? extends Cat> catClass = cat.getClass();
Field nameField = catClass.getField("name");
nameField.setAccessible(true);
nameField.set("Jerry");
nameField.setAccessible(false);
```

After changing a field, it is considered good practice to set the “Accessible” flag to false.

Dynamic Proxy

There are several ways in Java to change the functionality of a desired class...

One of them is to create a dynamic proxy (Proxy).

Java has a special class (java.lang.reflect.Proxy) **using which you can actually construct an object at runtime (dynamically) without creating a separate class for it.**

This is very simple to do:

```java
Reader reader = (Reader)Proxy.newProxyInstance();
```

Java uses a special **InvocationHandler** interface that can be used to **intercept all method calls** to a proxy object.

The **InvocationHandler** interface has a single **invoke method, to which all calls to the proxy object are routed**.

Homework

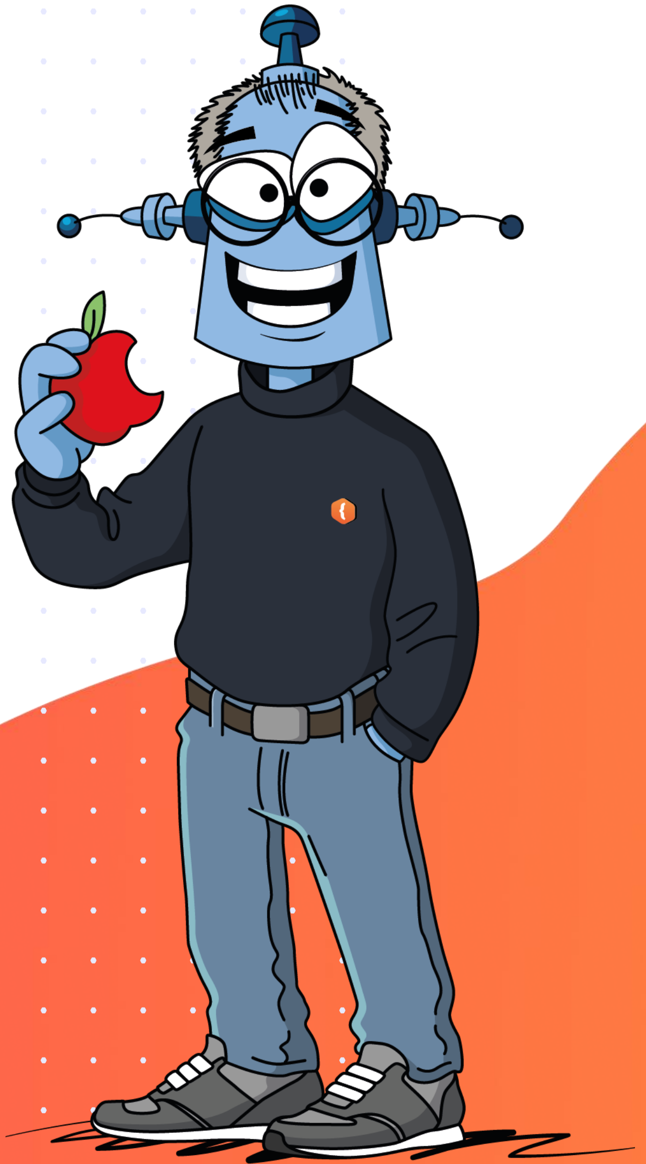
Module 2

Level 17 Reflection API

MODULE 2. JAVA CORE

CODEGYM

Questions and Answers



MODULE 2. JAVA CORE