




Lesson 22

28.03.2022

```
public class ex1 {  
    public static void main(String[] args) {  
        {  
            for (;;)   
                System.out.println("Java");  
        }  
    }  
}
```


```
public class ex2 {  
    public static void main(String[] args) {  
        try {  
            foo();  
        } catch (Exception ex) {  
            System.out.println("exMain");  
            ex.printStackTrace();  
        }  
    }  
  
    public static void foo() {  
        try {  
            throw new IllegalArgumentException("catch");  
        } finally {  
            try {  
                throw new RuntimeException("finally");  
            } catch (IllegalArgumentException ex) {  
                System.out.println("exFoo");  
                ex.printStackTrace();  
            }  
        }  
    }  
}
```

```
public class ex3 {  
    public static void main(String[] args) {  
        int i = 10 ;  
        System.out.println(i > 3 != false );  
    }  
}
```



```
public class ex4 {  
    public static void main(String[] args) {  
        byte var = 100 ;  
        switch (var) {  
            case 100 :  
                System.out.println( "var is 100" );  
                break ;  
            case 200 :  
                System.out.println( "var is 200" );  
                break ;  
            default :  
                System.out.println( "In default" );  
        }  
    }  
}
```

```
public class ex5 {  
    public static void main(String[] args) {  
        StringBuilder sb = new StringBuilder("TOMATO");  
        System.out.println(sb.reverse()  
            .replace('O', 'A'));  
    }  
}
```



```
ALTER TABLE table_name
ADD column_name datatype;
```

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

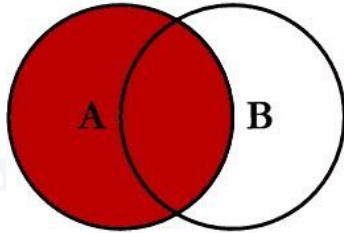
```
SELECT column1, column2, ...
FROM table_name;
```

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

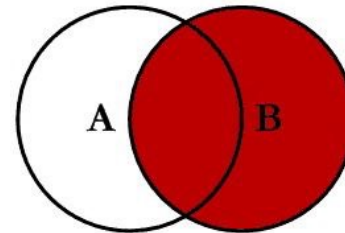
```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

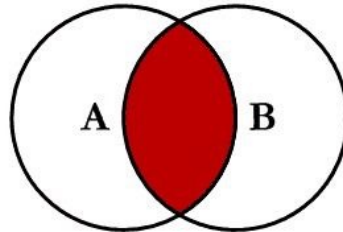
SQL JOINS



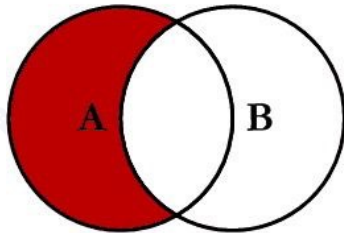
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key
```



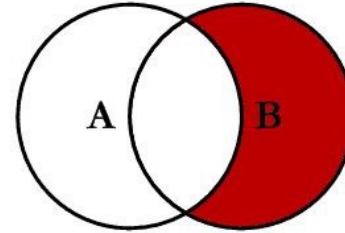
```
SELECT <select_list>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.Key = B.Key
```



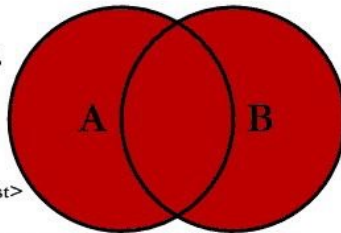
```
SELECT <select_list>  
FROM TableA A  
INNER JOIN TableB B  
ON A.Key = B.Key
```



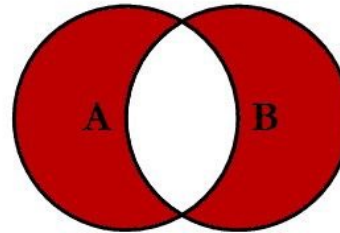
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key  
WHERE B.Key IS NULL
```



```
SELECT <select_list>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.Key = B.Key  
WHERE A.Key IS NULL
```



```
SELECT <select_list>  
FROM TableA A  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key
```



```
SELECT <select_list>  
FROM TableA A  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key  
WHERE A.Key IS NULL  
OR B.Key IS NULL
```


Employee

EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1002	Anna	1	3500
1003	James	1	2500
1004	David	2	5000
1005	Mark	2	3000
1006	Steve	3	4500
1007	Alice	3	3500

```
SELECT DeptID, AVG(Salary)  
FROM Employee  
GROUP BY DeptID;
```

GROUP BY
Employee Table
using DeptID

DeptID	AVG(Salary)
1	3000.00
2	4000.00
3	4250.00



Aggregate Functions

SUM(): Returns the sum or total of each group.

COUNT(): Returns the number of rows of each group.

AVG(): Returns the average and mean of each group.

MIN(): Returns the minimum value of each group.

MAX(): Returns the maximum value of each group.

Employee

EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1002	Anna	1	3500
1003	James	1	2500
1004	David	2	5000
1005	Mark	2	3000
1006	Steve	3	4500
1007	Alice	3	3500

```
SELECT DeptID, AVG(Salary)
FROM Employee
GROUP BY DeptID;
```

GROUP BY
Employee Table
using DeptID

DeptID	AVG(Salary)
1	3000.00
2	4000.00
3	4250.00

```
SELECT DeptID, AVG(Salary)
FROM Employee
GROUP BY DeptID
HAVING AVG(Salary) > 3000;
```

HAVING

DeptID	AVG(Salary)
2	4000.00
3	4250.00



Primary key (PK)

В каждой таблице БД может существовать первичный ключ. Под первичным ключом понимают поле или набор полей, однозначно (уникально) идентифицирующих запись. Первичный ключ должен быть минимально достаточным: в нем не должно быть полей, удаление которых из первичного ключа не отразится на его уникальности.

Foreign key(FK)

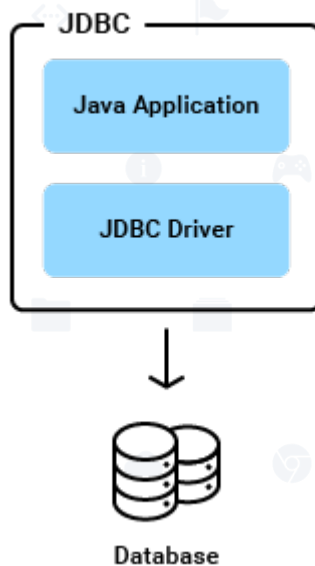
Обеспечивает однозначную логическую связь, между таблицами одной БД.



Первичный ключ (главный ключ, primary key, PK). Представляет собой столбец или совокупность столбцов, значения которых однозначно идентифицируют строки.

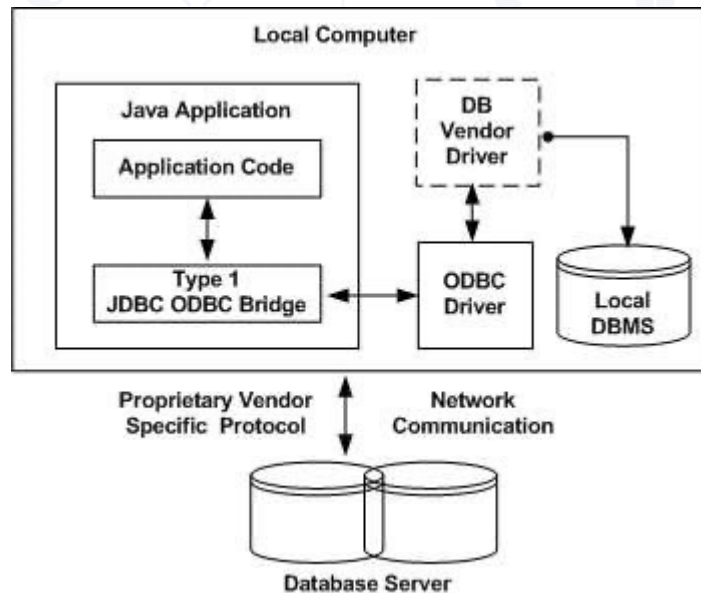
Вторичный ключ (внешний, foreign key, FK) - Столбец или совокупность столбцов, которые в данной таблице не являются первичными ключами, но являются первичными ключами в другой таблице.

JDBC (Java DataBase Connectivity — соединение с базами данных на Java) предназначен для взаимодействия Java-приложения с различными системами управления базами данных (СУБД). Всё движение в JDBC основано на драйверах которые указываются специально описанным URL.



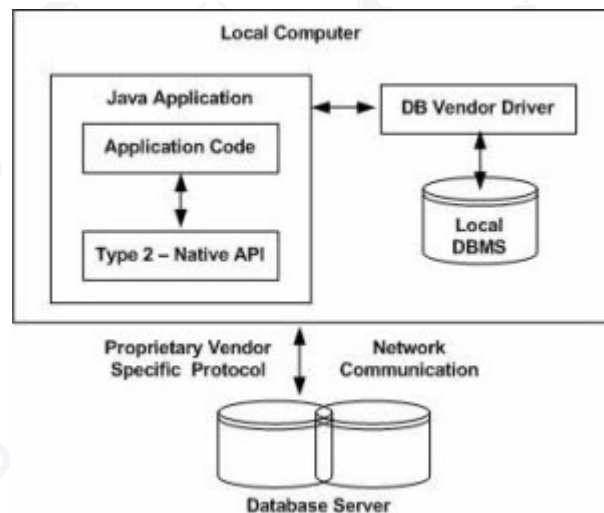
JDBC – ODBC транслятор

Этот тип драйвера транслирует JDBC в установленный на каждой машине клиентской машине ODBC. Использование ODBC требует конфигурации DSN, который является целевой БД.



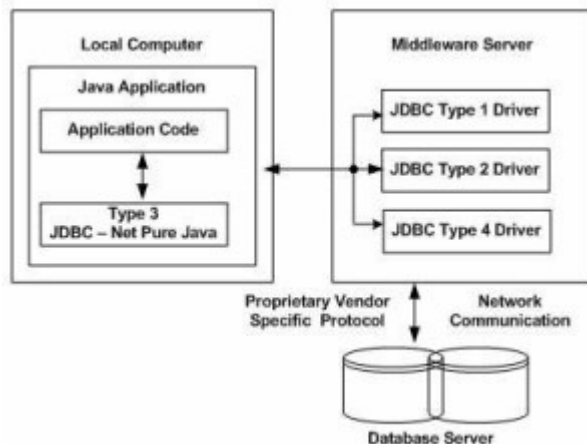
JDBC – нативный API

В этом драйвере JDBC API преобразовывается в уникальный для каждой БД нативный C/C++ API. Его принцип работы крайне схож с драйвером первого типа.



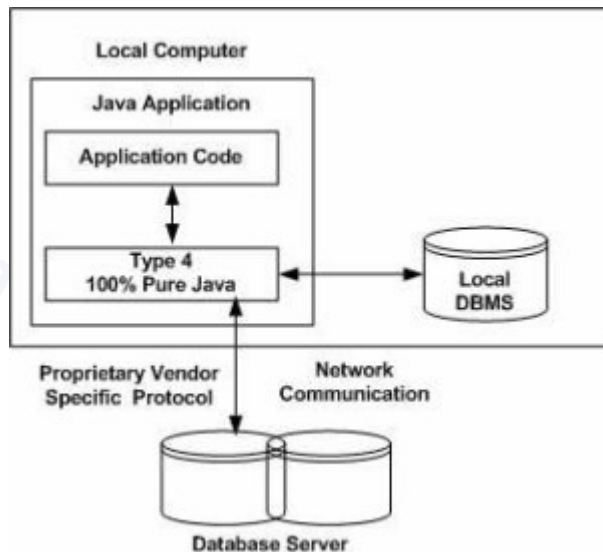
JDBC драйвер на основе библиотеки Java

Этот тип драйверов использует трёх-звенный подход для получения доступа к БД. Для связи 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.




```
<dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.38</version>
  </dependency>
</dependencies>
```

DriverManager и JDBC Driver

```
Class.forName(driverClass);  
Connection connection = DriverManager  
    .getConnection(url, user, password) ;
```

Где `driverClass` - это строка с полным именем класса JDBC драйвера, например `org.h2.Driver` для H2 Database или `com.mysql.jdbc.Driver` для MySQL.

`DriverManager` - это синглтон, который содержит информацию о всех зарегистрированных драйверах. Метод `getConnection` на основании параметра URL находит `java.sql.Driver` соответствующей базы данных и вызывает у него метод `connect`.



Все основные сущности в JDBC API, с которыми вам предстоит работать, являются интерфейсами:

- ✓ Connection;
- ✓ Statement;
- ✓ PreparedStatement;
- ✓ CallableStatement;
- ✓ ResultSet;
- ✓ Driver;
- ✓ DatabaseMetaData.



Соединение к базе данных

```
Class.forName("com.mysql.jdbc.Driver");  
Connection connect = DriverManager  
    .getConnection( url: "jdbc:mysql://localhost:3306/student?"  
        + "user=root&password=root");
```

Таким образом, мы получили реализацию интерфейса `java.sql.Connection` для нашей базы данных.

Statement и ResultSet

На основании соединения можно получить объект `java.sql.Statement` для выполнения запросов к базе.

```
Statement statement = connect.createStatement();  
statement.executeQuery( sql: "select * from city");
```

Statement можно использовать для выполнения любых запросов, будь то DDL, DML, либо обычные запросы на выборку данных.

ResultSet

Объект ResultSet - это результат выполнения запроса.

```
Statement statement = connect.createStatement();  
ResultSet resultSet = statement.executeQuery( sql: "select * from city");  
  
System.out.println(  
    |         resultSet.getMetaData().getTableName( column: 1)  
    |  
);
```

Объекты Connection, Statement и ResultSet после использования необходимо закрывать. Поэтому приведенный выше код необходимо обернуть в try-finally и в блоке finally добавить закрытие ресурсов:

PreparedStatement

Если вам нужно выполнить несколько похожих запросов, то разумным решением будет использование PreparedStatement.

PreparedStatement представляет собой скомпилированную версию SQL-выражения, выполнение которого будет быстрее и эффективнее.

```
PreparedStatement preparedStatement = connection.prepareStatement( sql: "insert into city(city) values (?)");
```

```
List<String> cityList = Arrays.asList("London", "Paris", "Madrid", "Berlin");
```

```
cityList.forEach(city -> {  
    try {  
        preparedStatement.setString( parameterIndex: 1, city);  
        preparedStatement.executeUpdate();  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
});
```



Транзакции в JDBC

Рто знакомился с Hibernate минуя JDBC, обычно очень удивляет работа с транзакциями. По умолчанию каждое SQL-выражение автоматически коммитится при выполнении `statement.execute` и подобных методов. Для того, чтобы открыть транзакцию сначала необходимо установить флаг `autoCommit` у соединения в значение `false`. Ну а дальше нам пригодятся всем знакомые методы `commit` и `rollback`.

```
connection.setAutoCommit(false);

Statement st = connection.createStatement();
try {
    st.execute("insert into user(name) values('kesha')");
    connection.commit();
} catch (SQLException e) {
    connection.rollback();
}
```



DatabaseMetaData

С помощью Connection можно получить очень полезную сущность DatabaseMetaData. Она позволяет получить метаинформацию о схеме базы данных, а именно какие в базе данных есть объекты - таблицы, колонки, индексы, триггеры, процедуры и так далее.