JDBC

JDBC (Java DataBase Connectivity) — платформенно-независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД
JDBC основан на концепции так называемых драйверов, позволяющих получать соединение с базой данных по специально описанному URL. Драйверы могут загружаться динамически (во время работы программы). Загрузившись, драйвер сам регистрирует себя и вызывается автоматически, когда программа требует URL, содержащий протокол, за который драйвер отвечает.
С помощью JDBC легко отсылать SQL-запросы почти ко всем реляционным БД. Другими словами, использование JDBC API избавляет от необходимости для каждой СУБД (MS SQL, Oracle и т.д.) писать свое приложение. Достаточно написать одну единственную программу, использующую JDBC API, и эта программа сможет отсылать SQL-запросы к требуемой БД. Кроме того, это приложение будет переносимо на различные платформы.

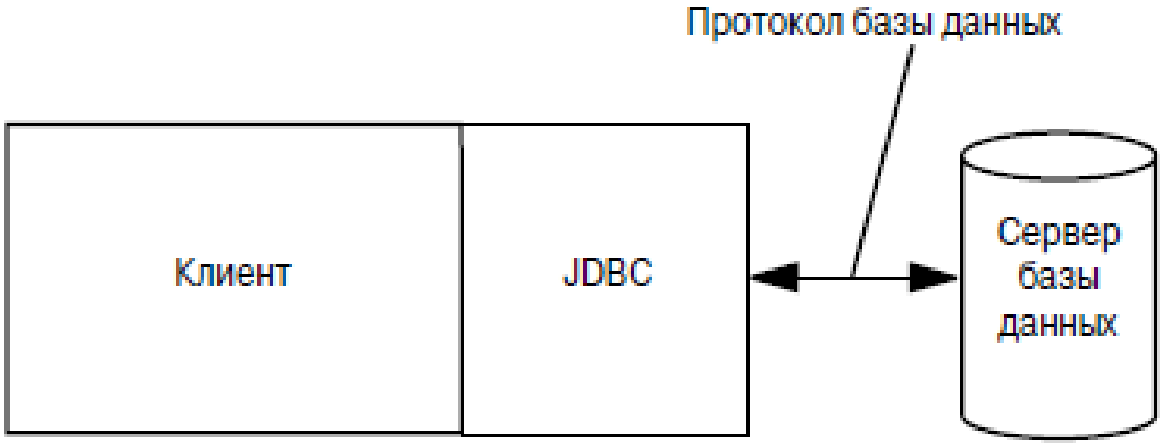
JDBC

Типы JDBCдрайверов

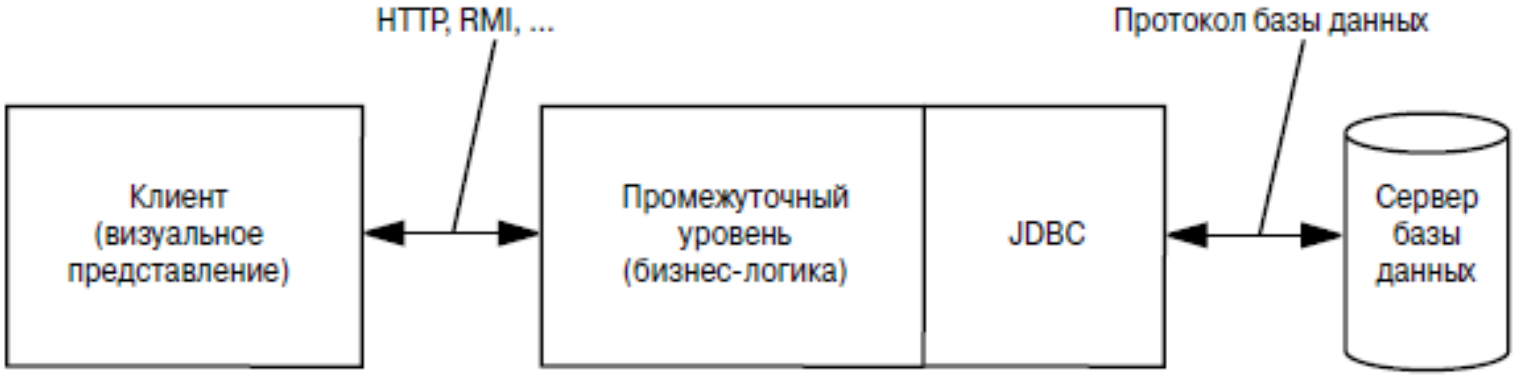
Каждый JDBCдрайвер принадлежит одному из перечисленных ниже *типов.*

- *Драйвер типа 1. Транслирует JDBC в ODBC и для взаимодействия с базой данных* использует драйвер ODBC.
- *Драйвер типа 2. Создается преимущественно на языке Java и частично на собст*венном языке программирования, который используется для взаимодействия с клиентским API базы данных. Для использования такого драйвера нужно помимо библиотеки Java установить специфический для данной платформы код.
- *Драйвер типа 3. Создается только на основе библиотеки Java, в которой испол*ьзуется независимый от базы данных протокол взаимодействия сервера и базы. Этот протокол позволяет транслировать запросы в соответствии со специфи кой конкретной базы. Если код, зависящий от базы данных, находится только на сервере, доставка программ существенно упрощается.
- *Драйвер типа 4. Представляет собой библиотеку Java, которая транслирует* JDBC запросы непосредственно в протокол конкретной базы данных.

JDBC



Двухзвенная архитектура



Трехзвенная архитектура

JDBC Соединение с базой

Для установления соединения с базой данных необходимо указать источник данных и, возможно, некоторые дополнительные параметры. Например, сетевым драйверам нужен номер порта, а драйверам ODBC могут потребоваться различные атрибуты.
В JDBC используется синтаксис описания источника данных, подобный обычным URL.

```
String url = "jdbc:oracle:thin:@localhost:1522:NCEDU";
```

```
URL для Oracle:
jdbc:oracle:<drivertype>:@<database>
```

Thin Driver –Драйвер клиентских приложений не требующий установки Oracle на машину клиента.
OCI толстый для локальной работы извне. На клиенте должен быть установлен клиент Oracle

JDBC Соединение с базой

Далее необходимо получить объект класса Connection

```
Connection cn =
      DriverManager.getConnection (url,”user",“password");
```

Предварительно нужно зарегистрировать класс драйверов. Это можно сделать несколькими способами

1. Class.forName("oracle.jdbc.OracleDriver");
2. System.setProperty("jdbc.drivers", "oracle.jdbc.OracleDriver") ;

После регистрации драйверов можно установить соединение с базой данных. Диспетчер перебирает все зарегистрированные драйверы, пытаясь найти тот, который соответствует подпротоколу, указанному в URL базы данных.

Properties

Файл Properties это файл состоящий из пар : ключ=значение

```
# Database configuration
Database.Driver=sun.jdbc.odbc.JdbcOdbcDriver
Database.DataURL=jdbc:odbc:MyDatabase
Database.Prop.user=user
Database.Prop.password=password
```

В Java есть готовый класс для чтения/записи таких файлов (java.util.Properties).

```
FileInputStream dbp =
      new FileInputStream("DBConnection.properties");
Properties props = new Properties();
props.load(dbp);

String url = props.getProperty("Database.DataURL");
String driver = props.getProperty("Database.Driver");
String user = props.getProperty("Database.User");
String pass = props.getProperty("Database.Password");
```

Statement

Для выполнения SQLкоманды нужно создать объект Statement. Для этой цели используется объект Connection, который можно получить, вызвав метод Driver Manager.getConnection().

```
Statement stat = conn.createStatement();
```

Далее используя один из методова Exec… можно выполнять SQL запросы.

```
 String SQLInsert ="insert into emp values(15,\'sefsdf\')";
stat.executeUpdate(SQLInsert);
```

Метод executeUpdate() может применяться для выполнения команд INSERT, UPDATE и DELETE, а также команд определения данных, в частности CREATE TABLE и DROP TABLE.

Для выполнения команды SELECT нужно использовать другой метод, а именно executeQuery().

Метод executeQuery() возвращает объект ResultSet, который можно использовать для построчного просмотра результатов.

```
ResultSet rs =
        st.executeQuery ("SELECT empno, empname FROM emp");
```

Statement

Для анализа результатов можно использовать цикл:

```
while (rs.next()) {
   System.out.println("Number=" +
        rs.getString(1) + " " +"Name=" + rs.getString(2));
}
```

Для каждого типа данных языка Java предусмотрен отдельный метод извлечения информации, например getString() и getDouble(). Получать данные можно либо по номеру столбца, либо по его имени.

Нумерация начинается с 1

JDBC простой пример

```
String url = "jdbc:oracle:oci:@localhost:1522:NCEDU";
Class.forName("oracle.jdbc.OracleDriver") ;
Connection cn =
       DriverManager.getConnection (url,"maickel",“123");
Statement st = cn.createStatement();
ResultSet rs =
      st.executeQuery ("SELECT empno, empname FROM emp");
while (rs.next()) {
  System.out.println("Number=" +
       rs.getString(1) + " " +"Name=" + rs.getString(2));
}
rs.close();
st.close();
cn.close();
```

Пример, в общем, работает но это плохой пример!!!

JDBC простой пример

```
try (
     Connection cn =
        DriverManager.getConnection (url,user,pass);
      Statement st = cn.createStatement()
     ) {
 ResultSet rs =
       st.executeQuery ("SELECT empno, empname FROM emp");
while (rs.next()) {
   System.out.println("Number=" + rs.getString(1) + " "
+"Name=" + rs.getString(2));
         }
rs.close();
}
```

Работа с соединениями

Каждый объект Connection может создать один или несколько объектов Statement. Один и тот же объект Statement можно использовать для нескольких не связанных между собой команд и запросов. Однако для такого объекта допускается наличие не более одного открытого набора результатов.

```
ResultSet rs =
       st.executeQuery("SELECT empno, empname FROM emp");
ResultSet rs1 =
       st.executeQuery("select id,fio from students");
while (rs.next()) {
   System.out.println("Number=" +
        rs.getString(1) + " " +"Name=" + rs.getString(2));
}
```

Если требуется выполнить несколько команд с одновременным анализом предоставленных ими результатов, понадобится несколько объектов Statement.

Работа с соединениями

```
System.out.println(cn.getMetaData().getMaxStatements());
```

После окончания работы с Connection, ResultSet, Statement их надо закрывать, дабы освободить ресурсы ими занимаемые.
Метод close() класса Statement автоматически закрывает связанные с ним наборы результатов (если, конечно, эти наборы открыты). Аналогично метод close() класса Connection закрывает все объекты Statement для этого соединения.

PreparedStatement

```
String studentNumber = sc.nextLine();
String prepSQL =
     "select fio from students where id="+studentNumber;
ResultSet rs = st.executeQuery(prepSQL);
while (rs.next()){
  System.out.println(rs.getString(1));
}
```

Для удобной работы с такими запросами существует класс PreparedStatement

```
String prepSQL = "select fio from students where id=?";
PreparedStatement pst = cn.prepareStatement(prepSQL);
pst.setString(1, studentNumber);
ResultSet rs = pst.executeQuery();
```

PreparedStatement

Плюсом использования является:

1. Удобство использования.
2. Код более понятен для восприятия
3. Возможно, это предотвратит повторное создание плана запроса
4. Снижает вероятность атаки с помощью SQL-инъекций.

В запросе все параметры обозначаются знаком ?

```
String prepSQL = "select fio from students where id=?";
```

Создание объекта происходит при помощи вызова метода `prepareStatement` объекта `Connection`

Перед выполнением запроса необходимо установить все значения параметров при помощи вызовов методов `SetXXX(номер параметра, значение)`.

Номера параметров идут в том порядке, в котором они встречаются в запросе.

ResultSet: навигация

По умолчанию, ResultSet позволяет осуществлять навигацию только вперед при помощи метода next().

```
PreparedStatement pst =
      cn.prepareStatement(prepSQL);
ResultSet rs = pst.executeQuery();
rs.previous();//Ошибка
```

Если есть необходимость произвольной навигации по набору данных, то необходимо получить Statement особым образом

```
PreparedStatement pst = cn.prepareStatement(prepSQL,
ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
Или
Statement stm
=cn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
```

ResultSet: навигация

```
Statement stat = conn.createStatement(type, concurrency);
```

Для предварительно подготовленного запроса нужно использовать следующий вызов:

```
PreparedStatement stat = conn.prepareStatement(command,
type, concurrency);
```

Первый параметр, указывающий на желательность «прокручиваемости» ResultSet, может принимать одно из трех допустимых значений:

ResultSet.TYPE_FORWARD_ONLY: это значение по умолчанию.

ResultSet.TYPE_SCROLL_INSENSITIVE: такой ResultSet допускает итерации назад и вперед, но если данные в базе данных изменятся, ResultSet не отразит этого.

ResultSet.TYPE_SCROLL_SENSITIVE: этот тип ResultSet не только допускает двунаправленные итерации, но и создает «живое» представление данных в базе данных по мере их изменения.

ResultSet: навигация

Для обратной прокрутки ResultSet, полученного посредством выражения Statement, достаточно вызвать функцию previous(), которая направлена назад, а не вперед, как next(). Или же можно вызвать first(), чтобы вернуться в начало набора ResultSet, или last() для перехода в конец.

Также могут быть полезны методы relative() и absolute(): первый перемещает курсор на указанное количество строк (вперед, если значение положительно, или назад, если оно отрицательно), а второй - к указанной строке ResultSet, где бы он ни находился в данный момент. Конечно, номер текущей строки, можно получить с помощью метода getRow().

ResultSet: Обновление

JDBC поддерживает не только двунаправленную прокрутку, но и редактирование наборов ResultSet. Это означает, что вместо того чтобы создавать новый оператор SQL для изменения значений, хранящихся в базе данных, можно просто изменить значение внутри ResultSet, и оно автоматически отразится в соответствующем столбце нужной строки базы данных.

Для этого вторым параметром нужно выставить значение

`ResultSet.CONCUR_UPDATEABLE`

Если драйвер поддерживает обновляемые курсоры то любое заданное значение в ResultSet можно обновить, перейдя в нужную строку и вызвав один из методов update...() Однако это приведет лишь к обновлению значения в наборе ResultSet. Чтобы передать его в базу данных, вызовите updateRow(). cancelRowUpdates() отменит все ожидающие обновления.

ResultSet: Обновление

```
Statement stm =
      cn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
      ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stm.executeQuery(
                        "select id,fio from students");
while (rs.next()){
      studentNumber = sc.nextLine();
      rs.updateString(2, studentNumber);
      rs.updateRow();
}
```

ResultSet: Обновление

JDBC 2.0 поддерживает не только обновления. Чтобы добавить новую строку без создания нового объекта Statement и выполнения оператора INSERT, просто вызовите метод moveToInsertRow(), метод update...() для каждого столбца, и, наконец, метод insertRow(). Если значение для столбца не указано, предполагается SQL NULL. По окончании вставки применяется метод moveToCurrentRow() для перемещения курсора назад в позицию, которую он занимал до вызова метода moveToInsertRow().

```
rs.moveToInsertRow();
rs.updateInt(1,55);
rs.updateString(2, "Ivanov");
rs.insertRow();
```

Если ResultSet поддерживает обновление строки, он должен поддерживать и ее удаление посредством вызова метода deleteRow().

ResultSet: Обновление

Даже если СУБД поддерживает работу во всех описанных режимах, в некоторых запросах нельзя получить результат в соответствии со всеми заданными свойствами. (Например, результат выполнения сложного запроса может оказаться необновляемым.) В этом случае метод executeQuery() возвращает объект ResultSet с меньшими возможностями и с *предупреждением SQLWarning, подклю*ченным к объекту соединения. Подобные предупреждения можно просмотреть с помощью метода getWarnings() класса Connection. Кроме того, для поиска фактически используемого режима работы можно применять методы getType() и getConcurrency() класса ResultSet. Иногда отсутствие проверки фактического режима работы приводит к использованию неподдерживаемой операции, например к применению метода previous() для непрокручиваемого результата выполнения запроса. В таком случае неизбежно возникнет исключение SQLException.

RowSet

Для того, чтобы осуществлять работу в отключенном режиме целесообразно использовать *набор строк (row set). Ин*терфейс RowSet расширяет интерфейс ResultSet, но набор строк не привязан к соединению с базой данных.

В JDBC пять «реализаций» (т.е. расширений) интерфейса Rowset. **JdbcRowSet** — это реализация Rowset с подключением; остальные четыре без подключения:

CachedRowSet — это просто Rowset без подключения;

WebRowSet — это подкласс CachedRowSet, который "знает", как преобразовать свои результаты в XML и обратно;

JoinRowSet — это WebRowSet, который также "умеет" формировать эквивалент SQL JOIN без необходимости подключения к базе данных;

FilteredRowSet — это WebRowSet, способный еще и отфильтровать полученные данные без необходимости подключения к базе данных.

RowSet

```
String studentSQL = "select id, fio from
                                    students";
CachedRowSet rs = new CachedRowSetImpl();
try (Connection cn =
    DriverManager.getConnection (url,user,pass);
    Statement st = cn.createStatement()) {
    rs.populate(st.executeQuery(studentSQL));
}
rs.setTableName("students");
while (rs.next()){
  System.out.println(rs.getString(2));
  rs.updateString(2,rs.getString(2).trim()+ "a");
  rs.updateRow();}
try (Connection cn =
   DriverManager.getConnection (url,user,pass);){
          rs.acceptChanges(cn);     }
```

RowSet

Если данные в базе изменились с того момента, как набор строк был заполнен информацией, то возникают дополнительные сложности, связанные с несоответствием данных. В базовой реализации проверяется, совпадают ли исходные значения набора строк (т.е. значения перед редактированием) с текущими значениями базы. Если проверка дает положительный результат, содержимое базы заменяется модифицированными данными. В противном случае генерируется исключение SyncProviderException и изменения не записываются. В других реализациях могут использоваться другие способы синхронизации данных.

Транзакции

Группа команд может быть оформлена в виде *транзакции (transaction). Которая* может быть *зафиксирована, или окончена (commit), после успешного выполнения всех* команд либо отвергнута (rollback), если при выполнении хотя бы одной из команд произойдет какая-то ошибка.

По умолчанию соединение с базой данных обладает возможностью *автоматической фиксации (autocommit mode), т.е. каждая SQLкоманда фиксируется после ее успешного* выполнения. Причем зафиксированную команду нельзя отвергнуть.

Для проверки текущего режима автоматической фиксации нужно вызвать метод getAutoCommit() класса Connection.

Для отключения режима автоматической фиксации используется метод

```
conn.setAutoCommit(false);
```

Транзакции

Повысить возможности контроля за процессом отката позволяют точки сохранения. При создании точки сохранения помечается позиция, в которую затем можно перейти, не возвращаясь в точку начала транзакции.

```
Statement stat = conn.createStatement();
stat.executeUpdate(command1);
Savepoint svpt = conn.setSavepoint();
stat.executeUpdate(command2);
if (. . .) conn.rollback(svpt);
conn.commit();
```

NULL-значение

- Для передачи в NULL в качестве значения параметра запроса необходимо использовать метод setNull()

```
PreparedStatement ps = con.prepareStatement(
    “insert into NAMES(NAME, AGE) values(?, ?));
ps.setString(1, “Dmitry”);
ps.setNull(2);
```

Получение значений первичного ключа

- Используйте метод `execute` с параметром
  - `Statement.RETURN_GENERATED_KEYS`
  - `int[] columnIndexes`
- Для получения значений используйте метод getGeneratedKeys

```
String insert = "INSERT INTO STUDENTS VALUES (NULL, 'Joe
Black', 1988)";
pstmt = con.prepareStatement(insert,
Statement.RETURN_GENERATED_KEYS);
pstmt.executeUpdate();
ResultSet keys = pstmt.getGeneratedKeys();
keys.next();
int key = keys.getInt(1);
```

Пакет запросов

- JDBC 2.0 предоставляет возможность отправить несколько Update-запросов в составе одного пакета

```
PreparedStatement ps = con.prepareStatement(
  "INSERT INTO NAMES(NAME) VALUES(?)");

ps.setString(1, "Alexander");
ps.addBatch();
ps.setString(1, "Oleg");
ps.addBatch();
ps.executeBatch();
```

Домашнее задание

- Переделать задание с предыдущей лекции для работы с JDBC