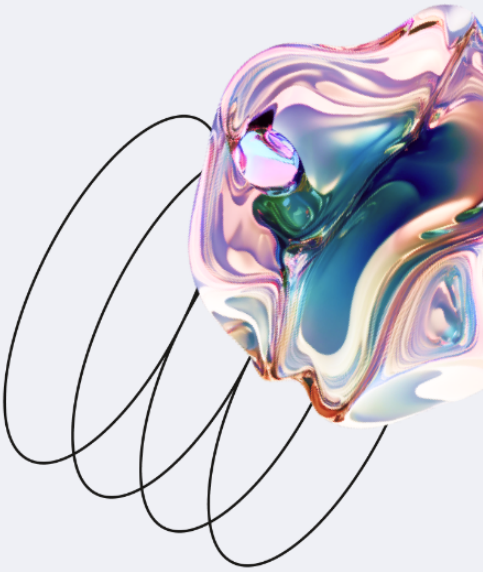
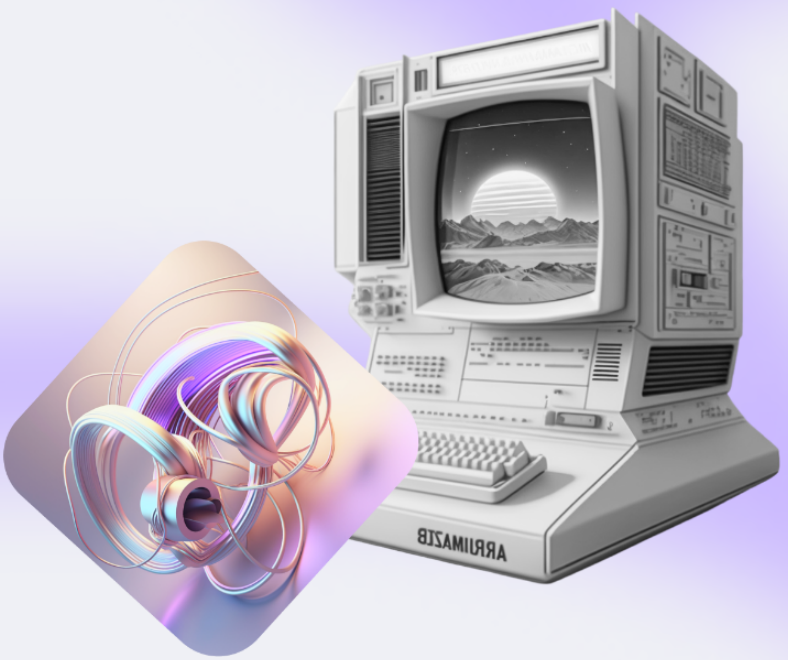
GeekBrains



Reflection API

Java Junior



Оглавление

Reflection API

Добрый день! Я приветствую вас на второй лекции, и посвящена она будет рефлексии! Это поздняя латынь и означает взгляд в себя или обращение в себя или обращение назад. Это способность мыслящих, разумных существ анализировать прошедший день или прошедшее событие. Анализировать свои реакции на события и на других участников событий. Возможно менять себя так, чтобы следующий день был ярче и продуктивней! И, казалось бы, к программированию это не имеет никакого отношения. Однако на платформе Java есть библиотека под названием reflect. И интерфейс работы с ним Reflection API. В общем-то программисты Java могут написать приложение способное рефлексировать! Что же это значит? Ну, во-первых, чтобы проанализировать свои данные и реакции нужно к ним иметь доступ! И рефлексия такой доступ даёт! Во-вторых, менять себя. Конечно речь не о самомодифицирующимся коде, его в Яве просто нет, а речь о параметрах полей. Рефлексия и тут даёт нам полный простор действий и контроль происходящего! А знаете, считаю пора перейти к примерам. И показать уже на них!

Пример:

```
class Car{}
Car car = new Car();
```

Тут всё просто, вложенный класс Car и его экземпляр car с конструктором по умолчанию. Мы сами написали класс и сами знаем что конструктор не переопределён и поэтому используем конструктор по умолчанию! А вдруг не знаем? Вдруг мы тестеры и понятия не имеем что там наворотили программисты а работу очень хочется автоматизировать. Например вот так:

Пример:

```
class Car{
   String name;
   public Car(String name) {
      this.name = name;
   }
}
Car car = new Car();
```

Появилось поле имя и конструктор изменился. Теперь приложение даже не соберётся, скажет:"Что то у вас с конструктором не то!". Как же быть? Ну мы можем покопаться в коммитах и почитать отчёты или в самом коде поискать изменения. А можем найти конструктор методами рефлексии. Заменим объявление и реализацию объекта car.

Пример:

```
Class<?> car = Class.forName("Car");
```

Стало как то по другому. Во первых теперь car это экземпляр параметризированного Class, а вместо ключевого new и конструктора некий метод и строка в параметре!? Что это? Class это и есть основа рефлексии, именно через него мы и будем с ней работать! Вообще Class создаётся при загрузке любого динамического класса и содержит информацию о его структуре! То есть как раз то что нам нужно! Объект car содержит не информацию о машине, а информацию о классе, машину описывающем! Посмотрим что можно у него узнать. Ну например конструктор.

Пример:

```
Constructor<?>[] constructor = car.getConstructors();
```

Тут всё довольно просто, метод getConstructors() возвращает список всех публичных конструкторов объявленных в классе. Есть ещё метод getDeclaredConstructors(), он возвращает и приватные конструкторы тоже но об этом поговорим сегодня позже. Итак у нас есть список конструкторов в данном случае из одной строки.

public Car(java.lang.String)

Понятно что это публичный конструктор и что у него один параметр String, и это уже хорошо!) Теперь мы можем создать экземпляр класса Car вызвав его конструктор с

правильным параметром. Ну а если конструкторов несколько позволит выбрать самый удобный. Вообще-то любая среда разработки и так скажет нам какие конструкторы есть у класса и какие параметры они ждут. Такой подход применяется если вам надо создать экземпляр класса, которого не было во время разработки программы. А появился он уже во время её работы. Это одна из особенностей рефлексии. Она позволяет работать с новыми классами и изменять уже инициализированные объекты во время исполнения программы. Давайте немного изменим класс, а потом загрузим его средствами рефлексии.

Пример:

```
class Car{
   String name;
   public Car(String name) {this.name = name;}

   @Override
   public String toString() {return "Car {name = " + name + '}';
   }
}

Object gaz = constructor[0].newInstance("ГАЗ-311055");
System.out.println(gaz);
```

В классе Car переопределён метод toString и всё. А вот в основном классе я создал объект gaz, но тоже не обычно. Во первых в объявлении он Object а не Car а во вторых с правой стороны не обычный конструктор а тот самый наш constructor. Тут надо сделать уточнение, класс Constructor импортируется из пакета reflect и это значит что, используя его методы, мы фактически пользуемся Reflection API. Метод newInstance() возвращает инициализированный объект указанного конструктора, а нужные для конструктора параметры передаются в параметрах ему. В нашем случае это строка. Дальше я вывожу в консоль строку возвращаемую методом toString() и всё как надо, в консоли мы видим "Car {name = ГАЗ-311055}" а значит перед нами экземпляр класса Car а не Object! Теперь мы можем подключать библиотеки, загружать из них классы и создавать экземпляры классов, даже не зная точно их имён или параметров конструкторов! И всё это в рантайме! Немного погрузились в рефлекшен, двинемся дальше. В любом классе кроме конструкторов должны быть методы и поля данных. У нас есть одно поле name, давайте добавим ещё несколько.

Пример:

```
class Car{
   public String name;
   private String price;
   public String engType;
   public int engPower;
   public int maxSpeed;

    public Car(String name) {this.name=name; engType="DOHC 2.4L"; engPower=137;
maxSpeed=190; price="Не доступно!";}

   @Override
   public String toString() {return "Car {name = " + name + '}';
   }
}

Field[] fields = gaz.getClass().getFields();
int tmp = fields[fields.length-1].getInt(gaz);
fields[fields.length-1].setInt(gaz, tmp+100);
```

В классе Car появилось несколько полей и инициализация их в конструкторе. Обычный код никаких особенностей. В мэйн я добавил ещё один класс Reflection API это класс Field. В нём можно хранить поля данных анализируемого нами класса. Но заполняется данными из инициализированного объекта, а значит и получить их мы можем из объекта gaz вызвав метод getClass() и getField(). Мы получили доступ к полям!) Конечно можно прочесть их имя но сейчас я записал в tmp просто текущее значение последнего поля. Для примитивных типов в классе Field определены специальные геттеры, чтобы привидением типов не заниматься! А для установки новых значений есть в классе Field метод set, для примитивных типов специальные сеттеры. Им мы воспользовались для обновления значения поля maxSpeed! Вернёмся к коду нашего класса и посмотрим подробнее. Одно из полей приватно! И в массиве fields ссылки на это поле нет, так как метод getField() возвращает список только публичных полей! Но есть ещё один метод!

Пример:

```
Field[] fields = gaz.getClass().getDeclaredFields();
for (Field field: fields) {
   if (field.getName().equals("price")) {
      field.setAccessible(true);
      field.set(gaz, "Доступно!");
      field.setAccessible(false);
   }
```

Изменений снова не так много, но они важные. Вопервых я использую метод getDeclaredField() и получаю список полей и публичных и приватных. Потом в цикле методом getName() нахожу поле с именем "price". И вот теперь самый пока интересный момент. Методом setAccessible() я могу сделать поле публичным, true, или снова приватным false. Это немного обходит правила инкапсуляции, но позволяет читать и даже изменять значения любых полей объекта! Теперь мы можем в рантайме загрузить любую библиотеку, выбрать из неё нужный класс и создать экземпляр с корректным конструктором. А также можем прочитать и даже изменить значения всех полей объекта! Но это ещё не всё, ведь у любого класса должны быть и методы! Давайте добавим пару методов в наш класс.

Пример:

```
class Car{
   public String name;
   private String price;
   public String engType;
   public int engPower;
   public int maxSpeed;

    public Car(String name) {this.name=name; engType="DOHC 2.4L"; engPower=137;
maxSpeed=190; price="Не доступно!";}

   @Override
   public String toString() {return "Car {name = " + name + '}';
   }
```

```
    public String getPrice() {return price;}
    private void setPrice(String newPrice) {price = newPrice;}
}
```

Два новых метода. Оба посвящены цене. Обычный геттер и сеттер. Ну и изменения в нашей основной программе

Пример:

```
Method[] methods = gaz.getClass().getMethods();
for (int i = 0; i < methods.length; i++) {
    System.out.println(methods[i]);
}
```

У нас появился ещё один класс из Reflection API, это класс methods. Он описывает методы, и заполнить его можно вызвав методы getClass() и getMethods() нашего объекта. Массив methods заполнится всеми доступными методами, и вот что мы увидим в консоли.

Пример:

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

public java.lang.String Car.getPrice()

public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException

public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException

public final void java.lang.Object.wait() throws java.lang.InterruptedException

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

```
public native int java.lang.Object.hashCode()

public final native java.lang.Class java.lang.Object.getClass()

public final native void java.lang.Object.notify()

public final native void java.lang.Object.notifyAll()
```

Немного не то, что ожидали! Здесь и нативный getClass(), которым мы только что пользовались и нативные notifyAll() и notify(). hashCode(), equals и три перезагрузки wait. И всего два метода объявленных в самом классе. Дело в том, что getMethods(), как и getField(), возвращает все публичные методы и поля объявленные непосредственно в классе и унаследованные из супер классов тоже! А давайте уберём наследие и оставим только свои методы.

Пример:

```
Method[] methods = gaz.getClass().getDeclaredMethods();
for (int i = 0; i < methods.length; i++) {
   System.out.println(methods[i]);
}
```

Всего изменений это getMethods() изменился на getDeclaredMethods()!

А вот вывод в консоль.

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

public java.lang.String Car.getPrice()

private void Car.setPrice(java.lang.String)
```

Остались только объявленные в классе Car методы и приватный метод setPrice()!

Заключение

В данной лекции мы рассмотрели понятие рефлексии, ее значение для программирования и применение в Java. Вы узнали о библиотеке reflect и ее интерфейсе Reflection API, который позволяет программе получать информацию о себе, а также выполнять различные операции с классами и объектами во время выполнения.

Мы рассмотрели основные классы и интерфейсы Reflection API, такие как Class, Field, Method, Constructor, Parameter, TypeVariable, Member, Module, Package и Annotation. Вы научились работать с этими классами, создавая экземпляры классов, получая информацию о них, а также выполняя различные операции, включая вызов методов, изменение значений полей и параметров, создание объектов и многое другое.

На протяжении лекции вы выполняли практические задания, которые помогли вам лучше понять и усвоить материал. В качестве домашнего задания вам предлагалось написать программу, демонстрирующую использование Reflection API для создания объектов и вызова методов с параметрами.