одина

Базы данных на Java — введение

Мы приступаем к одному из очень важных разделов программирования на Java — работа с базами данных. Данные являются наверно наиглавнейшей составляющей программирования и вопрос их хранения крайне актуален. Не буду больше говорить о важности этого вопроса — тут можно писать много-много-много разных интересных слов.

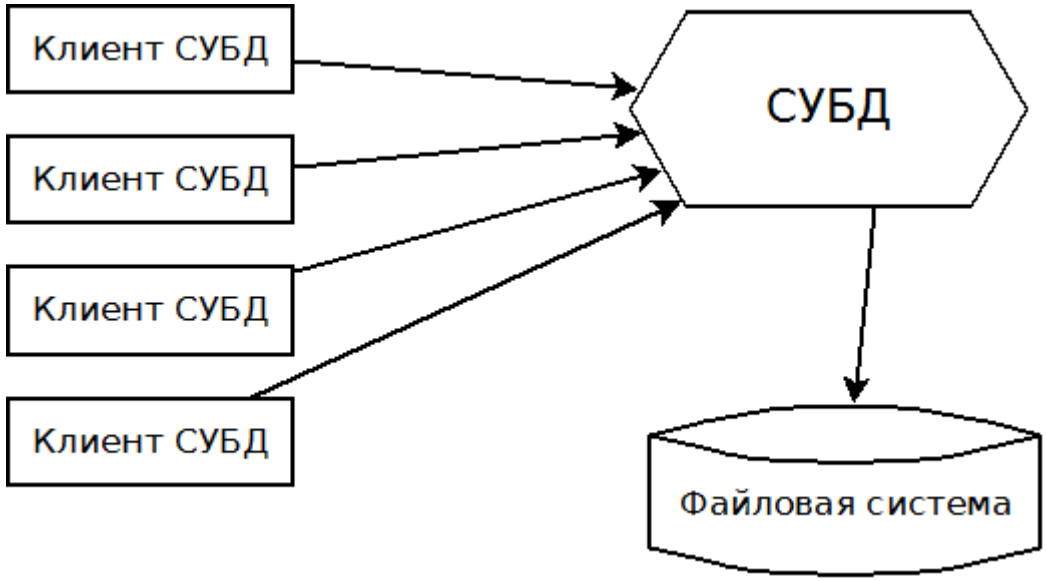
Сервер баз данных

Сама идея сервера баз данных и СУБД в виде отдельной программы появилось по совершенно очевидным причинам. Базы данных мгновенно стали МНОГОПОЛЬЗОВАТЕЛЬСКИМИ. Данные нужны всем и возможность одновременного доступа к ним является очевидной. Проблема базы данных в виде обычного файла заключается в том, что к этому файлу будет обращаться сразу много программ, каждая из которых захочет внести изменения или получить данные. Организовать такой доступ на уровне файловой системы — по сути, невыполнимая задача.
Во-первых — файл должен быть доступен всем пользователям, что требует перекачку данных по сети и хранение этого файла где-то на сетевом диске. Большие объемы данных по сети (пусть даже с высокой скоростью) — кроме слова “отвратительно” у меня ничего не приходит на ум.

Во-вторых — попытка одновременной записи в файл несколькими программами обречена на провал. Для организации такого доступа обычной файловой системы явно не достаточно.
В-третьих — организация прав доступа к тем или иным данным тоже становится непосильной задачей.
В-четвертых — надо “разруливать” конфликты при одновременном доступе к одним и тем же данным.
После небольшого анализа, кроме этих вопросов, можно увидеть еще немалое количество проблем, которые надо решить при мультипользовательском доступе к данным.
В итоге было принято (и реализовано) вполне здравое решение — написать специальную программу, которая имеет несколько названий — Система Управления Базами Данных (СУБД), сервер баз данных и т.д. Я буду называть ее СУБД.
Суть и цель этой программы — организовать централизованный доступ к данным. Т.е. все запросы на получение или изменение данных от клиентских приложений (клинетов) посылаются (обычно по сети и по протоколу TCP/IP) именно в эту программу. И уже эта программа будет заниматься всеми вышеупомянутыми проблемами:

1. СУБД будет иметь некоторый набор команд, который позволит записывать и получать данные
2. СУБД будет сама работать с файловой системой (нередко у нее бывает своя собственная файловая система для скорости)
3. СУБД предоставит механизмы разграничения доступа к разным данным
4. СУБД будет решать задачи одновременного доступа к данным

В итоге мы получаем достаточно ясную архитектуру — есть СУБД, которая сосредоточена на работе с данными и есть клиенты, которые могут посылать запросы к СУБД.



При работе с СУБД клиенты должны решить достаточно четкие задачи:

1. Клиент должен соединиться с СУБД. Как я уже упоминал, чаще всего для общения используется сетевой протокол TCP/IP. В момент подключения клиент также передает свой логин/пароль, чтобы СУБД могла его идентифицировать и в дальнейшем позволить (или не позволить) производить те или иные действия над данными
2. Клиент может посылать команды для изменения/получения данных в СУБД
3. Данные внутри СУБД хранятся в определенных структурах и к этим структурам можно обратиться через команды

SQL базы данных

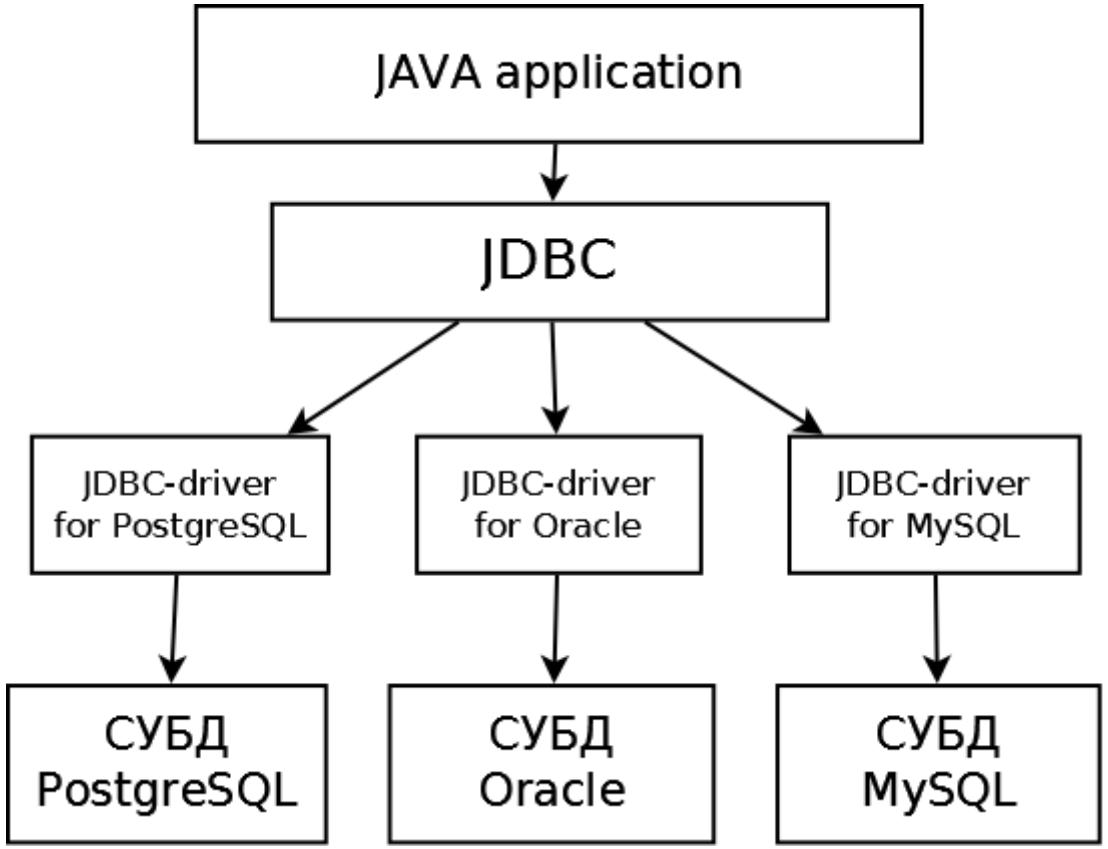
Могу предположить,что вышеупомянутые задачи и породили именно SQL-базы данных. В них есть удобные и понятные структуры для хранения данных — таблицы. Эти таблицы можно связывать в виде отношений и тем самым дается возможность хранить достаточно сложно организованные данные. Был придуман специальный язык — SQL (Structured Query Language — структурированный язык запросов). Этот язык хоть и имеет всего 4 команды для манипулирования данными, позволяет создавать очень сложные и заковыристые запросы.
На сегодняшний день SQL-базы данных являются самыми распространенными. В последние годы наметилась тенденция к использованию баз данных, основанные на других способах хранения и обработки данных, но пока их применение достаточно узконаправлено, хотя в некоторых случаях они действительно помогают решать важные задачи более эффективно, но все-таки пока SQL — самое главное направление баз данных. Почему я про это упоминаю ? Потому, что все наше знакомство с технологией работы с базами данных из Java будет сконцентрировано на SQL базах данных. С основными командами SQL вы можете познакомиться в различных учебниках. Их сейчас достаточно много и в большинстве своем они вполне понятны.
Возможно, что я тоже когда-нибудь внесу свою лепту в рассказы про SQL, но в данном разделе предполагается, что вы уже знакомы с основными идеями построения реляционных баз данных и с самим языком SQL.

JDBC — Java Database Connectivity — архитектура

Если попробовать определить JDBC простыми словами, то JDBC представляет собой описание интерфейсов и некоторых классов, которые позволяют работать с базами данных из Java. Еще раз: JDBC — это набор интерфейсов (и классов), которые позволяют работать с базами данных.
И вот с этого момента я попробую написать более сложное и в тоже время более четкое описание архитектуры JDBC. Главным принципом архитектуры является унифицированный (универсальный, стандартный) способ общения с разными базами данных. Т.е. с точки зрения приложения на Java общение с Oracle или PostgreSQL не должно отличаться. По возможности совсем не должно отличаться.
Сами SQL-запросы могут отличаться за счет разного набора функций для дат, строк и других. Но это уже строка запроса другая, а алгоритм и набор команд для доставки запроса на SQL-сервер и получение данных от SQL-сервера отличаться не должны.

Наше приложение не должно думать над тем, с какой базе оно работает — все базы должны выглядеть одинаково. Но при всем желании внутреннее устройство передачи данных для разных СУБД разное. Правила передачи байтов для Oracle отличается от правил передачи байтов для MySQL и PostgreSQL. В итоге имеем — с одной стороны все выглядят одинаково, но с другой реализации будут разные. Ничего не приходит в голову ?
Еще раз — разные реализации, но одинаковый набор функциональности.

Думаю, что вы уже догадались — типичный полиморфизм через интерфейсы. Именно на этом и строится архитектура JDBC. Смотрим рисунок.



Как следует из рисунка, приложение работает с абстракцией JDBC в виде набора интерфейсов. А вот реализация для каждого типа СУБД используется своя. Эта реализация называется “JDBC-драйвер”. Для каждого типа СУБД используется свой JDBC-драйвер — для Oracle свой, для MySQL — свой. Как приложение выбирает, какой надо использовать, мы увидим чуть позже.
Что важно понять сейчас — система JDBC позволяет загрузить JDBC-драйвер для конкретной СУБД и единообразно использовать компоненты этого драйвера за счет того, что мы к этим компонентам обращаемся не напрямую, а через интерфейсы.
Т.е. наше приложение в принципе не различает, обращается оно к Oracle или PostgreSQL — все обращения идут через стандартные интерфейсы, за которыми “прячется” реализация.

Пока я предлагаю отметить несколько важных интерфейсов, которые мы будем рассматривать позже, но мне бы хотелось, чтобы у вас этот список уже был, чтобы вы могли по мере прочтения отмечать — “да, вот он важный интерфейс/класс и я теперь знаю, куда он встраивается”. Вот они:

1. **java.sql.DriverManager**
2. **java.sql.Driver**
3. **java.sql.Connection**
4. **java.sql.Statement**

5. **java.sql.PreparedStatement**
6. **java.sql.CallableStatement**
7. **java.sql.ResultSet**

Теперь давайте рассмотрим несложный пример и поймем, как работает JDBC.

JDBC — пример соединения и простого вызова

Попробуем посмотреть на несложном примере, как используется JDBC-драйвер. В нем же мы познакомимся с некоторыми важными интерфейсами и классами.

Предварительно нам необходимо загрузить JDBC-драйвер для PostgreSQL. На данный момент это можно сделать со страницы PostgreSQL JDBC Download

Если вы не нашли эту страницу, то просто наберите в поисковике “PostgreSQL JDBC download” и в первых же строках найдете нужную страницу.
Т.к. я пишу эти статьи для JDK 1.7 и 1.8, то я выбрал строку “JDBC41 Postgresql Driver, Version 9.4-1208” — может через пару-тройку лет это будет уже не так.
Если вы выполнили SQL-скрипт из раздела Установка PostgreSQL, который создавал таблицу JC_CONTACT и вставил туда пару строк, то эта программа позволит вам “вытащить” эти данные и показать их на экране. Это конечно же очень простая программа, но на ней мы сможем посмотреть очень важные моменты. Итак, вот код:

```
1  package edu.javacourse.database;
2
3  import java.sql.Connection;
4  import java.sql.DriverManager;
5  import java.sql.ResultSet;
6  import java.sql.Statement;
7
8  public class SimpleDb
9  {
10     public static void main(String[] args) {
11         SimpleDb m = new SimpleDb();
12         m.testDatabase();
13     }
14
15     private void testDatabase() {
16         try {
17             Class.forName("org.postgresql.Driver");
18             String url = "jdbc:postgresql://localhost:5432/contactdb";
19             String login = "postgres";
20             String password = "postgres";
21             Connection con = DriverManager.getConnection(url, login, password);
```

```
            Connection con = DriverManager.getConnection(url, login, password);
            try {
                Statement stmt = con.createStatement();
                ResultSet rs = stmt.executeQuery("SELECT * FROM JC_CONTACT");
                while (rs.next()) {
                    String str = rs.getString("contact_id") + ":" + rs.getString(2);
                    System.out.println("Contact:" + str);
                }
                rs.close();
                stmt.close();
            } finally {
                con.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Для запуска этой програмы необходимо подключить JDBC-драйвер для PostgreSQL. Прочитайте раздел Что такое JAR-файлы для того, чтобы подключить нужный JAR с JDBC-драйвером к проекту в NetBeans.
Для запуска нашей программы из командной строки достаточно собрать этот код (причем здесь не надо подключать JAR на этапе компиляции — только на момент запуска).
Итак, команда для сборки:

```
javac edu/javacourse/database/SimpleDb.java
```

И теперь команда для запуска:

```
java -cp .;postgresql-9.4.1208.jre7.jar edu.javacourse.database.SimpleDb
```

Для запуска проекта в NetBeans предлагаю вам самостоятельно разобраться, как подключить JAR-файл — пример этого указан в статье Что такое JAR-файлы

Начнем разбор нашей программы с самого начала. Итак, в чем же заключается набор вызовов для создания соединения с базой

```
            Class.forName("org.postgresql.Driver");
            String url = "jdbc:postgresql://localhost:5432/contactdb";
```

```
String login = "postgres";
String password = "postgres";
Connection con con = DriverManager.getConnection(url, login, password);
```

Вызов **Class.forName()** мы уже встречали, когда разговаривали о рефлексии. Если вы этого не сделали — обязательно прочитайте, иначе многое будет непонятно. Так вот наш вызов загружает один из ключевых классов JDBC, который реализует очень важный интерфейс **java.sql.Driver**. Почему этот класс так важен, мы разберем чуть ниже.
Следующим важным вызовом является **DriverManager.getConnection(url, login, password);**.
Думаю, что параметры **login** и **password** достаточно оччевидны — это логин и пароль для подключения к СУБД. А вот первый параметр — **url** надо рассмотреть подробно.
Параметр **url** является строкой и я люблю его разбивать на две части. Первая часть **jdbc:postgresql:** позволяет идентифицировать, к какому типу СУБД вы подключаетесь — Oracle, MySQL, PostgreSQL, IBM DB2, MS SQL Server. В нашем случае тип базы данных — PostgreSQL.
Вторая часть — **//localhost:5432/contactdb** — определяет конкретный экземпляр выбранной базы данных. Т.е. если первая часть **url** указывает, что мы хотим работать с PostgreSQL, то вторая часть указывает на каком хосте и на каком порту (опять вспоминаем основы TCP/IP) работает конкретный экземпляр PostgreSQL. Еще раз — первая часть пределяет только тип, вторая часть — параметры оединения с конкретным экземпляром СУБД.
Как вы можете видеть, вторая часть включает помимо IP-адреса и порта (localhost:3306) включает имя базы данных, с которой вы будете соединяться.

И вот теперь возвращаемся к интерфейсу **java.sql.Driver**. Достаточно очевидно, что сложное приложение на Java может работать с несколькими типами СУБД и одновременно в приложнении участвуют несколько JDBC-драйверов для разных типов СУБД. Так как же класс **DriverManager** определяет, какой тип СУБД вы собираетесь использовать ?
Придется нам вернуться к моменту загрузки класса — **Class.forName()**. Большинство классов в момент своей загрузки выполняют очень важный шаг — они РЕГИСТРИРУЮТСЯ у класса **DriverManager**. Как они это делают ? Посмотрите документацию на класс **DriverManager** — например здесь: DriverManager
Среди методов мы можете найти этот: **registerDriver(Driver driver)**. Причем метод статический и создавать экземпляр **DriverManager** не надо.
Таким образом драйвер под конкретный тип СУБД регистрируется у **DriverManager**. У этого класса (можно глянуть в исходники) создается список драйверов, каждый из которых реализует интерфейс **java.sql.Driver**. Что же происходить дальше ? Зайдем в документацию java.sql.Driver. Там есть два очень интересных метода:

1. boolean acceptsURL(String url)
2. Connection connect(String url, Properties info)

Первый метод как раз и позволяет классу **DriverManager** пройти по всему списку зарегистрированных у него драйверов и у каждого спросить — “ты умеешь работать с этим URL”. Отметим, что драйвер под конкретный тип СУБД работает с уникальным набором — MySQL принимает строку “jdbc:mysql:”, PostgreSQL — “jdbc:postgresql:” и т.д. Т.е. первая часть параметра **url**, о которой мы говорили немного раньше, как раз и позволяет классу **DriverManager** выбрать драйвер для определенного типа СУБД. Первый шаг сделан — мы выбрали нужный драйвер.
И вот тут приходит очередь второго метода — именно он позволяет создать соединение — возвращает экземпляр класса, который реализует еще один важный интерфейс — **java.sql.Connection**. Второй метод использует вторую часть **url** с адресом, портом и именем базы, а также используется логин и пароль. Снова обращаю ваше внимание на тот факт, что реальный класс будет какой-то специальный, под конкретный тип СУБД, но он обязательно должен реализовать интерфейс **java.sql.Connection**.

java.sql.Connection — это реальное соединение с конкретным экземпляром СУБД определенного типа. Наше соединение готово. Можем продолжать. Следущий фрагмент кода уже будет проще:

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM JC_CONTACT");
```

Первая строка создает еще один важный элемент — запрос, который реализует интерфейс **java.sql.Statement**. Кроме этого интерфейса используются тажке **java.sql.PreparedStatement** и **java.sql.CallableStatement**, но о них мы поговорим несколько позже.
Что здесь важно отметить — создание запроса делается через обращение к методу объекта **java.sql.Connection — createStatement**. И опять обращаю ваше внмание, что каждый производитель СУБД пишет свою реализацию всех интерфейсов.
Т.к. реализация **java.sql.Connection** будет под определенный тип СУБД, то и реализация **java.sql.Statement** тоже будет под определенный тип СУБД.
В качестве домашного задания — попробуйте через рефлексию узнать настоящие имена этих класов.
Вторая строка с помощью объекта-запроса **java.sql.Statement** делает запрос в таблицу **st_student** и получает еще один важный элемент — объект **java.sql.ResultSet**.
После получения данных в виде объекта **ResultSet**, мы можем через его методы “пробежать” по всему набору данных (это очень похоже на итераторы в коллекциях) и выбрать поля из этого набора.

```
while (rs.next()) {
    String str = rs.getString("contact_id") + ":" + rs.getString(2);
    System.out.println("Contact:" + str);
}
```

Как видите, все достаточно несложно. Упрощенно **ResultSet** можно рассматривать, как указатель на строку в таблице. Метод **rs.next()** делает попытку передвинуться на следущую запись. В случае успеха он возвращает **true** и передвигает указатель на следущую строку. Если строки закончились (или их не было вообще), возвращается **false**.
Когда мы передвинулись на следущую строку, то с помощью набора методов можно получить значения колонок в строке — мы использовали метод **getString()** в двух вариантах — один находит колонку по имени, второй — по индексу. Учтите, что номера колонок начинаются с 1, а не с 0, как это делается в массивах и коллекциях. Кроме метода **getString()** для получения строк, **ResultSet** имеет методы для получения чисел (цеых и вещественных), дат и много чего еще.
И наконец обратите внимание, что я вызываю у всех объектов метод **close()**. Особенно важным (я бы даже сказал критичным) является закрытие **Connection**. Закрытие **Statement** тоже является достаточно хорошим решением, но не настолько критичным, В этом случае вы просто быстрее освобождаете память от ресурсов, которые создавались при запросе. Учтите, что **Statement** закрывается автоматически при уничтожении объекта.
Что же касается **ResultSet**, то он автоматически закрывается в момент закрытия **Statement**.
Этот момент весьма важен, но пока я не буду останавливаться на тонкостях — просто примите это как важную необходимость.
Также советую обратить внимание на способ построения обработки исключений. Сначала я создаю коннект во внешнем блоке **try … catch** и потом

уже во внутреннем блоке **try … catch** выполняю запрос и получаю из результата данные. В этом же блоке в разделе **finally** происходит закрытие соединения.
Такое построение дает уверенность, что вне зависимости от результата выполнения запроса и получения данных, соединение будет обязательно закрыто.
Т.к. при работе приложения исключения при выполнении запроса не должны быть частыми, то “незакрытие” **Statement** не должно повлечь каких-либо осложнений. Хотя кто-то может со мной не согласиться и решительно скажет, что надо закрывать все — вполне допускаю такой вариант.

Исходный код проекта на NetBeans вы можете закачать здесь — SimpleDb
Я поместил в архив JDBC-драйвер для PostgreSQL и ссылка на него из проекта относительная, так что по идее все должно работать сразу. Если не будет, то это значит — вам придется приложить некоторое количество усилий. Может это и хорошо — лучше станете понимать материал.
Итак, мы с вами сделали первый шаг на пути изучения JDBC — в качестве самостоятельной работы попробуйте создать свою базу данных, таблицу, заполнить ее данными и сделать запрос из приложения на Java.

И теперь нас ждет следующая статья: Возможности JDBC — второй этап

12 comments to *Базы данных на Java — первые шаги*

- 
  Октябрь 25, 2017 at 23:01
  Андрей says:

  Очень интересная статья, но мало теории и много практики, тем не менее, спасибо за предоставленную информацию

  Reply
- 
  Октябрь 28, 2017 at 16:23
  Ренат says:

  Крутая статья. Спасибо!-)

  Reply

- 
  Ноябрь 21, 2017 at 22:55
  Игорь says:

  Ошибка в слове «одинакого»!Неужели так сложно чекать в ворде?

  Reply

  - 
    Ноябрь 22, 2017 at 09:47
    admin says:

    Спасибо за замечание — исправил. Кстати — у меня нет ворда.

    Reply

- 
  Ноябрь 21, 2017 at 22:56
  Игорь says:

  Кстати статья помогла спс

  Reply

- 
  Февраль 4, 2018 at 23:09
  Максим says:

  >>Итак, в чем же заключается yf,jh вызовов для создания соединения с базой набор

  Reply

Февраль 5, 2018 at 09:42
admin says:

Спасибо — исправил.

Reply

Март 31, 2018 at 05:25
ezd says:

Спасибо, было полезно.

1) В примере кода про postgre, а в текст mysql. Понимаю что разницы нет, но несколько сбивает с толку.

2) Внешние ссылки кривые, там в урлах лишние кавычки href=»»smth»» и одна ссылка вообще не ссылка DriverManager 🙂

Reply

Март 31, 2018 at 13:33
admin says:

Спасибо за замечание, исправил.

Reply

Июль 2, 2018 at 20:53
Артур says:

Просто великолепная статья. Для меня, как для новичка в web-программировании данная статья оказалась понятной и очень практичной. Спасибо за проделанную работу.

Reply

- 
  Август 31, 2018 at 14:07
  andy says:

  Class.forName(«org.postgresql.Driver»); — можно не нужно указывать, учитывая, что в 2018 году навряд ли будут использовать старую версию джавы.

  Reply

  - 
    Сентябрь 1, 2018 at 20:35
    admin says:

    Такое решение существует уже почти 25 лет и другого пока не предложено.

    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

6 × = 42

Add comment