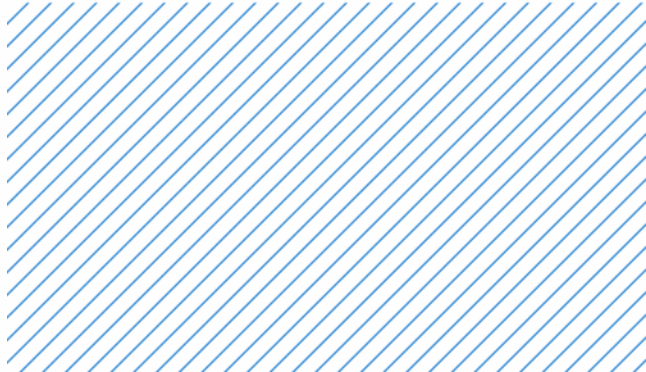
HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

Лекция 15

Рефлексия и аннотации

Программирование на языке Java

Роман Гуров
ВШЭ БИ 2021



Рефлексия

Рефлексия (англ. **reflection**) – знание кода о самом себе

К рефлексии можно отнести возможность проитерироваться по всем полям класса
или найти и создать объект класса, по имени, заданному через текстовую строку

Тот факт, что `Java` работает через прослойку в виде `JVM`,
открывает богатый набор функциональности, связанной с рефлексией

Дальше мы просто рассмотрим различные методы и классы для рефлексии

Класс `Class<T>`

Класс `Class` – основной класс для рефлексии в `Java`

```
public final class Class<T> implements /* куча всего незнакомого */ {
    // много всяких методов и полей
}
```

Его объекты описывают некоторый конкретный класс, и позволяют выполнять с ним множество действий

Дженерик-параметр `T` задаёт этот самый класс, который описывем

Конструкторы приватные, поэтому создать объект просто так через `new` не выйдет

Как же тогда получить объект класса `Class`?

Получение Class<T>

Есть три способа получить объект Class

1) Использовать псевдополе .class:

```
Class<String> cl = String.class;
```

Работает даже с примитивными типами!

Важно, что это на самом деле специальная конструкция языка, а не настоящее поле
Её нельзя использовать на объекте, только на самом классе

2) Вызвать метод getClass() класса Object:

```
CharSequence seq = "My string";
Class<? extends CharSequence> cl = seq.getClass(); // вернётся Class<String>
```

Этот метод учитывает полиморфизм и возвращает реальный класс, которым является объект

Получение `Class<T>`

Есть три способа получить объект `Class`

3) Найти этот класс по строчному имени с помощью статического метода `Class::forName`:

```
Class<?> integer_class = Class.forName("java.lang.Integer");
```

В случае, если класс с таким именем не найден,
будет прошено **проверяемое** исключение `ClassNotFoundException`:

```
try {
    Class<?> integer_class = Class.forName("java.lang.Integer");
    // ... construct object? do something?
} catch (ClassNotFoundException ex) {
    System.out.print("Couldn't find the specified class");
}
```

Получение имени класса из `Class`

Теперь перейдём к рассмотрению возможностей `Class`

Имеется несколько методов для получения различных вариантов имени класса:

- `getName()`
  - Полное имя класса (`java.lang.Integer`)
- `getSimpleName()`
  - Короткое имя без пакета (`Integer`)
- `getCanonicalName()`
  - *Каноническое* имя класса, как в импортах (`java.lang.Integer`)
- `getTypeName()`
  - Чисто информативная строка для имени типа (`java.lang.Integer`)
- `getPackage().getName()`
  - Получить только имя пакета (`java.lang`)

Отличия проще всего посмотреть в этом ответе на StackOverflow

Получение модификаторов класса

Метод `getModifiers()` позволяет узнать все модификаторы, с которыми был объявлен класс:

```
public native int getModifiers();
```

Модификаторы пакуются внутри битов инта, распаковать информацию можно методами класса `Modifier`:

```
public static boolean isPublic(int mod) { }
public static boolean isPrivate(int mod) { }
public static boolean isProtected(int mod) { }
public static boolean isStatic(int mod) { }
public static boolean isFinal(int mod) { }
public static boolean isInterface(int mod) { }
public static boolean isAbstract(int mod) { }
```

В печатном объяснении не нуждаются :)

```
Modifier.isPublic(Main.class.getModifiers())
```

Получение родителей

Метод `getSuperclass()` возвращает `Class` родителя текущего класса

```
public native Class<? super T> getSuperclass();
```

Если текущий `Class` описывает `Object`, интерфейс, примитивный тип или `void` (и такое бывает!), то вернётся `null`

Метод `getInterfaces()` возвращает список `Class`'ов интерфейсов, реализуемых текущим классом

```
public Class<?>[] getInterfaces()
```

Интерфейсы будут перечислены в том же порядке, что и при объявлении класса

Если вызвать метод от интерфейса, то вернётся список всех интерфейсов, которые он `extend`'ит

Если вызвать от примитивного типа, `void`'а или класса без реализуемых интерфесов,
то вернётся пустой массив

Задача: получение всех интерфейсов

Метод `getInterfaces()` возвращает только интерфейсы, указанные в `implements` конкретного класса

При этом, суперклассы и сами интерфейсы могут наследовать нам дополнительные интерфесы:

```
interface A {}
interface B {}
class C implements A {}
class D extends C implements B {}
```

Для класса `D` метод вернёт только `B`:

```
for (Class<?> cls : D.class.getInterfaces())
    System.out.println(cls.getName());
```

```
ru.hse.lecture15.B
```

Задача: написать метод, получающий все интерфейсы,
которые имплементирует ребенок в иерархии наследования
(для простоты, не учитываем родителей у интерфейсов)

Задача: получение всех интерфейсов

```
interface A {}
interface B {}
class C implements A {}
class D extends C implements B {}
```

Задача: написать метод, получающий все интерфейсы, которые имплементирует ребенок в иерархии наследования

```
public class Main {
    public static Class<?>[] getAllInterfaces(Class<?> cls) {
        List<Class<?>> interfaces = new ArrayList<>();
        while (cls != Object.class) {
            interfaces.addAll(Arrays.asList(cls.getInterfaces()));
            cls = cls.getSuperclass();
        }
        return interfaces.toArray(Class<?>[]::new);
    }

    public static void main(String[] args) {
        System.out.println(Arrays.toString(getAllInterfaces(D.class)));
    }
}
```

```
[interface ru.hse.lecture15.B, interface ru.hse.lecture15.A]
```

Работа с полями

Метод `getFields()` возвращает все публичные поля класса или интерфейса, включая унаследованные:

```
public Field[] getFields() throws SecurityException
```

- `SecurityException` бросается в случае запрета доступа к пакету, в котором лежит класс
  - Этот механизм мы разбирать не будем
- Статические поля также возвращаются
- Порядок полей в массиве произвольный

Метод `getDeclaredFields()` возвращает вообще все поля класса или интерфейса, но *исключая* унаследованные:

```
public Field[] getDeclaredFields() throws SecurityException
```

`getField()` и `getDeclaredField()` позволяют найти поле по его строчному имени:

```
public Field getDeclaredField(String name) throws NoSuchFieldException, SecurityException
public Field getField(String name) throws NoSuchFieldException, SecurityException
```

- `NoSuchFieldException` бросается, если поле с таким именем найти не удалось

Класс `Field`

Поля представляются специальным типом `Field`:

```
package java.lang.reflect;

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

Рассмотрим основные его методы:

- `get()` и `set()` позволяют прочитать и записать значение поля:

```
public Object get(Object obj) throws IllegalArgumentException, IllegalAccessException
public void set(Object obj, Object value) throws IllegalArgumentException, IllegalAccessException
```

  - Если работаем со `static`-полем, то объект можно указать любой. Лучше – `null`
  - Примитивные типы автоматически превращаются в свои обертки и обратно
  - `IllegalArgumentException` бросается, если перед неподходящий объект
  - `IllegalAccessException` бросается при несоблюдении модификаторов доступа
- Вызов метода `setAccessible(true)` разрешает игнорировать модификаторы доступа для этого объекта `Field`
- `getName()` возвращает имя поля
- `getType()` возвращает объект `Class` для его типа
- `getModifiers()` тоже есть и работает так же, как для `Class`
- `toString()` возвращает красивую строку с описанием поля:
  “`public static final boolean ru.hse.lecture15.AA.a`”

Задача: вывод полей класса

```
class Task {
    public static boolean a = false;
    static String b = "";
    protected Integer i;
    private Scanner scanner;
}
```

Задача: вывести все поля класса и их типы

```
public class Main {
    public static void printAllFields(Class<?> cls) {
        for (Field field : cls.getDeclaredFields()) {
            int mods = field.getModifiers();
            if (Modifier.isPublic(mods)) System.out.print("public ");
            if (Modifier.isProtected(mods)) System.out.print("protected ");
            if (Modifier.isPrivate(mods)) System.out.print("private ");
            if (Modifier.isStatic(mods)) System.out.print("static ");
            if (Modifier.isFinal(mods)) System.out.print("final ");
            System.out.println(field.getType().getCanonicalName() + ' ' + field.getName());
        }
    }

    public static void main(String[] args) {
        printAllFields(Task.class);
    }
}
```

```
public static boolean a
static java.lang.String b
protected java.lang.Integer i
private java.util.Scanner scanner
```

Работа с методами

Методы для получения методов похожи на те, что были для полей:

```
public Method[] getMethods() throws SecurityException
public Method[] getDeclaredMethods() throws SecurityException
public Method getMethod(String name, Class<?>... parameterTypes) throws NoSuchMethodException, SecurityException
public Method getDeclaredMethod(String name, Class<?>... parameterTypes) throws NoSuchMethodException, SecurityException
```

Смысл абсолютно тот же, только возвращаются методы

Статические поля наследуемых или реализуемых интерфейсов не находятся этими методами

Для поиска метода по имени, также нужно перечилить его параметры их `Class`'ами, начиная со второго аргумента

`NoSuchMethodException`, логично догадаться, бросается, когда метод с такими именем и сигнатурой не найден

Класс Method

Посмотрим на основные методы класса Method:

- getModifiers()
- getName()
- toString()
- setAccessible()
  - Эквивалентно полям
- getParameterTypes()
  - Возвращает массив всех аргументов метода
- getReturnType()
  - Возвращает возвращаемый тип метода
- invoke()
  - Вызывает метод:

```
Class<?>[] getParameterTypes()
```

```
Class<?> getReturnType()
```

```
Object invoke(Object obj, Object... args) throws IllegalAccessException, IllegalArgumentException, InvocationTargetException
```

Первый аргумент – объект, у которого вызываем метод (для static-метода можно null)

Дальше передаются аргументы для метода, примитивные типы принимают класс-обертку

- IllegalArgumentException бросается, когда объект не является наследником хозяина метода или не совпали типы / количество аргументов
- InvocationTargetException оборачивает исключение, которые бросил метод в процессе вызова

Задача: вызов метода

```
class Task2 {
    private int secretStupidMethod(String s) {
        return s.chars().sum();
    }
}
```

Задача: вызвать метод и обработать все возможные ошибки

```
public static void main(String[] args) {
    Task2 instance = new Task2();
    Class<?> cls = instance.getClass();
    try {
        Method meth = cls.getDeclaredMethod("secretStupidMethod", String.class);
        meth.setAccessible(true);
        int result = (Integer) meth.invoke(instance, "qwerty");
        System.out.println(result);
    } catch (NoSuchMethodException ex) {
        System.out.println("No such method");
    } catch (IllegalAccessException ex) {
        System.out.println("The method access modifiers forbid calling it");
    } catch (IllegalArgumentException ex) {
        System.out.println("Incorrect arguments given");
    } catch (InvocationTargetException ex) {
        System.out.println(“The method has thrown an exception");
    }
}
```

Работа с конструкторами

Методы для получения конструкторов уже можно предугадать:

```
public Constructor<?>[] getConstructors() throws SecurityException
public Constructor<?>[] getDeclaredConstructors() throws SecurityException
public Constructor<T> getConstructor(Class<?>... parameterTypes) throws NoSuchMethodException, SecurityException
public Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes) throws NoSuchMethodException, SecurityException
```

Все элементы нам уже знакомы из полей и методов

Обратим внимание, что в возвращаемом массиве типы обозначены wildcard’ом (?)
Это сделано просто для того, чтобы в массив можно было накинуть потом и любых других конструкторов

Класс Constructor

```
package java.lang.reflect;

public final class Constructor<T> extends Executable {
```

Основные методы класса `Constructor` тоже не будут для нас сюрпризом:

- `getModifiers()`
- `getName()`
- `toString()`
- `getParameterTypes()`
- `setAccessible()`
  - Всё идентично методам
- `newInstance()`
  - Создаёт новый инстанс класса, вызвав конструктор

```
T newInstance(Object... initargs)
      throws InstantiationException, IllegalAccessException, IllegalArgumentException, InvocationTargetException
```

И тут тоже идентично методам, исключения тоже имеют тот же смысл

`InstantiationException` вылетит, если вызывается конструктор абстрактного класса

Для вложенных нестатических классов (не забыли, что это?) первым аргументом будет объект внешнего класса

Задача: сконструировать класс

```
class Task3 {
    public Task3(int a, String b) { }
}
```

Задача: вызвать конструктор и обработать все возможные ошибки

```
public static void main(String[] args) {
    Class<Task3> cls = Task3.class;
    try {
        Constructor<Task3> constructor = cls.getDeclaredConstructor(int.class, String.class);
        Object result = constructor.newInstance(1337, "qwerty");
        System.out.println(result);
    } catch (NoSuchMethodException ex) {
        System.out.println("No such method");
    } catch (IllegalAccessException ex) {
        System.out.println("The method access modifiers forbid calling it");
    } catch (IllegalArgumentException ex) {
        System.out.println("Incorrect arguments given");
    } catch (InvocationTargetException ex) {
        System.out.println("The method has thrown an exception");
    } catch (InstantiationException ex) {
        System.out.println("Class is abstract");
    }
}
```

```
ru.hse.lecture15.Task3@4c873330
```

Задача: вывести все методы и конструкторы

```
class Task4 {
    private Task4() {}
    public Task4(int a, String b) { }
    private int secretStupidMethod(String s) { }
    static boolean meth() { }
}
```

Задача: Вывести все методы и их сигнатуры
(в том числе конструкторы)

```
public static void printModifiers(int mods) {
    if (Modifier.isPublic(mods)) System.out.print("public ");
    if (Modifier.isProtected(mods)) System.out.print("protected ");
    if (Modifier.isPrivate(mods)) System.out.print("private ");
    if (Modifier.isStatic(mods)) System.out.print("static ");
    if (Modifier.isFinal(mods)) System.out.print("final ");
}

public static void printParams(Class<?>[] arg_types) {
    System.out.println(Arrays.stream(arg_types)
            .map(Class::getCanonicalName)
            .collect(Collectors.joining(", ", "(", ")")));
}
```

```
public static void printAllMethods(Class<?> cls) {
    for (Constructor<?> constructor : cls.getDeclaredConstructors()) {
        printModifiers(constructor.getModifiers());
        System.out.print(constructor.getName());
        printParams(constructor.getParameterTypes());
    }
    for (Method meth : cls.getDeclaredMethods()) {
        printModifiers(meth.getModifiers());
        System.out.print(meth.getReturnType().getSimpleName() + ' ' + meth.getName());
        printParams(meth.getParameterTypes());
    }
}

public static void main(String[] args) {
    printAllMethods(Task4.class);
}
```

```
private ru.hse.lecture15.Task4()
public ru.hse.lecture15.Task4(int, java.lang.String)
static boolean meth()
private int secretStupidMethod(java.lang.String)
```

ПОВТОР

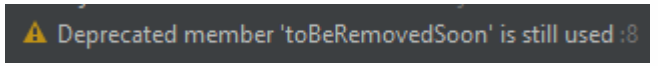
Аннотации

Аннотации – специальные метки, которые можно применять к разным сущностям языка

```
public class Main {
    public static void main(String[] args) {
        String result = toBeRemovedSoon();
        System.out.println(result);
    }

    @Deprecated
    public static String toBeRemovedSoon() {
        return "ded";
    }
}
```

@Deprecated – аннотация, используемая, чтобы сообщить о том, что нечто устарело и может быть удалено в следующей версии



@SuppressWarnings – аннотация, используемая, чтобы подавлять некоторые предупреждения

```
public class Main {
    public static void main(String[] args) {
        String result = iDontCare();
        System.out.println(result);
    }

    @SuppressWarnings("deprecated")
    public static String iDontCare() {
        return toBeRemovedSoon();
    }

    @Deprecated
    public static String toBeRemovedSoon() {
        return "ded";
    }
}
```

Аннотации

Аннотации – специальные метки, которые можно применять к разным сущностям языка

Важно уточнить, что аннотации ничего не делают сами по себе, это буквально метки, которые вешаются на различные сущности: классы, методы, поля и т.д.

Чтобы аннотация что-либо сделала, где-то в коде должен быть её обработчик

У аннотаций есть три области применения:

- Информация для компилятора
- Обработка на этапе компиляции и сборки, например для генерации файлов
- Обработка во время исполнения программы

Научимся делать свои аннотации

Создание аннотации

```
public class MyClass {
    // Author: John Doe
    // Date: 3/17/2002
    // Current revision: 6
    // Last modified: 4/12/2004
    // By: Jane Doe
    // Reviewers: Alice, Bill, Cindy

    // class code goes here
}
```

Допустим, при разработке для каждого класса было принято писать информацию о нём

Всё это можно объявить с помощью аннотации

Аннотация объявляется как интерфейс, только с символом @

Параметры аннотации объявляются как методы без аргументов

`default` позволяет сделать параметры опциональными

При обработке аннотации мы действительно будем работать с ней как с интерфейсом

```
@interface ClassPreamble {
    String author();
    String date();
    int currentRevision() default 1;
    String lastModified() default "N/A";
    String lastModifiedBy() default "N/A";
    // Note use of array
    String[] reviewers();
}
```

```
@ClassPreamble (
    author = "John Doe",
    date = "3/17/2002",
    currentRevision = 6,
    lastModified = "4/12/2004",
    lastModifiedBy = "Jane Doe",
    // Note array notation
    reviewers = {"Alice", "Bob", "Cindy"}
)
public class MyClass2 {
    // class code goes here
}
```

Так будет выглядить применение нашей аннотации к классу

Но использовать их мы пока не умеем

Создание аннотации

Существуют аннотации, которые применимы к другим аннотациям при объявлении:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({/* ... */})
public @interface Deprecated {
    String since() default "";

    boolean forRemoval() default false;
}
```

Их ещё называют *мета-аннотациями*

Рассмотрим 4 основных мета-аннотации:

- `@Documented`
  - Заставляет аннотацию с её параметрами попадать в JavaDoc объекта, на который повешена
- `@Inherited`
  - Наследники аннотированного класса тоже будут иметь эту аннотацию
- `@Target`
  - Перечисляет, к чему можно применить аннотацию
- `@Retention`
  - Определяет, как далеко эта аннотация доживает в процессе компиляции

Создание аннотации

Последние две мета-аннотации разберём поподробнее

```java
package java.lang.annotation;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.ANNOTATION_TYPE})
public @interface Target {
    ElementType[] value();
}
```

```java
package java.lang.annotation;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.ANNOTATION_TYPE})
public @interface Retention {
    RetentionPolicy value();
}
```

- `@Target` – принимает массив:
  - `ElementType.ANNOTAION_TYPE` – другие аннотации
  - `ElementType.CONSTRUCTOR` – конструкторы
  - `ElementType.FIELD` – поля классов
  - `ElementType.LOCAL_VARIABLE` – локальные переменные внутри функций
  - `ElementType.METHOD` – методы
  - `ElementType.PACKAGE` – целые пакеты
  - `ElementType.PARAMETER` – аргументы функций
  - `ElementType.TYPE` – классы и интерфейсы
- `@Retention` – принимает ровно одно значение:
  - `RetentionPolicy.SOURCE` – аннотация только для кода и не попадает в программу
  - `RetentionPolicy.CLASS` – аннотация сохраняется в байткоде, но игнорируется в рантайме
  - `RetentionPolicy.RUNTIME` – аннотация видна при выполнени и к ней можно будет обратиться

Обращение к аннотации в рантайме

Для проверки наличия аннотации у класса, существуют методы класса `Class`:

```
boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)
<A extends Annotation> A getAnnotation(Class<A> annotationClass)
Annotation[] getAnnotations()
Annotation[] getDeclaredAnnotations()
```

- `isAnnotationPresent()`
  - Проверяет, есть ли у класса данная аннотация
- `getAnnotation()`
  - Возвращает аннотацию, если она есть у класса, иначе `null`
- `getAnnotations()`
  - Возвращает все аннотации, применённые к классу
- `getDeclaredAnnotations()`
  - Как предыдущий, но не учитывает аннотации, унаследованные с помощью `@Inherited`

wen code is dark alway rember doc:
Class, Modifier, Field, Method, Constructor, Annotations Tutorial

Пример

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@interface Service {
    boolean lazyLoad () default false;
}

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@interface Init {
    boolean suppressException () default false;
}
```

```java
@Service
class SimpleService {
    @Init
    public void init () {
        System.out.println("Simple service initialized");
    }
}

@Service(lazyLoad = true)
class LazyService {
    @Init(suppressException = true)
    public void init () {
        System.out.println("Lazy service initialized");
        throw new RuntimeException("Exception from lazy service");
    }
}
```

```java
class Main {
    public static void main (String[] args) {
        Object[] services = new Object[] { new SimpleService(), new LazyService(), new String("Hello!") };

        for (final Object service : services)
            tryInitService(service);
    }
```

Пример

```
    private static void tryInitService (final Object service) {
        Class<?> serviceClass = service.getClass();

        if (!serviceClass.isAnnotationPresent(Service.class))
            return;

        for (final Method method : serviceClass.getMethods())
            if (method.isAnnotationPresent(Init.class)) {
                try {
                    method.invoke(service);
                } catch (final IllegalAccessException exception) {
                    System.out.println("Can't access the init method of the service " + serviceClass.getSimpleName());
                    System.out.println(exception.getMessage());
                } catch (final IllegalArgumentException exception) {
                    System.out.println("Init method signature mismatch in " + serviceClass.getSimpleName());
                    System.out.println(exception.getMessage());
                } catch (final InvocationTargetException exception) {
                    if (!method.getAnnotation(Init.class).suppressException()) {
                        System.out.println(
                                "An exception has been thrown during the initialization of the service " +
                                        serviceClass.getSimpleName()
                        );
                        System.out.println(exception.getTargetException().getMessage());
                    }
                }
            }
    }
}
```