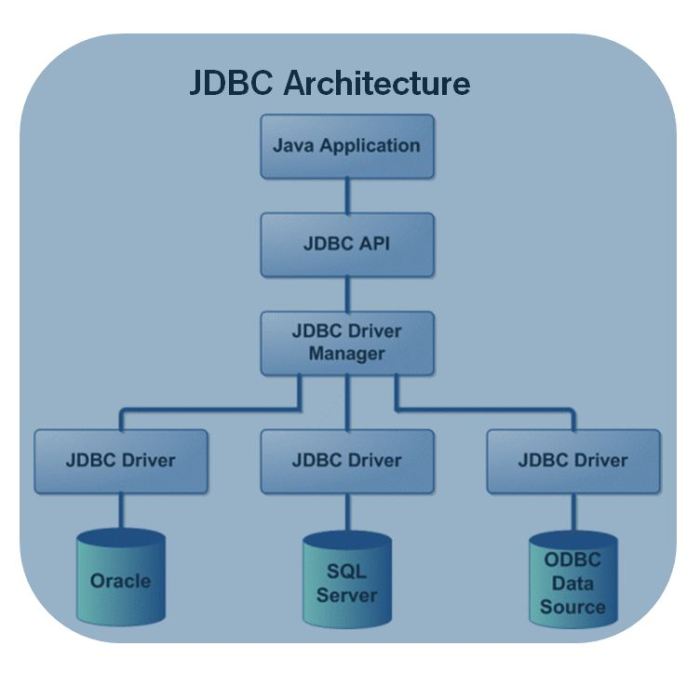
**Java Database Connectivity –**это стандартный API для независимого соединения языка программирования Java с различными базами данных (далее – БД).

JDBC (Java Database Connectivity)– это стандарт взаимодействия приложения с различными СУБД. JDBC основан на концепции драйверов, позволяющей получать соединение с БД по специальному url.

JDBC решает следующие задачи:

* Создание соединения с БД.
* Создание SQL выражений.
* Выполнение SQL – запросов.
* Просмотр и модификация  полученных записей.

JDBC API использует менеджер драйверов и специальные драйверы БД  для обеспечения подключения к различным базам данных.



JDBC API состоит из следующих элементов:

* **Менеджер драйверов (Driver Manager)**Этот элемент управляет списком драйверов БД. Каждой запрос на соединение требует соответствующего драйвера. Первое совпадение даёт нам соединение.
* **Драйвер (Driver)**Этот элемент отвечает за связь с БД. Работать с ним нам приходится крайне редко. Вместо этого мы чаще используем объекты DriverManager, которые управляют объектами этого типа.
* **Соединение (Connection)**Этот интерфейс обеспечивает нас методами для работы с БД. Все взаимодействия с БД происходят исключительно через Connection.
* **Выражение (Statement)**Для подтверждения SQL-запросов мы используем объекты, созданные с использованием этого интерфейса.
* **Результат (ResultSet)**Экземпляры этого элемента содержат данные, которые были получены в результате выполнения SQL – запроса. Он работает как итератор и “пробегает” по полученным данным.
* **Исключения (SQL Exception)**Этот класс обрабатывает все ошибки, которые могут возникнуть при работе с БД.

**ДРАЙВЕР**

Драйвер – это сущность, которая реализует определённые интерфейсы JDBC API для взаимодействия с сервером базы данных (далее – БД).

Например, именно драйвер даёт нам возможность открывать соединения  и выполнять SQL – запросы и т.д.

**JDBC драйвер на основе библиотеки Java**

Этот тип драйверов использует трёх-звенный подход для получения доступа к БД. Для свзяи с промежуточным сервером приложения используется стандартный сетевой сокет. Информация, полученная от этого сокета транслируется промежуточным сервером в формат, который необходим для конкретной БД и направляется в сервер БД.

**Чистая Java.**

Этот тип драйверов разработан полностью с использованием языка программирования Java и работает с БД через сокетное соединение. Главное его преимущество – наибольшая производительность и, обычно, предоставляется разработчиком БД.

**Менеджер драйверов (Driver Manager)**Класс **DriverManager** является уровнем управления JDBC, отслеживает все доступные драйверы и управляет установлением соединений между БД и соответствующим драйвером.

JDBC **DriverManager** является "хребтом" JDBC-архитектуры, и его основная функция очень проста - соединить Java-программу и соответствующий JDBC драйвер.

Чтобы сказать диспетчеру драйверов JDBC, какой именно драйвер следует загрузить, необходимо выполнить одну из команд :

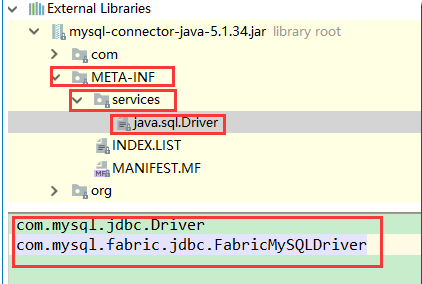
1. Class.forName(“полное имя класса”)
2. Class.forName(“полное имя класса”).newInstance()
3. DriverManager.registerDriver(new “полное имя класса”)

Метод forName класса java.lang.Class используется для получения экземпляра этого класса (возвращает объект Class) с указанным именем класса. Это имя класса указывается в качестве строкового параметра. Таким образом мы можем передавать имя класса динамически.

Эта процедура включает инициализацию статических полей и выполнение блока статической инициализации. В статическом блоке все драйвера создают свой экземпляр и регистрируют этот класс в DriverManager с помощью метода DriverManager.registerDriver().

Начиная с jdbc4.0 при вызове getConnection DriverManager попытается найти подходящий драйвер из тех, которые были загружены при инициализации и явно загруженные с помощью того же загрузчика классов, что и текущее приложение.

Для этого файл jar драйвера должны содержать файл *java.sql.Driver, который находится в каталоге META-INF / services файла jar.* При запуске он определит все jar пакеты и проверит содержат ли они этот файл.



**Connection**

Класс Connection используется для установления соединения с базой данных.

Для того, чтобы установить соединение с БД, нам необходимо выполнить следующие шаги:

* Импортировать пакет JDBC в наш Java код.
* Зарегистрировать JDBC драйвер.
* Передать информацию для соединения с БД (URL, имя пользователя и пароль).
* Создать соединение с помощью метода getConnection().

Стандартный способ установления соединения - это вызов метода **getConnection** класса DriverManager. Методу getConnection необходимо передать строку URL (Uniform Resource Locator). Стандарт подключения к серверу базы данных позволяет использовать следующие методы **getConnection** с параметрами:

getConnection(url);

getConnection(url, properties);

getConnection(url, username, password);

При использовании первого варианта, все параметры подключения указываются в url.

После того, как закончим работу с нашей программой нам необходимо закрыть все соединения и освободить ресурсы

connecion.close();

**STATEMENT**

**Statement** используется для выполнения SQL-запросов.

 Существует три типа класса Statement, которые являются как бы контейнерами для выполнения SQL-выражений через установленное соединение:

* **Statement** предназначен для выполнения простых SQL-запросов без параметров; содержит базовые методы для выполнения запросов и извлечения результатов.
* **PreparedStatement** используется для выполнения SQL-запросов с или без входных параметров; добавляет методы управления входными параметрами.
* **CallableStatement** используется для вызовов хранимых процедур; добавляет методы для манипуляции выходными параметрами.

Statement создается с помощью метода **Connection.createStatement().**

Класс **Statement** содержит три различных метода выполнения SQL-выражений:

* **boolean execute (String SQL)**Этот метод возвращает логическое значение **true**, если объект ResultSet может быть получен. В противном случае он вовращает **false**.
* **int executeUpdate (String SQL)**Этот метода возвращает количесство столбцов в таблице, на которое повлиял наш SQL – запрос.
* **ResultSet executeQuery (String SQL)**Этот метод возвращает нам экземпляр ResultSet.

**PreparedStatement** предварительно компилирует запросы, которые могут содержать входные параметры обозначенные символом '?'. Создается методом **connection.prepareStatement(SQL).** Символы вопроса можно использовать только в параметрах, а не где угодно.

С помощью методов **setXXX()** можем устанавливать значения входных параметров.

pstmt.setString(1, "Кофе")

Экземпляр **CallableStatement** используется для выполнения процедур, непосредоственно в самой БД. Создается с помощью метода **connection.prepareCall(SQL).**

Существует три типа параметров: IN, OUT, INOUT. PreparedStatement использует только IN, а CallableStatement, в свою очередь, использует все три.

* **IN**Параметр, значение которого известно в момент, когда создаётся запрос. Мы назначем параметр IN с помощью метода типа **setXXX()**.
* **OUT**Параметр, значение которого возвращается SQL – запросом. Мы получаем значения из  OUT с помощью методов типа **getXXX()**.
* **INOUT**Параметр, который использует входные и выходные значения. Мы назначем параметр с помощью метода типа **setXXX()**, а получаем значения, с помощью метода типа **getXXX().**

Схожим с PreparedStatement способом, мы, используя экземпляр CallableStatement, должны установить значения параметров.

Когда мы используем параметры типа OUT и INOUT, нам необходимо задействовать дополнительный метод **registerOutParameter()**. Этот метод устанавливает тип данных JDBC в тип данных процедуры.

После того, как мы вызвали процедуру, мы получаем значение из параметра OUT с помощью соответствующего метода **getXXX()**. Этот метод преобразует полученное значение из типа данных SQL в тип данных Java.

**RESULT SET**

Класс ResultSet представляет результирующий набор данных и обеспечивает приложению построчный доступ к результатам запросов. При обработке запроса **ResultSet** поддерживает указатель на текущую обрабатываемую строку.

Методы ResultSet:

* **Методы получения данных.**Эти методы используются для просмотра данных конкретной записи, на которую ссылается указатель.
* **Методы изменения данных.**  
  Методы этой группы используются для изменения данных текущей записи. Эти изменения передаются в используемую БД.
* **Методы навигации.**  
  Эти методы используются для перемещения указателя.

Типы ResultSet:

* **ResultSet.TYPE-FORWARD\_ONLY (по умолчанию)**Указатель двигается только вперёд по множеству полученных результатов.
* **ResultSet.TYPE\_SCROLL\_INTENSIVE**Указатель может двигаться вперёд и назад и не чувствителен к изменениям в БД, которые сделаны другими пользователями после того, как ResultSet был создан.
* **ResultSet.TYPE\_SCROLL\_SENSITIVE**Указатель может двигаться вперёд и назад и чувствителен к изменениям в БД, которые сделаны другими пользователями после того, как ResultSet был создан.

Аргумент  RSConcurrency определяет, используется ли данный экземпляр ResultSet только для чтения, или для чтения и изменения также.

Всего существует два типа этого параметра:

* **ResultSet.CONCUR\_READ\_ONLY**Создаёт экземпляр ResultSet только для чтения. Устанавливается по умолчанию.
* **ResultSet.CONCUR\_UPDATABLE**Создаёт экземпляр ResultSet, который может изменять данные.

Методы для перемещения указателя (Изначально находится перед первым рядом):

* **public void beforeFirst () throws SQLException**Перемещает указатель на место перед первым рядом.
* **public void afterLast () throws SQLException**Перемещает указатель на место после крайнего ряда.
* **public boolean first () throws SQLException**Перемещает указатель на первый ряд.
* **public boolean last () throws SQLException**  
  Перемещает указатель на крайний ряд.
* **public boolean previous () throws SQLException**Перемещает указатель на предыдущий ряд. Возвращает **false**, если предыдущий ряд находится за пределами множества результатов.
* **public boolean next () throws SQLException**Перемещает указатель на следующий ряд. Возвращает **false**, если следующий ряд находится за пределами множества результатов.
* **public void absolute (int row) throws SQLException**Перемещает указатель на указанный ряд.
* **public void relative (int row) throws SQLException**Перемещает указатель на указанное количество рядов от текущего
* **public int getRow () throws SQLException**Возвращает номер ряда, на который в данный момент указывает курсор.
* **public void moveToInsertRow () throws SQLException**Перемещает указатель на ряд в полученном множестве, который может быть использован для того, чтобы добавить новую запись в БД. Текущее положение указателя запоминается.
* **public void moveToCurrentRow () throws SQLExcpetion**Возвращает указатель обратно на текущий ряд в случае, если указатель ссылается на ряд, в который в данный момент добавляются данные.

Получать данные можно по имени и по индексу (первая строка под индексом 1):

* **public XXX getXXX(int columnIndex) throws SQLException**
* **public XXX getXXX (String columnName) throws SQLException**

Чтобы проверить, было ли только что полученное значение NULL, можно использовать метод wasNull.

Мы можем изменять данные, как по имени, так и по индексу колонки:

* **public void updateString (int columnIndex, String s) throws SQLException**
* **public void updateString (String columnName, String s) throws SQLException**Чтобы зафиксировать изменения нужно вызвать updateRow().

Мы также можем работать с рядами в таблице БД:

* **public void insertRow()**Вставляет запись в таблицу БД. Может быть использован только в том случае, когда указатель ссылается на ряд для вставки. То есть предварительно нужно вызвать moveToInsertRow.
* **public void updateRow()**Изменяет текущий ряд в таблице БД.
* **public void deleteRow()**Удаляет текущий ряд из таблицы БД.

Типы данных JDBC

|  |  |  |  |
| --- | --- | --- | --- |
| SQL | JDBC/Java | setXXX | updateXXX |
| VARCHAR | java.lang.String | setString | updateString |
| CHAR | java.lang.String | setString | updateString |
| LONGVARCHAR | java.lang.String | setString | updateString |
| BIT | boolean | setBoolean | updateBoolean |
| NUMERIC | java.math.BigDecimal | setBigDecimal | updateBigDecimal |
| TINYINT | byte | setByte | updateByte |
| SMALLINT | short | setShort | updateShort |
| INTEGER | int | setInt | updateInt |
| BIGINT | long | setLong | updateLong |
| REAL | float | setFloat | updateFloat |
| FLOAT | float | setFloat | updateFloat |
| DOUBLE | double | setDouble | updateDouble |
| VARBINARY | byte[ ] | setBytes | updateBytes |
| BINARY | byte[ ] | setBytes | updateBytes |
| DATE | java.sql.Date | setDate | updateDate |
| TIME | java.sql.Time | setTime | updateTime |
| TIMESTAMP | java.sql.Timestamp | setTimestamp | updateTimestamp |
| CLOB | java.sql.Clob | setClob | updateClob |
| BLOB | java.sql.Blob | setBlob | updateBlob |
| ARRAY | java.sql.Array | setARRAY | updateARRAY |
| REF | java.sql.Ref | SetRef | updateRef |
| STRUCT | java.sql.Struct | SetStruct | updateStruct |

Когда мы работаем с JDBC, то по умолчанию наше соединение работает в режиме **auto-commit**. Это означает, что каждый SQL-запрос является транзакцией, которая автоматически фиксируется.

Для того чтобы получить доступ к управлению транзакциями, нам необходимо использовать метод

connection.setAutoCommit(false);

При отключении auto-commit не одно изменение не будет сохранено в БД до вызова commit(). Сервер базы данных будет блокировать необходимую часть базы данных до подтверждения транзакции, что является ресурсоемкой задачей.

После того, как мы выполнили необходимые нам изменения, мы должны вызвать метод commit() таким образом

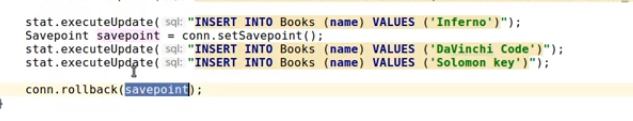
connection.commit();

Если же мы хотим выполнить откат изменений, то нам необходимо вызвать метод rollback():

connection.rollback([savepoint]);

Мы также можем создавать точки сохранения, до которых будет происходить откат. Для управления используются следующие методы:

* **conn.setSavepoint ()**Определяет новую точку сохранения и возвращает экземпляр Savepoint.
* **Conn.releaseSavepoint (savepointName)**Этот метод удаляет точку сохранения. В качестве параметра этот метод принимает экземпляр Savepoint.



Транзакции не распространяются на создание, изменение таблиц.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Уровень изоляции | Транзакции | Dirty reads | Non-repeatable reads | Phantom reads |
| TRANSACTION\_NONE | Не поддерживает | Не применяется | Не применяется | Не применяется |
| TRANSACTION\_READ\_COMMITTED | поддерживает | Предотвращает | Разрешает | Разрешает |
| TRANSACTION\_READ\_UNCOMMITTED | поддерживает | Разрешает | Разрешает | Разрешает |
| TRANSACTION\_REPEATABLE\_READ | поддерживает | Предотвращает | Предотвращает | Разрешает |
| TRANSACTION\_SERIALIZABLE | поддерживает | Предотвращает | Предотвращает | Предотвращает |

JDBC также позволяет устанавливать **уровни изоляции транзакций.** Уровни определены в виде констант интерфейса Connection.

Установить уровень изоляции транзакций можно с помощью метода setTransactionIsolation(int level), а получить методом int getTransactionIsolation(). Эти методы находятся в интерфейсе Connection.

**Обработка пакетов** позволяет нам объединять в логические группы связанные SQL-запросы и посылать их одним “пакетом” в базу данных.

Интерфейсы Statement, PrepparedStatement и CallableStatement имеют метод **addBatch()**, который используется для того, чтобы добавить отдельный SQL-запрос в “пакет”. Метод **executeBatch()** используется для выполнения всех запросов, которые находятся в данном пакете. Возвращает массив с результатами каждого запроса.

После того, как мы добавили запросы в пакет, мы можем удалить их используя метод **clearBatch()**. Этот метод удаляет все запросы, которые были добавлены в “пакет”.

**Spring RowMapper**

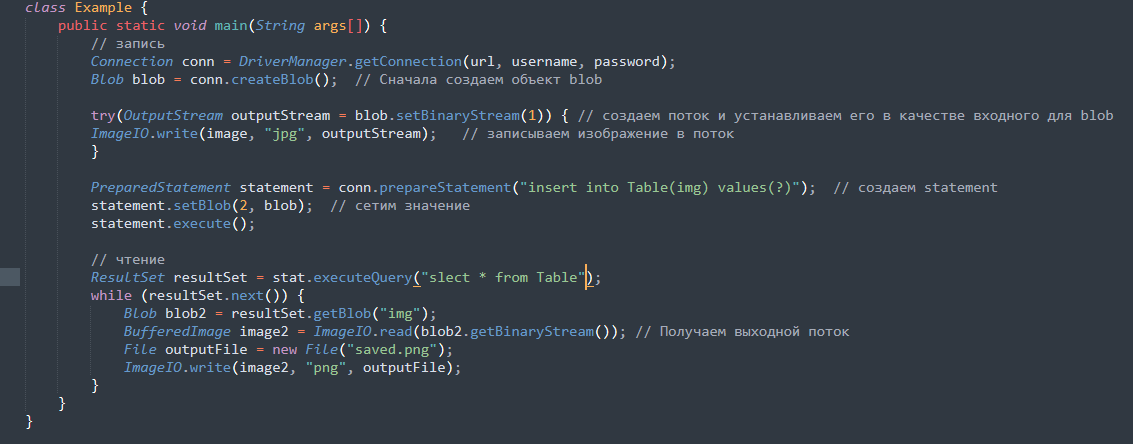
Используется JdbcTemplate для отображения строк ResultSet для каждой строки. Реализации этого интерфейса выполняют фактическую работу по отображению каждой строки в объект результата.

Этот класс обрабатывает отдельно каждую запись, полученную из БД, и возвращает уже готовый объект - модель данных.

**Работа с CLOB и BLOB**

 **BLOB** (от binary large object) - данные хранятся в базе данных в отдельном сегменте\*, а значение типа BLOB содержит указатель на них (LOB locator); данные считаются двоичными.

 **CLOB** (от character large object) - данные хранятся в базе данных в отдельном сегменте\*, а значение типа CLOB содержит указатель на них (LOB locator); данные интерпретируются как текст в кодировке базы данных (database character set).



**Чтение метаданных**

JDBC включает два интерфейса (DatabaseMetaData, ResultSetMetaData) для получения дополнительной информации MetaData о параметрах подключения к серверу БД, самой базе данных и результирующему набору.

**DatabaseMetaData** входит в состав пакета java.sql.\* и содержит методы, описывающие таблицы базы данных, поддержку SQL, хранимые процедуры и другие сведения, относящиеся к базе данных и текущему Connection, которые не относятся непосредственно к выполнению команд и извлечению результирующих наборов.

Типы всех столбцов таблицы БД можно получить с помощью метода **getColumns** класса DatabaseMetaData. Необходимо передать четыре параметра: наименование каталога, наименование схемы, наименование таблицы и наименование столбца.

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| getDatabaseProductName() | Название сервера базы |
| getDatabaseProductVersion() | Версия сервера |
| getDriverName() | Имя драйвера |
| getDriverVersion() | Версия драйвера |
| getUserName() | Пользователь |
| getTableTypes() | Типы таблиц |
| getNumericFunctions() | Математические функции |
| getStringFunctions() | Строковые функции |
| getSystemFunctions() | Системные функции |
| getTimeDateFunctions() | Дата/Время функции |
| getExtraNameCharacters() | Символы для имён, кроме стандартных |
| getSchemaTerm() | Термин для "схема" |
| getProcedureTerm() | Термин для "процедура" |
| getCatalogTerm() | Термин для "каталог" |
| getMaxColumnNameLength() | Макс. длина имени колонки |
| getMaxProcedureNameLength() | Макс. длина имени процедуры |
| getMaxTableNameLength() | Макс. длина имени таблицы |
| getMaxColumnsInSelect() | Макс. кол-во колонок в SELECT |
| getMaxStatementLength() | Макс. длина SQL запроса |
| getMaxColumnsInTable() | Максимальное кол-во колонок в TABLE |
| getMaxConnections() | Максимальное кол-во одновременных соединений с базой |

Класс **ResultSetMetaData** определяет объект метаданных для текущего набора данных ResultSet. **ResultSetMetaData** содержит информацию о результирующей таблице - количество колонок, тип значений колонок и т.д.

|  |  |
| --- | --- |
| getColumnCount() | Количество полей ResultSet |
| getColumnLabel (i) | Имя i-ого поля |
| getColumnType (i) | Максимальное кол-во одновременных соединений с базой |