Reflection (Рефлексия) — основы

На мой взгляд, рефлексия для Java напоминает электричество в жизни обычного человека — все им очень активно пользуются, но как оно реально работает, мало кто знает досконально и это незнание совсем не мешает. Есть общее понимание — и достаточно.
Рефлексия как идея и инструмент крайне важна для понимания огромного количества технологий, пакетов и прочего.. Поэтому понимать, что это такое и знать основы крайне важно.
С другой стороны, использование непосредственно Reflection API (набор функций для рефлексии) в жизни обычного программиста, который занимается прикладными системами, встречается достаточно редко.
Т.е. знать, что это такое, понимать основы и держать в голове плюсы и минусы — это просто необходимо. Необходимо ли досконально помнить набор функций ? Вряд ли вы часто будете их использовать. На мой взгляд, лучше понимать рефлексию и знать куда идти за дополнительной информацией. Хотя конечно же многое зависит от вашего конкретного проекта — может именно ваш будет просто “завален” кодом с использованием Reflection API. и вы достигнете такого понимания этого вопроса, который редко встречается среди профессионалов. Кто знает.

Так что же такое рефлексия и какие задачи она позволяет решать ?
Если брать технические возможности, то в первую очередь можно выделить две группы действий:

- Расширенное описание классов
- Исключения
- Решения на основе классов
- Список контактов — начало
- Коллекции — базовые принципы
- Коллекции — продолжение
- Список контактов — GUI приложение
- Что такое JAR-файлы
- Многопоточность — первые шаги
- Многопоточность и синхронизация
- Работаем с XML
- Reflection — основы
- Установка СУБД PostgreSQL
- Базы данных на Java — первые шаги
- Возможности JDBC — второй этап
- JDBC — групповые операции
- Список контактов — работаем с БД
- Переезжаем на Maven
- Потоки ввода-вывода
- Сетевое взаимодействие
- С чего начинается Web

1. Можно узнать всю информацию о классе — методы, поля, конструкторы, константы, суперклассы, реализуемые классом интерфейсы.
2. Можно работать с классом (объектом), а именно — создать объект класса, выполнить методы, получить или установить значения полей.

Если расширить список, то вот что позволяет выполнить рефлексия:

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

И еще раз в короткой форме — рефлексия позволяет вам получить информацию о внутреннем строении класса — поля, методы и т.д. — и позволяет обратиться к полям, методам и другим артифактам через эту информацию.
Причем, что самое важное — ЭТО ВСЕ МОЖНО ДЕЛАТЬ УЖЕ ПРИ ИСПОЛНЕНИИ ПРОГРАММЫ — как часто говорят, в рантайме (runtime).
Если вы никогда не сталкивались с такого рода механизмами, то у вас может возникнуть вполне резонный вопрос — а зачем все это надо ? Ответ в этом случае достаточно простой, но в то же время очень неопределенный — вы кардинально повышаете ГИБКОСТЬ и возможность НАСТРАИВАТЬ ваше приложение.

Используя рефлексию программа может вызвать абсолютно неизвестный метод абсолютно неизвестного класса уже во время своего исполнения. В момент компиляции об этом классе и методе было совершенно ничего не известно. 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.

Первое знакомство

Рассмотрим несложный вариант использования рефлексии. Ваше приложение по расписанию (или как-то иначе) должно выполнить определенный метод определенного класса. Например, вам необходимо загружать обновления финансовой информации.
До какого-то времени эта информация находилась в базе данных и метод класса умел эту информацию из этой базы получать.
Если проектирование велось с учетом возможной замены класса, то скорее всего был определен интерфейс и была сделана реализация в виде класса для базы данных. Что-то вроде такого варианта.

Сама структура, которую получаем из интерфейса может быть каким-то классом

```
public class FinanceInformation
{
    //  Здесь определяется набор полей, который описывает всю получаемую информацию
}
```

Вот наш интерфейс

```
public interface FinanceInfoBuilder
{
    FinanceInformation buildFinacneInformation();
}
```

Ну и наконец какая-то реализация для базы данных

```
public class DbFinanceInfoBuilder implements FinanceInfoBuilder
{
    public FinanceInformation buildFinacneInformation() {
        // Тут мы ходим в базу данных - как-будто
    }
}
```

Где-то внутри приложения был написан код, который создавал объект нужного класса и вызывал нужный метод. Вот достаточно простое приложение, которое демонстрирует нашу задумку уже в виде готовых интерфейсов и классов.

Класс FinanceInformation

```
/*
 * Класс, который содержит финансовую информацию
 *  По сути этот класс включает некоторый набор полей,
 *  который описывает структуру финансовой информации
 */
package edu.javacourse.reflection;

public class FinanceInformation
{
    // Какие-то важные поля
```

```
}
```

Интерфейс FinanceInfoBuilder

```
/*
 * Интерфейс, который позволяет сделать менее жесткую связь
 * для вызова нужного метода
 */
package edu.javacourse.reflection;

public interface FinanceInfoBuilder
{
    FinanceInformation buildFinacneInformation();
}
```

Класс DbFinanceInfoBuilder

```
/*
 * Реализация интерфейса FinanceInfoBuilder
 * которая работает с информацией из базы данных
 */
package edu.javacourse.reflection;

public class DbFinanceInfoBuilder implements FinanceInfoBuilder
{
    @Override
    public FinanceInformation buildFinacneInformation() {
        System.out.println("Вызов метода для объекта DbFinanceInfoBuilder");
        return new FinanceInformation();
    }
}
```

Класс FinanceInfoBuilderFactory

```
/*
 * Класс для создания экземпляра нужного класса
 */
package edu.javacourse.reflection;
```

```
5
6  public class FinanceInfoBuilderFactory
7  {
8      public static FinanceInfoBuilder getFinanceInfoBuilder() {
9          // Обратите внимание, что здесь мы ВЫНУЖДЕНЫ писать реальный класс
10         // Хорошо, что это мы делаем только в одном месте, ане в каждой
11         // строке, где нам потребуется обращение к FinanceInfoBuilder
12         return new DbFinanceInfoBuilder();
13     }
14 }
```

И наконец вариант вызова — упрощенно

```
1  package edu.javacourse.reflection;
2
3  public class FininceExample
4  {
5      public static void main(String[] args) {
6          // Получаем нужный объект
7          FinanceInfoBuilder fib = FinanceInfoBuilderFactory.getFinanceInfoBuilder();
8          // Вызываем метод
9          FinanceInformation info = fib.buildFinacneInformation();
10         // Вызов getClass() позволяет получить описание класса у объекта
11         System.out.println("Имя класса:" + fib.getClass().getCanonicalName());
12         // Дальше можем делать с полученной информацией все, что захотим
13     }
14 }
```

В общем все достаточно симпатично — есть отдельный класс FinanceInfoBuilderFactory, который нам позволяет локализовать создание конкретной реализации интерфейса FinanceInfoBuilder. И все было хорошо, пока вдруг в связи с новыми веяниями эту информацию теперь надо получать с помощью веб-сервиса. Если вы не знаете, что такое веб-сервис — не страшно. Самое главное — вы уже догадываетесь, что метод и класс будут другими. Совсем другими. Т.е. мы должны написать новый класс

```
1  /*
2   * Реализация интерфейса FinanceInfoBuilder
3   * которая работает с информацией из веб-сервиса
4   */
5  package edu.javacourse.reflection;
6
7  public class WebFinanceInfoBuilder implements FinanceInfoBuilder
```

```
{
    @Override
    public FinanceInformation buildFinacneInformation() {
        System.out.println("Вызов метода для объекта WebFinanceInfoBuilder");
        return new FinanceInformation();
    }
}
```

И теперь (какие мы молодцы) нам надо сделать очень простое изменение внутри FinanceInfoBuilderFactory

```
/*
 * Класс для создания экземпляра нужного класса
 */
package edu.javacourse.reflection;

public class FinanceInfoBuilderFactory
{
    public static FinanceInfoBuilder getFinanceInfoBuilder() {
        // Теперь мы создаем объект другого класса WebFinanceInfoBuilder
        return new WebFinanceInfoBuilder();
    }
}
```

Мы написали новый класс — он работает. Наш класс реализует интерфейс FinanceInfoBuilder. Замечательно. Мы даже внесли изменения в класс FinanceInfoBuilderFactory для создания объекта теперь уже нового класса.
Здесь я использовал шаблон проектирования типа “Фабричный метод”, чтобы уменьшить и упростить изменения, но в любом случае надо брать исходный код, изменять, перекомпилировать и заново устанавливать. Как вы уже можете догадаться, это крайне неудобно. В условиях, когда система должна работать постоянно, не всегда просто переустановить систему и прочая, это совсем не радует.
Даже если учесть, что изменения самого кода в общем не такие уж и большие. Но их надо делать каждый раз, когда нам захочется поменять класс. Например, вернуть старый вариант. В идеале мы можем иметь несколько классов на выбор и устанавливать какой-то из них в определенной ситуации. Но каждый раз надо перекомпилировать, переустанавливать. Как-то корявенько получается.
Что же можно придумать в такой ситуации ? Напрашивается достаточно очевидное решение — надо где-то в виде строки хранить имя класса (например в той же базе данных, текстовом файле или еще в каком-то общедоступном месте) и в момент вызова считывать это имя, создавать объект указанного класса и, ЗНАЯ, ЧТО КЛАСС РЕАЛИЗУЕТ ОПРЕДЕЛЕННЫЙ ИНТЕРФЕЙС, привести этот класс к интерфейсу FinanceInfoBuilder и вызвать метод buildFinacneInformation.

Давайте воспользуемся Reflection API для решения нашей задачи. Имя класса пропишем в файле свойств — мы его разбирали в разделе Коллекции — продолжение

Создадим файл **builder.properties** в каталоге с нашими классами вот с таким содержимым

```
builder.class=edu.javacourse.reflection.WebFinanceInfoBuilder
```

И теперь посмотрим на новый вариант FinanceInfoBuilderFactory. Комментарии по коду посмотрим после

```
/*
 * Класс для создания экземпляра нужного класса
 */
package edu.javacourse.reflection;

import java.util.PropertyResourceBundle;
import java.util.logging.Level;
import java.util.logging.Logger;

public class FinanceInfoBuilderFactory
{
    // Сделаем констуанту для имения нужного нам свойства
    private static final String BUILDER_CLASS = "builder.class";

    public static FinanceInfoBuilder getFinanceInfoBuilder() {
        // Получаем имя класса из файла builder.properties
        PropertyResourceBundle pr = (PropertyResourceBundle)
                PropertyResourceBundle.getBundle("edu.javacourse.reflection.builder");
        String className = pr.getString(BUILDER_CLASS);

        try {
            // Загружаем класс по имени
            Class cl = Class.forName(className);
            // Т.к. наш класс должен имплементировать интерфейс FinanceInfoBuilder
            // то мы можем сделать приведение к интерфейсу
            FinanceInfoBuilder builder = (FinanceInfoBuilder)cl.newInstance();
            return builder;
        } catch (ClassNotFoundException | InstantiationException | IllegalAccessException ex) {
            ex.printStackTrace(System.out);
        }
        return null;
    }
}
```

Первая часть в методе getFinanceInfoBuilder загружает файл свойств и получает свойство «builder.class», в котором мы храним ПОЛНОЕ ИМЯ КЛАССА (включая пакеты). В итоге в переменной className у нас есть строка с именем нужного нам класса.
Вторая часть нашего метода должна решить две задачи:

1. Загрузить класс по имени
2. Создать переменную нужного класса

Первую задачу решает вызов

```
1            Class cl = Class.forName(className);
```

В этой строке есть несколько моментов, на которые надо обратить внимание (всего одна строка, а сколько интересного)
Во-первых, результатом вызова является переменная типа Class (даже звучит необычно класс Class). Но шутки в сторону — класс Class является важной частью Reflection API — именно этот класс позволяет “покопаться” внутри любого класса — посмотреть конструкторы, поля, методы. Позже мы посмотрим, как это можно сделать. А пока просто отметим — класс Class очень-очень важный.
Теперь обратим наши взоры к самому вызову Class.forName(). На данный момент я не хочу сильно углубляться в спецификации и документацию, постараюсь упрощенно описать, что происходит в момент вызова.
Так вот, когда приложение на Java стартует, это совсем не означает, что все классы, которые содержаться в вашем приложении (включая классы в JAR-файлах), будут загружены сразу. Т.е. если внутри вашего приложения нет обращения к классу DbFinanceInfoBuilder, то он даже не загрузиться внутрь JVM. Только в момент обращения к классу в коде, он будет загружаться специальным классом — ClassLoader. Тема ClassLoader любопытная, но оставим это на будущее — можете поискать информацию сами. Это достаточно познавательное чтение.
Второй вариант заставить ClassLoader загрузить нужный класс — сделать это явно через вышеупомянутый вызов Class.forName(). Этот вызов загрузит описание класса (если его еще нет в JVM) и вернет его описатель в виде объекта класса Class. Если класс загружен, то вызов просто вернет описатель. Метод forName имеет более интересный вариант вызова, который регулирует инициализацию класса и даже какой ClassLoader для этого использовать, но пока остановимся на простом варианте.
Что важно — мы указываем имя класса в виде строки. Это не код, который надо было компилировать — это просто строка. И получить ее можно откуда угодно.
Еще раз повторюсь — я упрощенно описал этот момент, но в принципе этого пока достаточно.
Итак, у нас есть описание класса. Второй вызов позволяет нам создать экземпляр (объект) класса

```
1            FinanceInfoBuilder builder = (FinanceInfoBuilder)cl.newInstance();
```

Метод newInstance() позволяет создать объект указанного класса. Но что ВАЖНО отметить — такой вызов возможен только в случае, если класс имеет конструктор БЕЗ ПАРАМЕТРОВ. Если такого конструктора в вашем классе нет, то воспользоваться таким простым вызовом не получится — придется действовать сложнее. Так что иметь конструктор без параметров — это неплохая идея.
Как вы уже возможно догадались, вызов возвращает объект класса Object и нам надо вручную привести его к нужному типу. Т.к. мы полагаемся на “порядочность” нашего класса, которая выражается в поддержке интерфейса FinanceInfoBuilder, то мы к нему и приводим наш объект.
Теперь для смены класса для загрузки финансовых показателей достаточно просто отредактировать файл builder.properties. Ничего больше. Далее следуют бурные продолжительные аплодисменты.

Проект в NetBeans можно загрузить здесь: FinanceExample

Класс Class

Мы посмотрели вариант загрузки класса и создание объекта нужного нам класса. На самом деле такой вариант использования рефлексии весьма распространен. Так что очень рекомендую запомнить это решение. Теперь мы посмотрим, как можно не только создать объект нужного класса, но и как можно обратиться к полям или методам.
Определим простой класс для наших экспериментов

```
1  /*
2   * Класс для демонстрации рефлексии
3   */
4  package edu.javacourse.reflection;
5
6  public class SimpleClass
7  {
8      private String first;
9
10     public String getFirst() {
11         return first;
12     }
13
14     public String simple() {
15         return "Method1";
16     }
17
18     public String concat(String s1, String s2) {
19         return s1 + s2;
20     }
21 }
```

И теперь “поиграем” с нашим классом — обратимся к его полю и методам через Reflection API

```
1  /*
2   * Класс для демонстрации рефлексии
3   */
4  package edu.javacourse.reflection;
5
6  import java.lang.reflect.Field;
7  import java.lang.reflect.Method;
```

```
 8
 9  public class SimpleReflectionExample
10  {
11      public static void main(String[] args) {
12          //Пример обращения к полям и методам класса
13          try {
14              demoReflection();
15          } catch (Exception ex) {
16              ex.printStackTrace(System.out);
17          }
18      }
19
20      private static void demoReflection() throws Exception {
21          // Заружаем описание класса
22          Class example = Class.forName("edu.javacourse.reflection.SimpleClass");
23          SimpleClass sc = (SimpleClass) example.newInstance();
24
25          // Обращение к полю
26          demoReflectionField(example, sc);
27
28          // Обращение к методу
29          demoReflectionMethod(example, sc);
30      }
31
32      private static void demoReflectionField(Class example, SimpleClass sc) throws Exception {
33          // Получить обхект типа Field - обратите внимание, что поле private
34          Field f = example.getDeclaredField("first");
35          // Выставить разрешение для доступа к полю
36          f.setAccessible(true);
37          // Получить значение поля - оно у нас пока NULL
38          String test = (String)f.get(sc);
39          System.out.println("Field before SET:" + sc.getFirst());
40          // Установить значение поля
41          f.set(sc, "Test");
42          System.out.println("Field after SET:" + sc.getFirst());
43      }
44
45      private static void demoReflectionMethod(Class example, SimpleClass sc) throws Exception {
46          // Вызов метода без параметров
47          // Получить обхект типа Method по имени
48          Method method1 = example.getMethod("simple");
49          // Вызвать метод с помощью invoke - передать туда только объект
50          String simple = (String)method1.invoke(sc);
51          System.out.println("Simple:" + simple);
52
53          // Вызов метода с параметрами
54          // Сначала надо определить список параметров - вспоминаем overloading
```

```
55          // У нас это две строки - String
56          Class[] paramTypes = new Class[] {String.class, String.class};
57          // Получить обхект типа Method по имени и по списку параметров
58          Method concat = example.getMethod("concat", paramTypes);
59          // Вызвать метод - передать туда объект и два параметра типа строка
60          String answer = (String)concat.invoke(sc, "1", "2");
61          System.out.println("Concat:" + answer);
62      }
63  }
```

Итак, давайте разбираться. Метод demoReflcetion уже знакомым нам способом создает объект типа SimpleClass. В принципе этот момент был не обязателен и выглядит достаточно натянуто — смысл загружать класс и следующей строкой его же и использовать, но для демонстрации подойдет. Дальше начинаются гораздо более интересные вещи.
Метод demoReflectionField показывает способ обращения к полю (причем к приватному полю). Самое главное — это получение обхекта типа Field по имени, с помощью которого можно уже работать с конкретным полем. Дальше код демонстрирует такие возможности.
Особо хочу отметить вызов setAccessible(true), который позволяет работать с приватным полем.
Метод demoReflectionMethod демонстрирует вариант получения метода по имени и по имени и параметрам — вспоминаем, что такое overloading. Здесь уже используется другой тип — Method — с помощью которого можно вызвать конкретный метод объекта. Дальше я предлагаю вам самим запустить пример а также прочитать код и комментарии к нему.

Проект в NetBeans можно загрузить здесь: SimpleReflection

Аннотации

В версии Java 1.5 появился очень интересный и очень мощный инструмент — аннотации. По сути аннотация — это именованный блок информации, который содержит набор именованных параметров и этот блок можно “прикрепить” к классу, методу, полю и даже параметру в методе. Другими словами — у аннотации есть имя и у нее есть список параметров с именами, которые можно выставить в определенные значения.
И теперь еще раз подумайте — вы можете прикрепить к основным артефактам кода (класс, метод, поле) блок с информацией и ЭТОТ БЛОК ДОСТУПЕН через рефлексию.
И что в этом таког, можете спросить вы ? Дело в том, что теперь есть возможность написать библиотеку (набор классов), которая может обрабатывать классы с определенными аннотациями. Например, именно так работает система ORM (Object Relation Mapping) — система сохранения объектов в базу данных (если честно, то не только так). С появлением аннотаций это теперь очень несложно сделать — вы аннотируете класс, который надо сохранить в базу данных специальным набором аннотаций и все. Дальше библиотека смотрит по аннотациям в какую таблицу и какое поле этого объекта к какую колонку записывается. Там еще можно “навесить” дополнительные условия, связи и много чего еще. Скорость разработки возрастает, написание системы упрощается, становится более лаконичной.
Точно также можно делать EJB, сервлеты. JUnit работает по этому принципу. Системы автоматического создания набора нужных объектов (IoC/DI — Inversion of Control/Dependency Injection) с нужными значениями полей использует аннотации. Веб-сервисы строятся на основе аннотаций, работа с XML и много чего еще.
По сути, библиотека просто говорит вам: “если у твоего класса есть такие-то аннотации, то я смогу произвести над ним нужную тебе работу. Просто напиши нужные аннотации с нужными параметрами”.

Аннотации настолько “вросли” в различные технологии Java, что на сегодня без них работать гораздо сложнее.
Для примера я создал свою аннотацию (хотя прикладной программист чаще всего использует уже готовые) для расширения нашего примера с финансовой информацией.
Вот как выглядит аннотация (это файл .java)

```java
package edu.javacourse.reflection;

import java.lang.annotation.Retention;

// Мы собираемся использовать аннотацию во время исполнения
@Retention(java.lang.annotation.RetentionPolicy.RUNTIME)
@interface FinanceAnnotation
{
    // Объявляем параметр для имени класса со значением по умолчанию
    String financeBuilder() default "edu.javacourse.reflection.DbFinanceInfoBuilder";
}
```

И теперь мы можем через аннотацию установить имя класса

```java
/*
 * Класс для создания экземпляра нужного класса
 */
package edu.javacourse.reflection;

import java.lang.annotation.Annotation;

// Объявление аннотации для класса
@FinanceAnnotation(financeBuilder = "edu.javacourse.reflection.WebFinanceInfoBuilder")
public class FinanceInfoBuilderFactory
{
    public static FinanceInfoBuilder getFinanceInfoBuilder() {
        try {
            // Получаем аннотацию к классу. Т.к. это наш класс, то можно его приводить
            Annotation ann =
                    FinanceInfoBuilderFactory.class.getAnnotation(FinanceAnnotation.class);
            FinanceAnnotation fa = (FinanceAnnotation)ann;

            // Загружаем класс по имени
            Class cl = Class.forName(fa.financeBuilder());
            // Т.к. наш класс должен имплементировать интерфейс FinanceInfoBuilder
            // то мы можем сделать приведение к интерфейсу
```

```
            FinanceInfoBuilder builder = (FinanceInfoBuilder)cl.newInstance();
            return builder;
        } catch (ClassNotFoundException | InstantiationException | IllegalAccessException ex) {
            ex.printStackTrace(System.out);
        }
        return null;
    }
}
```

Конечно же, в нашем случае такое использование аннотации имеет сомнительную эффективность, но здесь важно увидеть принцип — я создал аннотацию и теперь я могу с над аннотированным классом произвести нужные мне действия, используя информацию из этой аннотации.

Проект в NetBeans можно загрузить здесь: FinanceExampleAnnotation

Выводы

Как вы могли видеть, рефлексия является очень мощным инструментом, который повышает гибкость системы. Но за это приходится платить немалую цену — скорость вызова методов (или обращение к полю) через рефлексию значительно замедляет работу системы — в разы и даже в десятки и сотни раз. Будьте очень внимательны при решении использовать рефлексию.
В интернете существует немало интересных статей, посвященных сторонним реализациям рефлексии, возможностям повысить производительность. В некоторых случаях они настолько успешно работают, что по скорости немногим уступают прямым вызовам, но все-таки увлекаться этим не стоит — в конце концов, если настройка вашей системы будет сложнее самой системы, то зачем она такая нужна.
Разумеется, что в данной статье мы не затронули дстаточно большое количество моментов, которые решает Reflection API. Но мне хотелось, чтобы вы увидели основы рефлексии, которые позволят вам понять, что многие “чудесные” возможности Java решаются именно с помощью рефлексии и основные моменты для этого мы рассмотрели. Надеюсь, что дальше вам будет гораздо проще находить тонкости при работе с рефлексией.

И теперь нас ждет следующая статья: Установка СУБД PostgreSQL.

4 comments to *Reflection — основы*

- 
  Февраль 6, 2018 at 18:22
  Максим says:

  Что-то строку «Class[] paramTypes = new Class[] {String.class, String.class};» не понять, хоть Вы и оставили там комментарий. Если разъяснения слишком сложные, будьте добры, дайте ссылку на соответствующие материалы.

Reply

- 

  Февраль 7, 2018 at 09:21
  admin says:

  Вам надо найти метод с определенным именем и определенным набором параметров (помните про overloading — можно иметь несколько методов с одинаковыми именами, но с разным набором входных параметров).
  Так вот указанная строка

  Class[] paramTypes = new Class[] {String.class, String.class};

  определяет массив, члены которого являются списком входных параметров. В данном случае мы ищем метод с именем «concat», у которого два входных параметра и оба имеют тип String.

  Reply

- 

  Август 18, 2018 at 18:18
  Владимир says:

  Здравствуйте. Не работает Class.ForName в Idea, кидает ClassNotFoundException. Такая же проблема была, когда пытался сделать сервлет, обращающийся к БД, но тогда решил отложить этот вопрос. В интернете не смог найти внятного и доступного ответа(Как понимаю, проблема распространённая, что делать в такой ситуации?

Reply

- 

  Август 19, 2018 at 07:17
  admin says:

  Проверить, где находится загружаемый класс. Его нельзя просто так загрузить из файла откуда угодно. Он должен быть доступен в рамках указанного CLASSPATH. Т.е. он либо должен быть в проекте, либо в JAR-файле, который указан в проекте.
  Очень часто такая ошибка возникает, когда вы указываете один JAR, которого достаточно для компиляции, а в момент запуска нужен еще один JAR с дополнительными классами. Но как я понимаю, у Вас не такая проблема.

Reply

Leave a reply

Comment

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code class="" title="" data-url=""> <del datetime=""> <em> <i> <q cite=""> <s> <strike> <strong> <pre class="" title="" data-url=""> <span class="" title="" data-url="">`

Имя *

E-mail *

Сайт

7 − = два

Add comment