Reflection API. Рефлексия. Темная сторона Java

(тут 5 листов - дальше комментарии)

Закрывать эту возможность нужно с помощью класса SecurityManager, который нужно запустить, добавив в свой класс статический блок:

static{

if(System.getSecurityManager()==null)

System.setSecurityManager(new SecurityManager());

}

А так все эти геттеры \ сеттеры, что “мертвому припарка” !!!

Если не закрыто статическим блоком, то не ломается только **final !!!!!**

Приветствую вас, юный падаван. В этой статье я поведаю о Силе, мощь которой java-программисты используют только в, казалось бы, безвыходной ситуации. Итак, темная сторона Java — Reflection API

Рефлексия в Java осуществляется с помощью Java Reflection API. Что такое эта рефлексия? Существует короткое и точное, а также популярное на просторах интернета определение. **Рефлексия** *(от позднелат. reflexio — обращение назад)* — это механизм исследования данных о программе во время её выполнения. Рефлексия позволяет исследовать информацию о полях, методах и конструкторах классов. Сам же механизм рефлексии позволяет обрабатывать типы, отсутствующие при компиляции, но появившиеся во время выполнения программы. Рефлексия и наличие логически целостной модели выдачи информации об ошибках дает возможность создавать корректный динамический код. Иначе говоря, понимание принципов работы рефлексии в java открывает перед вами ряд удивительных возможностей. Вы буквально можете жонглировать классами и их составляющими.

Вот основной список того, что позволяет рефлексия:

* Узнать/определить класс объекта;
* Получить информацию о модификаторах класса, полях, методах, константах, конструкторах и суперклассах;
* Выяснить, какие методы принадлежат реализуемому интерфейсу/интерфейсам;
* Создать экземпляр класса, причем имя класса неизвестно до момента выполнения программы;
* Получить и установить значение поля объекта по имени;
* Вызвать метод объекта по имени.

Рефлексия используется практически во всех современных технологиях Java. Сложно себе представить, могла бы Java, как платформа, достигнуть такого огромного распространения без рефлексии. Скорее всего не смогла бы. С общим теоретическим представлением о рефлексии вы ознакомились, теперь приступим к ее практическому применению! Не будем изучать все методы Reflection API, только то, что реально встречается на практике. Так как механизм рефлексии подразумевает работу с классами, то и у нас будет простой класс — **MyClass**:

**public** **class** MyClass {

**private** **int** number;

**private** String name = "default";

// public MyClass(int number, String name) {

// this.number = number;

// this.name = name;

// }

**public** **int** getNumber() {

**return** number;

}

**public** **void** setNumber(**int** number) {

**this**.number = number;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**private** **void** printData(){

System.out.println(number + name);

}

}

Как мы видим, это самый обычный класс. Конструктор с параметрами закомментирован не просто так, мы к этому еще вернемся. Если вы внимательно просмотрели содержимое класса, то наверняка увидели отсутствие **getter**’a для поля **name**. Само поле **name** помечено модификатором доступа **private**, обратиться к нему вне самого класса у нас не выйдет => мы не можем получить его значение. “Так в чем проблема? — скажете вы. — Допиши **getter** или измени модификатор доступа”. И вы будете правы, но, что если **MyClass** находится в скомпилированной jar библиотеке или в другом закрытом модуле без доступа к редактированию, а на практике такое случается крайне часто. И какой-то невнимательный программист просто забыл написать **getter**. Самое время вспомнить о рефлексии! Попробуем добраться до **private** поля **name** класса **MyClass**:

**public** **static** **void** main(String[] args) {

MyClass myClass = **new** MyClass();

**int** number = myClass.getNumber();

String name = null; //no getter =(

System.out.println(number + name);//output 0null

**try** {

Field field = myClass.getClass().getDeclaredField("name");

field.setAccessible(true);

name = (String) field.get(myClass);

} **catch** (NoSuchFieldException | IllegalAccessException e) {

e.printStackTrace();

}

System.out.println(number + name);//output 0default

}

Разберем что тут сейчас произошло:

В java есть замечательный класс **Class**. Он представляет классы и интерфейсы в исполняемом приложении Java.

Далее, чтобы получить поля этого класса нужно вызвать метод **getFields()**, этот метод вернет нам все доступные поля класса. Нам это не подходит, так как наше поле **private**, поэтому используем метод **getDeclaredFields()**, этот метод также возвращает массив полей класса, но теперь и **private** и **protected**.

- getDeclaredXXX(**Constructors/Fields/Methods**) возвращает список членов класса (конструкторов, полей или методов), объявленных при описании класса, но не унаследованных от классов-предков. При этом в список включаются все члены, независимо от их уровня защиты – т.е. public, protected, private и дружественные члены.

- getXXX(**Constructors/Fields/Methods**) возвращает полный список членов, объявленных в самом классе либо унаследованных от одного из предков. Но в этот список уже попадают только public-члены.

В нашей ситуации мы знаем имя поля, которое нас интересует, и можем использовать метод **getDeclaredField(String)**, где **String** — имя нужного поля.

Примечание: **getFields()** и **getDeclaredFields()** не возвращают поля класса-родителя!

Отлично, мы получили объект **Field** с ссылкой на наш **name**. Т.к. поле не было **публичным** (public) следует дать доступ для работы с ним. Метод **setAccessible(true)** разрешает нам дальнейшую работу. Теперь поле **name** полностью под нашим контролем!

Получить значение поля **name** можно вызовом **get(Object)** у объекта **Field**, где **Object** — экземпляр нашего класса **MyClass**. Приводим к типу **String** и присваиваем нашей переменной **name**. На тот случай если у нас вдруг не оказалось **setter**’a, для установки нового значения полю name можно использовать метод **set**:

field.set(myClass, (String) "new value");

**Поздравляю! Вы только что овладели базовым механизмом рефлексии и смогли получить доступ к private полю!** Обратите внимание на блок **try/catch** и типы обрабатываемых исключений. IDE сама укажет на их обязательное присутствие, но по их названию итак ясно зачем они здесь.

Идем дальше!

Как вы могли заметить, наш **MyClass** уже имеет метод для вывода информации о данных класса:

**private** **void** printData(){

System.out.println(number + name);

}

Но этот программист и тут наследил. Метод находится под модификатором доступа **private**, и нам пришлось самим каждый раз писать код вывода. Не порядок, где там наша рефлексия?… Напишем вот такую функцию, добирающуюся до метода “printData”:

**public** **static** **void** printData(Object myClass){

**try** {

Method method = myClass.getClass().getDeclaredMethod("printData");

method.setAccessible(true);

method.invoke(myClass);

} **catch** (NoSuchMethodException | InvocationTargetException | IllegalAccessException e) {

e.printStackTrace();

}

}

Здесь примерно такая же процедура как и с получением поля — получаем нужный метод по имени и даем доступ к нему. И для вызова объекта **Method** используем **invoke(Оbject, Args)**, где **Оbject** — все также экземпляр класса **MyClass**. **Args** — аргументы метода — наш таковых не имеет. Теперь для вывода информации мы используем функцию **printData**:

**public** **static** **void** main(String[] args) {

MyClass myClass = **new** MyClass();

**int** number = myClass.getNumber();

String name = null; //?

printData(myClass); // outout 0default

**try** {

Field field = myClass.getClass().getDeclaredField("name");

field.setAccessible(true);

field.set(myClass, (String) "new value");

name = (String) field.get(myClass);

} **catch** (NoSuchFieldException | IllegalAccessException e) {

e.printStackTrace();

}

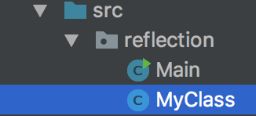
printData(myClass);// output 0new value

}

Ура, теперь у нас есть доступ к приватному методу класса.

Но что делать если у метода все таки будут аргументы, и зачем тот закомментированный конструктор? Всему свое время.

Из определения в начале ясно, что рефлексия позволяет создавать экземпляры класса в режиме **runtime** (во время выполнения программы)! Мы можем создать объект класса по полному имени этого класса. Полное имя класса — это имя класса, учитывая путь к нему в **package**.



В моей иерархии **package** полным именем **MyClass** будет “**reflection.MyClass**”. Также узнать имя класса можно простым способом (вернет имя класса в виде строки):

MyClass.**class**.getName()

Создаем экземпляр класса с помощью рефлексии(если у класса есть конструктор по умолчанию(без параметров)):

**public** **static** **void** main(String[] args) {

MyClass myClass = null;

**try** {

Class clazz = Class.forName(MyClass.**class**.getName()); //запускаем текущим ClassLoader-ом наш класс в работу (в runtime-e)

myClass = (MyClass) clazz.newInstance(); //создаем объект класса

} **catch** (ClassNotFoundException | InstantiationException | IllegalAccessException e) {

e.printStackTrace();

}

System.out.println(myClass);//output created object reflection.MyClass@60e53b93

}

На момент старта java приложения далеко не все классы оказываются загруженными в JVM. Если в вашем коде нет обращения к классу **MyClass**, то тот, кто отвечает за загрузку классов в JVM, а им является **ClassLoader**, никогда его туда и не загрузит. Поэтому нужно заставить **ClassLoader** загрузить его и получить описание нашего класса в виде переменной типа **Class**. Для этой задачи существует метод **forName(String)**, где **String** — имя класса, описание которого нам требуется. Получив **Сlass**, вызов метода **newInstance()** вернет **Object**, который будет создан по тому самому описанию. Остается привести этот объект к нашему классу **MyClass**. Круто!

Было сложно, но, надеюсь, понятно. Теперь мы умеем создавать экземпляр класса буквально из одной строки!

К сожалению описанный способ будет работать только с конструктором по умолчанию (без параметров)!!! Как же вызывать методы с аргументами и конструкторы с параметрами? Самое время раскомментировать наш конструктор. Как и ожидалось, **newInstance()** не находит конструктор по умолчанию и больше не работает. Перепишем создание экземпляра класса:

**public** **static** **void** main(String[] args) {

MyClass myClass = null;

**try** {

Class clazz = Class.forName(MyClass.**class**.getName());

Class[] params = {**int**.**class**, String.**class**};

**myClass = (MyClass) clazz.getConstructor(params).newInstance(1, "default2");**

} **catch** (ClassNotFoundException | InstantiationException | IllegalAccessException | NoSuchMethodException | InvocationTargetException e) {

e.printStackTrace();

}

System.out.println(myClass);//output created object reflection.MyClass@60e53b93

}

Для получения конструкторов класса следует у описания класса вызвать метод **getConstructors()**, а для получения параметров конструктора - **getParameterTypes()**:

Constructor[] constructors = clazz.getConstructors();

**for** (Constructor constructor : constructors) {

Class[] paramTypes = constructor.getParameterTypes();

**for** (Class paramType : paramTypes) {

System.out.print(paramType.getName() + " ");

}

System.out.println();

}

Таким образом получаем все конструкторы и все параметры к ним. В моем примере идет обращение к конкретному конструктору с конкретными уже известными параметрами. И для вызова этого конструктора используем метод **newInstance**, в котором указываем значения этим параметрам. Точно так же будет и с **invoke** для вызова методов. Возникает вопрос: где может пригодится рефлексивный вызов конструкторов? Современные технологии java, как уже говорилось в начале, не обходятся без Reflection API. Например, DI (Dependency Injection), где аннотации в сочетании с рефлексией методов и конструкторов образуют популярную в Android разработке библиотеку Darer. После прочтения этой статьи вы с уверенностью можете считать себя просвещенным в механизмы Reflection API. Темной стороной java рефлексия называется не зря. Она напрочь ломает парадигму ООП. В java инкапсуляция служит для сокрытия и ограничения доступа одних компонентов программы к другим. Используя модификатор private мы подразумеваем, что доступ к этому полю будет только в пределах класса, где это поле существует, основываясь на этом мы строим дальнейшую архитектуру программы. В этой статье мы увидели, как с помощью рефлексии можно пробираться куда угодно. Хорошим примером в виде архитектурного решения является порождающий шаблон проектирования — **Singleton**. Основная его идея в том, чтобы на протяжении всей работы программы класс, реализующий этот шаблон был только в одном экземпляре. Осуществляется это при помощи установки конструктору по умолчанию private модификатор доступа. И будет очень нехорошо, если какой-то программист со своей рефлексией будет плодить такие классы. Кстати, есть очень интересный вопрос, который я недавно услышал от своего сотрудника: может ли быть у класса, реализующий шаблон **Singleton**, наследники? Неужели в этом случае бессильна даже рефлексия? Пишите ваши feedback’и по статье и ответ в коментарии, а также задавайте свои вопросы! Истинная Сила Reflection API раскрывается в комбинации c Runtime Annotations, о чем мы, возможно, поговорим в следующей статье про темную сторону Java. Спасибо за внимание!

[0](https://vk.com/share.php?url=https://javarush.ru/groups/posts/513-reflection-api-refleksija-temnaja-storona-java)[31](https://www.facebook.com/share.php?u=https://javarush.ru/groups/posts/513-reflection-api-refleksija-temnaja-storona-java)[0](http://www.linkedin.com/shareArticle?mini=true&url=https://javarush.ru/groups/posts/513-reflection-api-refleksija-temnaja-storona-java&source=https://javarush.ru/groups/posts/513-reflection-api-refleksija-temnaja-storona-java)[0](https://twitter.com/share?url=https://javarush.ru/groups/posts/513-reflection-api-refleksija-temnaja-storona-java)[0](https://plus.google.com/share?url=https://javarush.ru/groups/posts/513-reflection-api-refleksija-temnaja-storona-java)

**+126**

**Комментарии (35)**

* **популярные**
* **новые**
* **старые**

**ДЛЯ ТОГО, ЧТО БЫ ОСТАВИТЬ КОММЕНТАРИЙ ВЫ ДОЛЖНЫ**[**АВТОРИЗОВАТЬСЯ**](https://javarush.ru/login)

[**Vadim Podvoysky**](https://javarush.ru/users/1922575)0 уровень

7 октября, 10:13

Прекрасная статья, спасибо за работу создателю! Возникла одна проблема при воспроизведении: На строчке Field field = myClass.getClass().getDeclaredField("name"); Выделяет красным Field и пишет: " Cannot resolve symbol 'Field' " Подскажите пожалуйста, как с этим бороться?

**0**

[**Alexander Klimenko**](https://javarush.ru/users/1346582)13 уровень, Харьков

17 октября, 15:23

наверняка проблема с импортом класса Field!?

**0**

[**Danny Danny**](https://javarush.ru/users/1463344)1 уровень, Киев

12 августа, 14:33

Очень классная статья, спасибо большое за работу!

**0**

[**Max Petrov**](https://javarush.ru/users/1641303)40 уровень

11 мая, 16:18

На счет "Примечание: getFields() и getDeclaredFields() не возвращают поля класса-родителя!" getFields() возвращает public поля родителей

**0**

[**Alexander Klimenko**](https://javarush.ru/users/1346582)13 уровень, Харьков

27 мая, 20:08

Не возращает, только что проверил.

**0**

[**Andrey Ivanov**](https://javarush.ru/users/1178793)21 уровень

30 апреля, 20:40

А в чем тогда вообще смысл инкапсулировать поля и некоторые методы объявляя их закрытыми, если к любому из них можно напрямую обратиться посредством рефлексии? Все равно что поставить ворота на участок, которые можно просто обойти сбоку.

**+2**

[**Denis Zheleznov**](https://javarush.ru/users/1098315)19 уровень, Москва

2 мая, 21:57

:)

**0**

[**Alexander Klimenko**](https://javarush.ru/users/1346582)13 уровень, Харьков

27 мая, 20:09

я же описал ситуацию, вот когда нужно взять и обойти всю инкапсуляцию, вот никак по другому.

**0**

[**Евгений Гродно**](https://javarush.ru/users/1549560)36 уровень

8 апреля, 02:58

Вот что нам говорит java 9, на это колдовство: WARNING: An illegal reflective access operation has occurred WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations WARNING: All illegal access operations will be denied in a future release Пробовал достучаться в полю value в классе String, провернуть это получается, но предупреждение в последней строке, говорит что это скоро закроют.

**+1**

[**Valakas**](https://javarush.ru/users/1677723)1 уровень

7 марта, 03:50

как для нуба - сложна, но интересно. дави контент, кран. UP

**0**

[**Alexander Klimenko**](https://javarush.ru/users/1346582)13 уровень, Харьков

7 марта, 09:40

Спасибо, буду!

**0**

[**Mironova**](https://javarush.ru/users/1677695)0 уровень

7 марта, 01:57

Спасибо автору за проделанную работу, пойду жонглировать хD Кстати есть у меня интересный скрин на эту тему [https://javarush.ru/api/1.0/rest/images/1677695/8cae40bc-50c6-4213-a898-698c22227459?size=64](https://javarush.ru/api/1.0/rest/images/1677695/8cae40bc-50c6-4213-a898-698c22227459?size=0)

**+1**

[**Alexander Klimenko**](https://javarush.ru/users/1346582)13 уровень, Харьков

7 марта, 09:39

Рефлексия такая рефлексия!

**0**

[**Олег**](https://javarush.ru/users/1335087)40 уровень

13 июля, 11:35

Не понял :( Как мы получили 4. Not Java O\_o

**+1**

[**Lesha Koshmanov**](https://javarush.ru/users/1610513)20 уровень, Москва

7 марта, 00:43

Сегодня прошел статью уже с написанием кода, и своими экспериментами. Не могу понять почему в JavaRush нет урока по рефлексии. Ведь это и правда сильная сторона языка. Было бы интересно порешать задачи на рефлексию. Подумать о хитрых вариантах подменить что-то или залезть куда-то.

**0**

[**Alexander Klimenko**](https://javarush.ru/users/1346582)13 уровень, Харьков

7 марта, 00:55

Тема действительно очень интересная. Не знаю какие задачи уже придумали на использование механизма рефлексии, но вот еще одна задачка, которую я услышал от сотрудника: как можно ограничить использование рефлексии в своей программе?

**0**

[**Lesha Koshmanov**](https://javarush.ru/users/1610513)20 уровень, Москва

7 марта, 01:09

Некогда раньше не сталкивался с классом SecurityManager, но догадываюсь что именно с его помощью можно запретить рефлексию. Просто загуглил этот вопрос)) https://stackoverflow.com/questions/40218973/java-security-manager-completely-disable-reflection/40220017#40220017

**0**

[**Alexander Klimenko**](https://javarush.ru/users/1346582)13 уровень, Харьков

7 марта, 01:16

В принципе да, так и ограничивается рефлексия. А вот умение гуглить - важный скил. JavaRush в некоторых своих лекциях призывает к самостоятельному поиску информации, что очень важный аспект в любом обучении =) Кстати если хочется задач, то есть отличная статья, как дополнительный материал к этой: http://codingrus.ru/readarticle.php?article\_id=662

**0**

[**Lesha Koshmanov**](https://javarush.ru/users/1610513)20 уровень, Москва

7 марта, 01:25

Интересно, спасибо еще раз))

**0**

[**Alexander Klimenko**](https://javarush.ru/users/1346582)13 уровень, Харьков

7 марта, 01:36

Рад помочь =)

**0**

[**Дмитрий Меречко**](https://javarush.ru/users/527123)40 уровень

25 апреля, 12:42

В JavaRush есть уровни по рефлексии, но они после 30-го. Тема интересная) Спасибо за статью

**0**

[**Evgeny Zinovkin**](https://javarush.ru/users/1677268)0 уровень

6 марта, 23:52

Саня, привет, статья шикарная, так держать!

**0**

[**Alexander Klimenko**](https://javarush.ru/users/1346582)13 уровень, Харьков

6 марта, 23:53

Спасибо, Жека!

**0**

[**Uzyalb**](https://javarush.ru/users/1286186)5 уровень, Москва

6 марта, 23:36

И все же, может ли как-то помочь рефлексия обойти закрытый super?

**0**

[**Alexander Klimenko**](https://javarush.ru/users/1346582)13 уровень, Харьков

7 марта, 01:05

Рефлексия в том виде, в котором она показана в статье - увы, бессильна. IDE будет требовать наличие конструктора и в свою очередь о вызове его super'a. Даже если компилировать командой javac в терминале, то будет выдана ошибка еще на этапе компиляции.

**+1**

[**Uzyalb**](https://javarush.ru/users/1286186)5 уровень, Москва

6 марта, 23:32

Спасибо, статья очень полезная, узнал много нового, по поводу singleton наследника, ответ - не выйдет, т.к. наследник не сможет вызвать super(конструктор родителя)

**+2**

[**Alexander Klimenko**](https://javarush.ru/users/1346582)13 уровень, Харьков

7 марта, 01:00

Спасибо за отзыв, да, ответ верный.

**0**

[**Светлана Греча**](https://javarush.ru/users/1392657)8 уровень, Казань

6 марта, 10:43

Всё стало понятно

**0**

[**Alexander Klimenko**](https://javarush.ru/users/1346582)13 уровень, Харьков

6 марта, 23:49

Рад, что статья оказалась полезной)

**0**

[**vitalii**](https://javarush.ru/users/1676697)0 уровень, Харьков

5 марта, 21:31

Загрузчик классов загрузит класс уже при выполнении строки reflection.MyClass.class, таким образом создав объект класса Class описывающий класс reflection.MyClass. Поэтому вызов метода Class.forName() излишен. Обычно этот метод используют для загрузки неких взаимозаменяемых классов, при этом их имена берутся из внешнего ресурса, например: использовать java.util.ArrayList или java.util.LinkedList, при этом имя класса можно передать через аргументы запуска программы. Так что в первом случае, для создания экземпляра, можно обойтись строкой:

reflection.MyClass.**class**.newInstance();

а во втором:

reflection.MyClass.**class**.getConstructor(**int**.**class**, String.**class**).newInstance(1, "one");

**0**

[**Alexander Klimenko**](https://javarush.ru/users/1346582)13 уровень, Харьков

5 марта, 23:15

Согласен, ты верно подметил, что метод *forName* используется для загрузки классов, где имена берутся из внешнего ресурса, Например, это удобно когда у нас реализована какая-то фабрика (Abstract Factory или Virtual Constructor) и классов очень много, но все они реализуют определенный интерфейс, тогда имена таких классов можно выносить в отдельный файл .properties, и PropertyResourceBundle'ом их получать для создания экземпляра.