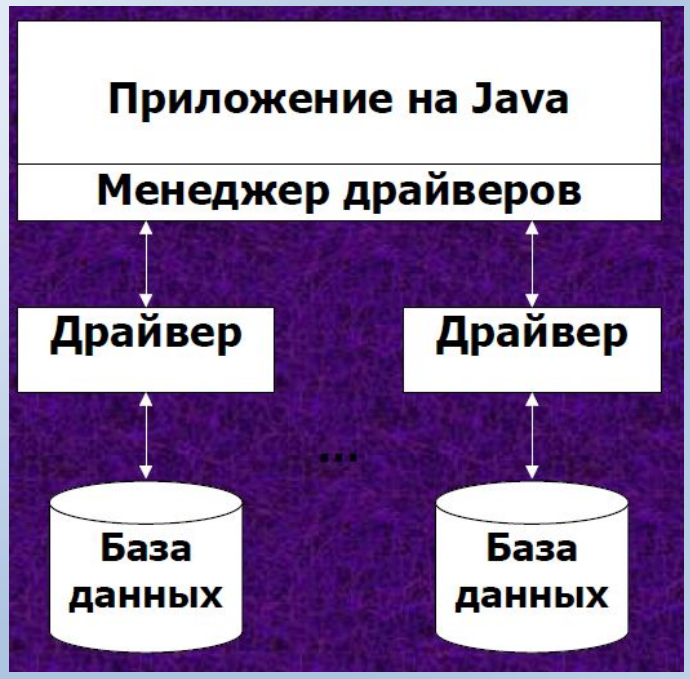
Базы данных

1. Архитектура JDBC (Java Database Connectivity)

- **JDBC** (Java DataBase Connectivity) – это платформенно-независимая технология, позволяющая из программы на Java получить доступ к любой SQL-совместимой базе данных.
- JDBC построен по **драйверной архитектуре**, типичной для универсальных систем доступа к данным
- В настоящий момент действует стандарт JDBC 3.0
- Главное достоинство JDBC — тесное взаимодействие с другими Java-технологиями в рамках создания распределенных систем. В первую очередь это касается JNDI (Java Naming and Directory Interface) и JTS (Java Transaction Service)

Структура JDBC

- В задачу менеджера драйверов входит присоединение Java-приложений к требуемому драйверу JDBC.
- Драйвер поддерживает обмен данными между приложением и базой данных



Принципы построения JDBC

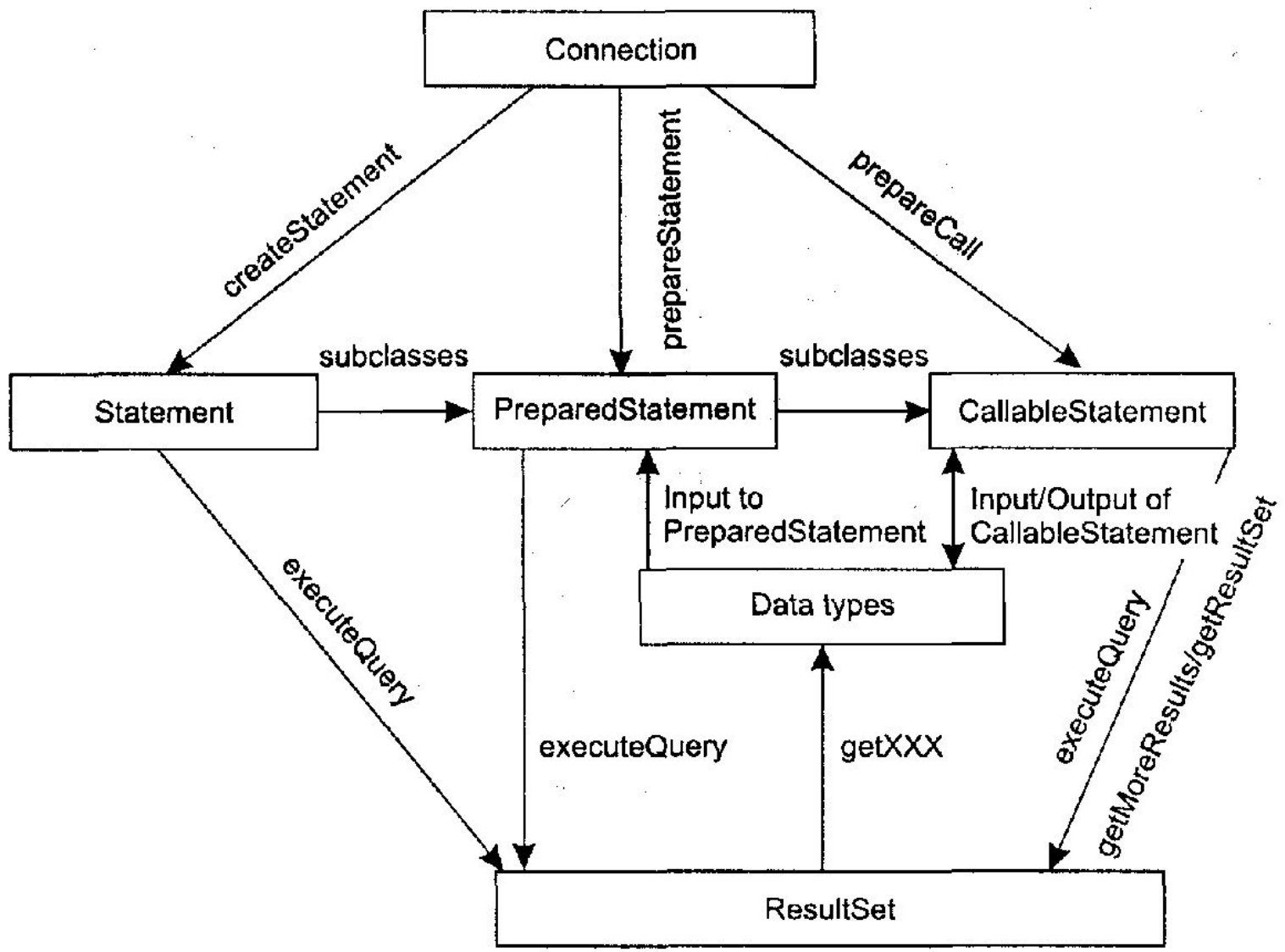
- С точки зрения разработчика можно считать, что JDBC состоит из двух основных частей:
  - **JDBC API**, который содержит набор классов и интерфейсов, определяющих Java – ориентированный доступ к базам данных. Эти классы и методы объявлены в двух пакетах (package) java.sql и javax.sql
  - **JDBC-драйвера**, специфического для каждой базы данных (или других источников данных) JDBC превращает (тем или иным способом) вызовы уровня JDBC API в "родные" команды того или иного сервера баз данных

2. Основные классы и интерфейсы JDBC

- **java.sql.DriverManager** - позволяет загрузить и зарегистрировать необходимый JDBC-драйвер, а затем получить соединение с базой данных.
- **javax.sql.DataSource** - предназначен для решения примерно тех же задач, что и DriverManager, но гораздо более удобным и универсальным образом. В JDBC 2 1 появились интерфейсы javax.sql.ConnectionPoolDataSource и javax.sq1.XADataSource, которые обеспечивают поддержку пула соединений, в том числе таких соединений, которые сопоставлены с внешними (по отношению к серверу базы данных) транзакциями.
- **java.sql.Connection** - обеспечивает формирование запросов к источнику данных и управление транзакциями. Предусмотрены также интерфейсы javax.sql.PooledConnection (логическое соединение с БД из пула соединений) и javax.sql.XAConnection (логическое соединение с БД из пула, сопоставленное с внешней транзакцией).

- java.sql.Statement, java.sql.PreparedStatement и java.sql.CallableStatenient - эти интерфейсы позволяют отправить запрос к источнику данных. Различные виды интерфейсов применяются в зависимости от того, используются в запросе параметры или нет и является ли запрос обращением к хранимой процедуре реляционной базы данных.
- **java.sql.ResultSet** - объявляет методы, которые позволяют перемещаться по набору данных, возвращаемых оператором SELECT, и считывать значения отдельных полей в текущей записи
- **java.sql.ResultSetMetaData** - позволяет получить информацию о структуре набора данных: количество полей, их названия, тип и т. д.
- **java.sql.DatabaseMetaData** - объявлено большое количество методов, позволяющих получить информацию о структуре самого источника данных

Схема взаимодействия классов JDBC



3. Типы данных JDBC

- JDBC поддерживает взаимное отображение между типами данных, характерных для использования SQL, и их Java-аналогами.
- В таблице приведены соответствия между ними (каким образом JDBC-типы отображаются на Java)

JDBC Type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	Boolean
BOOLEAN	Boolean
TINYINT	Byte
SMALL I NT	Short
INTEGER	Int
BIGINT	Long
REAL	Float
REF	Ref
…	…
JAVAOBJECT	Соответствующий класс Java

4.Группы драйверов JDBC

- **Драйверы JDBC 1.0.** Этот драйвер не поддерживает интерфейс DataSource Все соединения, которые могут быть получены через этот драйвер (с помощью вызова метода DriverManager.getConnection()), являются *физическими соединениями с базой данных*
- **Драйверы JDBC 2.0.** Эти драйверы реализуют интерфейс javax.sql DataSource и, как правило, ConnectionPoolDataSource, то есть предоставляют возможность получения и управления *логическими соединениями*, что позволяет значительно эффективнее использовать ресурсы сервера. Но такие драйверы не обеспечивают поддержки распределенных двухфазных транзакций (этот режим необходим, например, при использовании распределенных баз данных)
- **Драйверы JDBC XA.** XA - это протокол, с помощью которого "общаются" друг с другом так называемые *менеджеры транзакций* и *менеджеры ресурсов*. Протокол XA является частью модели объектных транзакций DTP (Distributed Transaction Process) консорциума X/Opeп. Такие драйверы реализуют интерфейс javax.sql.XADataSource, что позволяет управлять логическими соединениями, сопоставленными с контекстами транзакций

- **Мост JDBC-ODBC + драйвер ODBC**: Этот мост использует интерфейс JDBC для доступа к драйверам ODBC.
- **Наполовину-Java драйверы** - этот тип драйверов преобразуют вызовы JDBC в вызовы клиентского API для Oracle, Sybase, Informix, DB2 и т.д. Этот тип драйверов требует, чтобы на каждой клиентской станции был установлен некоторый исполняемый (двоичный) не java код.
- **Сетевые JDBC-драйверы**, которые написаны на Java и транслируют JDBC API вызовы в СУБД-независимый сетевой протокол, который затем транслируется сервером в СУБД-зависимый протокол. Этот сервер служит посредником между всеми своими Java-клиентами и различными СУБД.
  - Например pg73jdbc3.jar (Драйвер для PostgreSQL)
- **Драйверы СУБД, которые полностью написаны на Java** и сами, без дополнительного промежуточного сервера, способны "общаться" по сети с сервером БД, реализуя тот или иной сетевой протокол
  - Например: MM.MySQL Driver

5. Инсталляция драйверов JDBC

- Драйверы 3 и 4 типов написаны на Java, поэтому соответствующие классы должны быть доступны и путь к ним прописан в переменной среды CLASSPATH
- При использовании моста JDBC-ODBC должен быть создан источник данных DSN
- В JDBC описанием базы данных служит ее URL-адрес, который строится по образцу описателя ресурсов Интернет и имеет вид:

  протокол:подпротокол:информация_для_соединения
- В качестве протокола всегда указывается "jdbc".
- **Подпротокол** идентифицирует драйвер JDBC. Это может быть db2 - для СУБД DB-2, mysql - для СУБД MySQL, mssql – для MS SQL Server, odbc - для драйвера-моста и т.п.
- Информация для соединения зависит от выбранного драйвера и может включать имя хоста, номер порта, имя источника данных ODBC, параметры соединения и другие элементы.

  Пример: String url = "jdbc:mssql://localhost/mentor";

6. Регистрация драйверов

- Регистрация драйверов осуществляется классом DriverManager.
- Задача класса DriverManager - обеспечить поиск нужного JDBC-драйвера среди всех доступных при поступлении запроса клиента, который содержит URL нужной базы данных.
- Класс DriverManager хранит список объектов типа Driver, которые зарегистрировались с помощью вызова DriverManager.registerDriver.
- Класс дравера загружается с помощью вызова

  Class.forName("имя класса драйвера").newInstance()
- В случае работы с MySQL нужно:
  - Добавить библиотеку с драйвером для MySQL (mm.mysql-2.0.14-bin.jar) в CLASSPATH проекта.
  - Осуществить загрузку драйвера, вызвав метод

    Class.forName("org.gjt.mm.mysql.Driver").newInstance();

Регистрация драйвера MySQL

- Здесь приведена часть исходного кода класса org.gjt.mm.mysql.Driver

```
public class Driver implements java.sql.Driver {
…
static {
try {
java.sql.DriverManager.registerDriver(new Driver());
} catch (java.sql.SQLException E) {
throw new RuntimeException("Can't register driver!");
}
if (debug) {
Debug.trace("ALL");
}
}
…
}
```

7. Установление соединения с БД

- Чтобы получить доступ из программы к данным базы необходимо:
  - зарегистрировать драйвер
  - задать URL и создать объект соединения при помощи одного из статических методов **getConnection() класса** DriverManager

- **Пример.** Установить соединение с базой данных MySQL "mentor"

```
Class.forName("org.gjt.mm.mysql.Driver").newInstance();
String url = "jdbc:mysql://localhost/mentor";
Connection con = DriverManager.getConnection(url, "root", "1");
```

- Второй и третий параметры метода getConnection означают имя пользователя и пароль.

Создание ConnectionFactory

```
public class ConnectionFactory {
private static final String DB_USER_NAME="root";
private static final String DB_USER_PASS=“1";
private static final String DB_HOST=“localhost";
private static final String DB_DATABASE=“mybase";
static{
try{
Class.forName("org.gjt.mm.mysql.Driver").newInstance();
} catch(Exception e){
e.printStackTrace();
}
}
public static Connection getMySQlConnection() throws SQLException{
String url = "jdbc:mysql://" + DB_HOST + "/" + DB_DATABASE;
return DriverManager.getConnection(url, DB_USER_NAME,
DB_USER_PASS);
```

8. Выполнение SQL-команд

- Для выполнения запросов к БД в Java используются три интерфейса:
  - Statement - для операторов SQL без параметров
  - PreparedStatement - для операторов SQL с параметрами и часто выполняемых операторов
  - CallableStatement - для исполнения хранимых в базе процедур
- Интерфейсы PreparedStatement и CallableStatement расширяют интерфейс Statement, поэтому имеют все его методы.
- Объекты-носители интерфейсов создаются при помощи методов объекта Connection
  - createStatement() возвращает объект Statement
  - prepareStatement() возвращает объект PreparedStatement
  - prepareCall() возвращает объект CallableStatement

Методы выполнения SQL-команд

- Интерфейс **Statement** предоставляет три различных метода выполнения SQL-команд:
  - **executeQuery()** - для запросов, результатом которых является один единственный набор значений, таких как запросов SELECT. Метод возвращает набор данных, полученный из базы
  - **executeUpdate()** - для выполнения операторов INSERT, UPDATE или DELETE, а также для операторов DDL (Data Definition Language). Метод возвращает целое число, показывающее, сколько строк данных было модифицировано
  - **execute()** – исполняет SQL-команды, которые могут возвращать различные результаты. Например, может использоваться для операции CREATE TABLE, и т.д.

Пример запроса данных

```
try {
Connection con = ConnectionFactory.getMySQLConnection();
Statement st = con.createStatement();
// Выполнить SQL-оператор изменения данных
String sql = "INSERT INTO tutor (id_tutor, name) VALUES (123, 'Иванов')";
int n = st.executeUpdate(sql);
// Выполнить SQL-оператор выборки данных
sql = "SELECT * FROM student";
ResultSet rs = st.executeQuery(sql);
// Закрываем ресурсы !!!!!!!!!!!!!!!!
rs.close();
st.close();
con.close();
}
catch (ClassNotFoundException ex) { …
}
catch (SQLException ex) { …
}
```

Правильное закрытие ресурсов

```
Connection con = null;
try{
con = ConnectionFactory.getMySQlConnection();
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM user");
// doing something
rs.close();
st.close();
} catch(SQLException e){
e.printStackTrace();
} finally{
if (con != null){
try{
con.close();
} catch(SQLException e){
e.printStackTrace();
}
}
}
```

- Метод executeQuery() возвращает объект с интерфейсом ResultSet, который хранит в себе результат запроса к базе данных
- В наборе данных есть курсор, который может указывать на одну из строк таблицы, эта строка называется текущей.
  - **Курсор перемещается по строкам при помощи метода next().**
- Сразу после получения набора данных его курсор находится перед первой строкой. Чтобы сделать первую строку текущей надо вызвать метод **next()**
- Поля текущей записи (колонки таблицы) доступны программе при помощи методов интерфейса ResultSet: getInt() getFloat(), getString(), getDate() и им подобных.

Пример получения данных

```
// Получить набор данных
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM Student");
// Распечатать набор данных
while (rs.next()) {
// Напечатать значения в текущей строке.
int id_student = rs.getInt("id_student");
String name = rs.getString("name");
float average = rs.getFloat("average");
System.out.println("Строка = " + id_student + " " + name + " " +
average);
} // Закрыть объект оператора
st.close();
```

Значение NULL

- Некоторые поля таблиц в базах данных могут не иметь значения.
  - Те методы ResultSet.getXxx(), которые преобразуют данные в объектный тип, возвращают в этом случае значение null.
  - Те методы, которые преобразуют данные в простые типы, возвращают 0.
- Чтобы отличить не инициализированные поля от полей с нулевым значением, можно воспользоваться методом ResultSet.wasNull()