<epam> Delivering Excellence in Software Engineering

Введение в JDBC

Java Database Connectivity

План лекции

Вопросы:

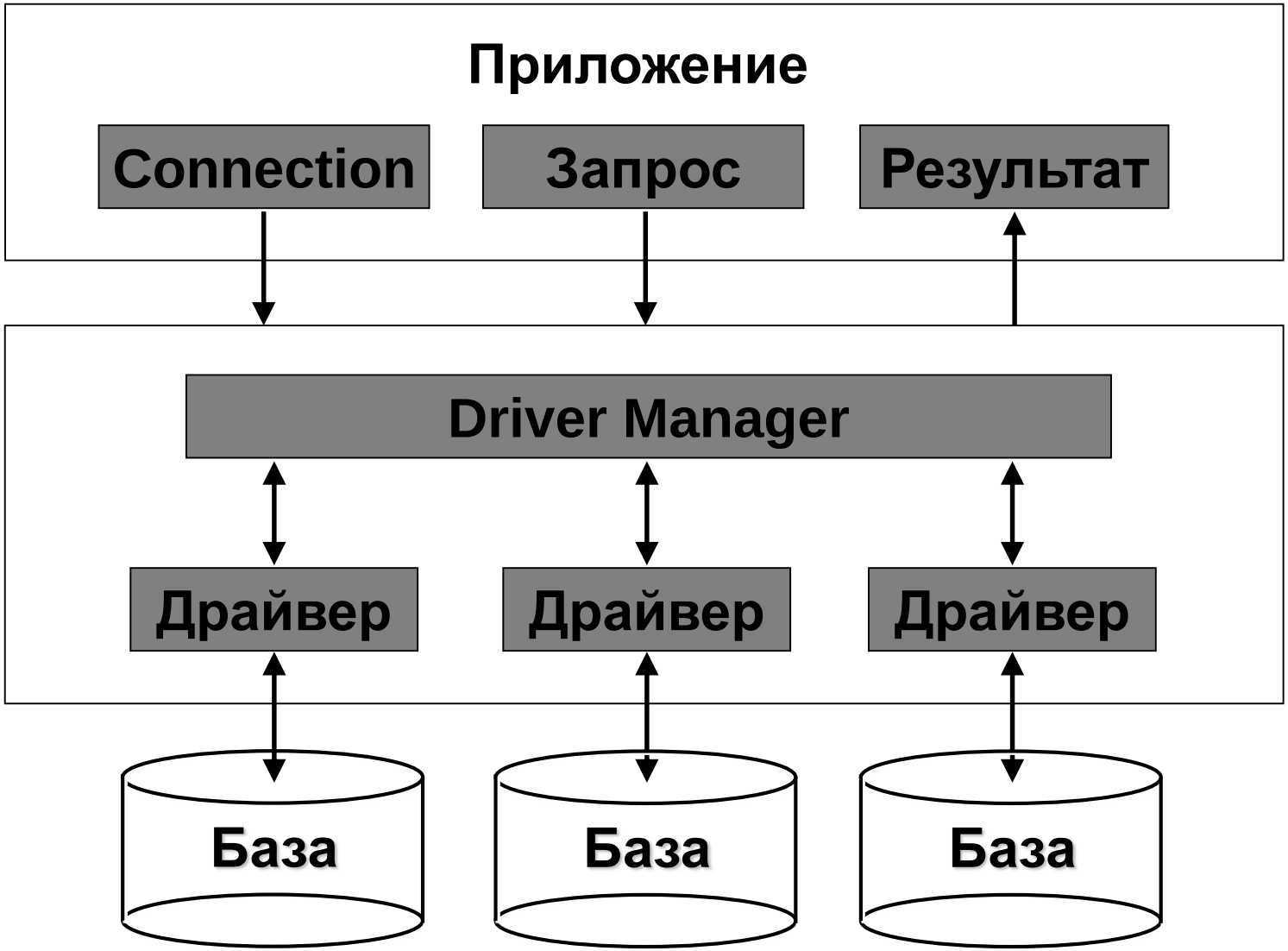
1. Что такое JDBC?
2. Использование JDBC API
3. DataSource & Connection Pooling
4. Transaction
5. Prepared and Callable Statements

Технология JDBC

JDBC – это стандартный прикладной интерфейс (API) языка Java для организации взаимодействия между приложением и СУБД.

- JDBC является частью стандартной версии Java и находится в пакете `java.sql`
- Дополнительная функциональность представлена в пакете `javax.sql`

Архитектура приложения, использующего JDBC



Компоненты JDBC

Driver Manager

- предоставляет средства для управления набором драйверов баз данных
- предназначен для выбора базы данных и создания соединения с БД.

Драйвер

- обеспечивает реализацию общих интерфейсов для конкретной СУБД и конкретных протоколов

Соединение (Connection)

- Сессия между приложением и драйвером базы данных

Компоненты JDBC (2)

Запрос

- SQL запрос на выборку или изменение данных

Результат

- Логическое множество строк и столбцов таблицы базы данных

Метаданные

- Сведения о полученном результате и об используемой базе данных

Доступ к данным через JDBC

Выполнение запросов

Использование JDBC.

Последовательность действий:

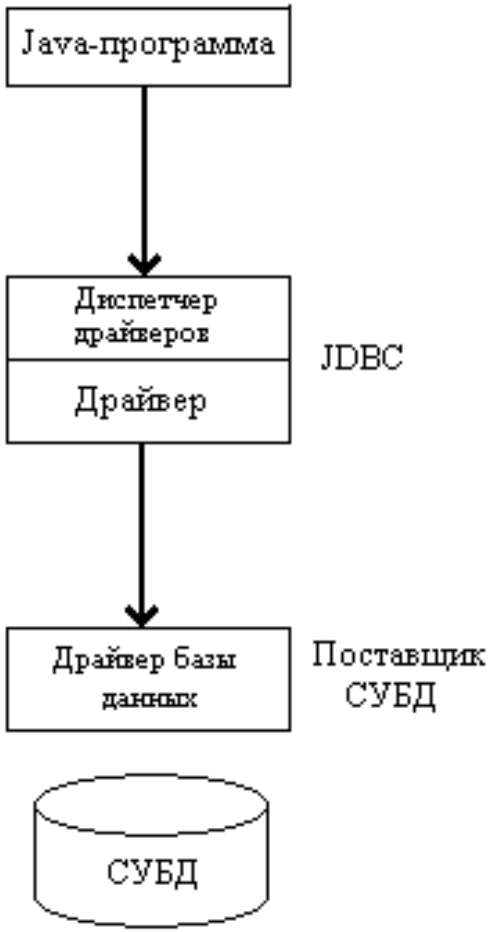
1. Загрузка класса драйвера базы данных.
2. Установка соединения с БД.
3. Создание объекта для передачи запросов.
4. Выполнение запроса.
5. Обработка результатов выполнения запроса.
6. Закрытие соединения.

Шаг 1: Загрузка класса драйвера базы данных

JDBC основана на концепции т. н. драйверов, позволяющих получать соединение с базой данных по специально описанному URL.

Драйверы JDBC обычно создаются поставщиками СУБД.

Их работа заключается в обработке JDBC-подключений и команд, поступающих от Java-приложения, и в генерации машинно-зависимых вызовов по отношению к базе данных.



Шаг 1: Загрузка класса драйвера базы данных

Загрузка класса драйвера базы данных:

в общем виде:

```
Class.forName([location of driver]);
```

для MySQL:

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

для JDBC-ODBC bridge (ex. MS Access) :

```
Class.forName("sun.Jdbc.odbc.jdbcodbcDriver ");
```

СУБД MySQL

СУБД MySQL совместима с JDBC и будет применяется для создания экспериментальных БД.

Последняя версия СУБД может быть загружена с сайта www.mysql.com.

Для корректной установки необходимо следовать инструкциям мастера установки.

Для запуска следует использовать команду из папки /mysql/bin:

mysqld-nt -standalone

Если не появится сообщение об ошибке, то СУБД MySQL запущена. Для создания БД и таблиц используются команды языка SQL.

Дополнительно требуется подключить библиотеку, содержащую драйвер MySQL

mysql-connector-java-3.1.12.jar.

Шаг 2: Установка соединения с БД

Для установки соединения необходимо вызвать метод **getConnection()** класса **DriverManager.**

В качестве параметров передаются:

- Тип, физическое месторасположение и имя БД;
- Логин и пароль для доступа.

```
Connection cn = DriverManager.getConnection(
"jdbc:mysql://localhost/my_db", "login", "password");
```

В результате будет возвращен объект **Connection**, содержащий одно установленное соединение с БД **my_db**.

Шаг 2: Установка соединения с БД

Еще один способ соединения с базой данных возможен с использованием файла ресурсов database.properties, в котором хранятся, как правило, путь к БД, логин и пароль доступа.

Например:

```
url=jdbc:mysql://localhost/my_db?useUnicode=true&
characterEncoding=Cp1251
driver=org.gjt.mm.mysql.Driver
user=root
password=pass
```

Шаг 2: Установка соединения с БД

В этом случае соединение создается в классе бизнес-логики, отвечающем за взаимодействие с базой данных, с помощью следующего кода:

```
public Connection getConnection() throws SQLException {
    ResourceBundle resource = ResourceBundle.getBundle("database");
    String url = resource.getString("url");
    String driver = resource.getString("driver");
    String user = resource.getString("user");
    String pass = resource.getString("password");
    try {
            Class.forName(driver).newInstance();
    } catch (ClassNotFoundException e) {
       throw new SQLException("Драйвер не загружен!");
    } catch (InstantiationException e) {
            e.printStackTrace();
    } catch (IllegalAccessException e) {
            e.printStackTrace();
    }
   return DriverManager.getConnection(url, user, pass);
}
```

Шаг 2: Установка соединения с БД

Полезные методы класса DriverManager:

registerDriver() - регистрация драйвера;

getDrivers() - получение списка всех драйверов.

Существует целый ряд методов интерфейса DatabaseMetaData, которые предоставляют возможность поиска информации о БД.

Метаданные

Получить объект DatabaseMetaData можно следующим образом:

```
DatabaseMetaData dbMetaData = cn.getMetaData();
```

В результате из полученного объекта DatabaseMetaData можно извлечь:

- название и версию СУБД методами getDatabaseProductName(), getDatabaseProductVersion(),
- название и версию драйвера - методами getDriverName(), getDriverVersion(),
- имя драйвера JDBC – методом getDriverName(),
- имя пользователя БД – методом getUserName(),
- местонахождение источника данных – методом getURL() .

Шаг 3: Создание объекта для передачи запросов

Объект **Statement**

- используется для выполнения запросов и команд SQL, а также для установки некоторых ограничений на запросы;
- один и тот же объект **Statement** может быть использован многократно для различных запросов.

```
Connection dbCon = DriverManager.getConnection(
 "jdbc:mysql://localhost/my_db","admin", "secret");
Statement stmt = dbCon.createStatement();
```

Шаг 4: Выполнение запроса

Метод **executeQuery()** выполняет предварительно созданный SQL запрос на выборку **(SELECT).**

Результаты выполнения запроса помещаются в объект **ResultSet**.

```
Connection dbCon =
  DriverManager.getConnection(
 "jdbc:mysql://localhost/my_db","admin",
"secret");

Statement stmt = dbCon.createStatement();

ResultSet rs = stmt.executeQuery(
  "SELECT first_name FROM employees");
```

Шаг 4: Выполнение запроса

Для **INSERT/UPDATE/DELETE** запросов используется метод **executeUpdate()**, который возвращает количество добавленных (измененных, удаленных) записей.

```
Statement stmt = dbCon.createStatement();
int rowsAffected = stmt.executeUpdate(
  "UPDATE employees SET salary = salary*1.2");
```

Шаг 5: Обработка результатов выполнения запроса

Содержится в объекте **ResultSet**

Методы:

- boolean next()
- xxx getXxx(int columnNumber)
- xxx getXxx(String columnName)
- void close()

Итератор первоначально устанавливается в позицию перед первой строкой

- Необходимо вызвать **next()** для перемещения в позицию первой строки.
- Когда строки закончатся, метод **next()** возвратит значение **false**.

Шаг 5: Обработка результатов выполнения запроса

Исходная таблица: Employees.

Id	FirstName	LastName	Address
1	**Илья**	**Петров**	**ул. Кульман, 16-45**
2	**Николай**	**Иванов**	**ул. Гамарника, 46-120**
3	**Иван**	**Сидоров**	**ул. Гикало, 32-24**

```
ResultSet rs = st.executeQuery("SELECT LastName + ' ' +
FirstName AS FullName, Address FROM Employees");
```

В результате rs содержит:

FullName	Address
Илья Петров	**ул. Кульман, 16-45**
Николай Иванов	**ул. Гамарника, 46-120**
Иван Сидоров	**ул. Гикало, 32-24**

```
while(rs.next())
{
    System.out.println(
            rs.getString("FullName") +
"\t" +      rs.getString("Address"));
}
```

<epam>

Необходимые сведения о структуре БД

Существует целый ряд методов интерфейса ResultSetMetaData с помощью которых можно определить типы, свойства и количество столбцов БД.

```
ResultSet rs = stmt.executeQuery(
  "SELECT * FROM employees");
ResultSetMetaData rsm = rs.getMetaData();
int number = rsm.getColumnCount();
for (int i=0; i<number; i++) {
  System.out.println(rsm.getColumnName(i));
}
```

ResultSetMetaData – Пример

```
ResultSetMetaData meta = rs.getMetaData();

//Return the column count
int iColumnCount = meta.getColumnCount();

for (int i =1 ; i <= iColumnCount ; i++){
  System.out.println("Column Name: " + meta.getColumnName(i));
  System.out.println("Column Type" + meta.getColumnType(i));
  System.out.println("Display Size: " +
    meta.getColumnDisplaySize(i) );
  System.out.println("Precision: " + meta.getPrecision(i));
  System.out.println("Scale: " +  meta.getScale(i) );
}
```

Шаг 6: Закрытие соединения

По окончании использования необходимо последовательно вызвать метод **close()** для объектов **ResultSet**, **Statement** и **Connection** для освобождения ресурсов.

```
try {
  Connection conn = ...;
  Statement stmt = ...;
  ResultSet rset = stmt.executeQuery(...);
  ...
} finally
    // clean up
    rset.close();
    stmt.close();
    conn.close();
}
```

<epam>

Объект SQLException

Обязательно обрабатывайте исключительные ситуации (java.sql.SQLException).

```
try  {
  rset = stmt.executeQuery(
    "SELECT first_name, last_name FROM employee");
  } catch (SQLException sqlex) {
      ... // Обрабатываем ошибки
  } finally {
    // Освобождаем использованные ресурсы
    try {
      if (rset != null) rset.close();
    } catch (SQLException sqlex)  {
      ... // Игнорируем ошибки при закрытии
    }
  ...
}
```

<epam>

Рекомендации к индивидуальному заданию:

1. **Конфигурацию базы хранить в XML.**
2. **Класс , который зачитывает конфигурацию из XML реализовать как Singleton.**
3. **Отдельным классом реализовать работу с Connection.**
4. **Сами SQL-запросы хранить как минимум в константах.**
5. **Реализовать интерфейс для работы с DAO (основные операции: чтение, запись, удаление, поиск).**

```
public class JdbcConnector {
   private Connection conn;
   public Connection getConnection() throws JDBCConnectionException {
      ConfigurationManager cfg = ConfigurationManager.getInstance();
      try {
         Class.forName(cfg.getDriverName);
         conn = DriverManager.getConnection(cfg.getURL(), cfg.getLogin(),
             cfg.getPassword());
      } catch (ClassNotFoundException e) {
        throw new JDBCConnectionException("Can't load database driver.", e);
      } catch (SQLException e) {
        throw new JDBCConnectionException("Can't connect to database.", e);
      }
      if(conn==null) {
         throw new JDBCConnectionException("Driver type is not correct in URL " +
    cfg.getProperty(ConfigurationManager.DB_URL) + ".");
      }
      return conn;
   }
...
}
```

Закрытие Connection

```
public void close() {
    if (conn != null) {
        try {
            conn.close();
        } catch (SQLException e) {
        }
    }
}

@Override
public void finalize() {
    if (conn != null) {
        try {
            conn.close();
        } catch (SQLException e) {
        }
    }
}
```

Использование в DAO

```
JdbcConnector connector = new JdbcConnector();
Connection con = connector.getConnection();
...
try {
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(sql);
    ...
}
} catch (SQLException ex) {
    throw new JDBCConnectionException("SQL query in class:
"+this.getClass().getName()+"is not correct.", ex);
} finally {
    connector.close ();
}
```

При использовании JDBC Connections, Statements должны быть закрыты:

```
Connection con = null;
PreparedStatement preparedstatement = null;
Resultset resultset = null;
try {
  con = db.getConnection(true);
  preparedstatement = con.prepareStatement("SELECT xxx  WHERE ID = ?");
  resultset = preparedstatement.executeQuery();
  ...
}
  catch (Exception e)
  {
      // Do error handing here
  }
  finally {
      if (resultset != null)
       try {
         resultset.close();
       } catch  (SQLException e) {}
      if (preparedstatement != null)
       try {
         preparedstatement.close();
       } catch  (SQLException e) {}
     if (con != null){
       db.releaseConnection(con);
     }
 }
```

JDBC Запросы

Интерфейсы PreparedStatement и CallableStatement

Подготовленные запросы и хранимые процедуры

PreparedStatement

- Предварительно готовится и хранится в объекте. Позволяет ускорить обмен информацией с БД.

CallableStatement

- Используется для выполнения хранимых процедур, созданных средствами самой СУБД.

PreparedStatement

Для компиляции SQL запроса, в котором отсутствуют конкретные значения, используется метод **prepareStatement(String sql),** возвращающий объект **PreparedStatement.**

Подстановка реальных значений происходит с помощью методов **setString(), setInt()** и подобных им.

Выполнение запроса производится методами **executeUpdate(), executeQuery().**

PreparedStatement-оператор предварительно откомпилирован, поэтому он выполняется быстрее обычных операторов ему соответствующих.

PreparedStatement - Пример

```
class Rec {
        static void insert(PreparedStatement ps, int id, String name,
                String surname, int salary) throws SQLException {
                ps.setInt(1, id);
                ps.setString(2, name);
                ps.setString(3, surname);
                ps.setInt(4, salary);
        //      выполнение компилированного запроса
                ps.executeUpdate();
        }
}
…
Connection cn = null;
…
String sql = "INSERT INTO emp(id,name,surname,salary)
                                                VALUES(?,?,?,?)";
//      компиляция запроса
PreparedStatement ps = cn.prepareStatement(sql);
Rec.insert(ps, 2203, "Иван", "Петров", 230);
```

Prepared Statement

PreparedStatement используется для:

- Выполнения запроса с параметрами;
- Улучшения производительности в случае частого использования запроса.

CallableStatement

В терминологии JDBC, хранимая процедура - последовательность команд SQL, хранимых в БД и доступных любому пользователю этой СУБД. Механизм создания и настройки хранимых процедур зависит от конкретной базы данных.

Интерфейс CallableStatement обеспечивает выполнение хранимых процедур

Объект CallableStatement содержит команду вызова хранимой процедуры, а не саму хранимую процедуру.

CallableStatement способен обрабатывать не только входные (IN) параметры, но и выходящие (OUT) и смешанные (INOUT) параметры. Тип выходного параметра должен быть зарегистрирован методом registerOutParameter().

После установки входных и выходных параметров вызываются методы execute(), executeQuery() или executeUpdate().

CallableStatement - Пример

Пусть в БД существует хранимая процедура getempname, которая по уникальному для каждой записи в таблице employee числу SSN будет возвращать соответствующее ему имя:

```
CREATE PROCEDURE getempname (emp_ssn IN INT,
                             emp_name OUT VARCHAR) AS
BEGIN
        SELECT name INTO emp_name FROM employee
                WHERE SSN = EMP_SSN;
END
```

Вызов данной процедуры из программы:

```
String SQL = "{call getempname (?,?)}";
CallableStatement cs = conn.prepareCall(SQL);
cs.setInt(1,822301);
//регистрация выходящего параметра
cs.registerOutParameter(2,java.sql.Types.VARCHAR);
cs.execute();
String empName = cs.getString(2);
System.out.println("Employee with SSN:" + ssn + " is " +
empName);
// Будет выведено: Employee with SSN:822301 is Spiridonov
```

Чтение автоматически сгенерированного ключа

Для СУБД, которые поддерживают "auto increment" поля

- Например MS SQL Server, MySQL, ...
- JDBC имеет доступ к автоматически сгенерированным ключам

```
// добавляем запись...
int rowCount = stmt.executeUpdate(
  "INSERT INTO Messages(Msg) VALUES ('Test')",
  Statement.RETURN_GENERATED_KEYS);

// ... и получаем ключ
ResultSet rs = stmt.getGeneratedKeys();
rs.next();
long primaryKey = rs.getLong(1);
```

Batch-команды

Механизм batch-команд позволяет запускать на исполнение в БД массив запросов SQL вместе, как одну единицу.

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.addBatch("INSERT INTO employee VALUES (10, 'Joe ')");
stmt.addBatch("INSERT INTO location VALUES (260, 'Minsk')");
stmt.addBatch("INSERT INTO emp_dept VALUES (1000, 260)");
// submit a batch of update commands for execution
int[] updateCounts = stmt.executeBatch();
```

Метод PreparedStatement.executeBatch() **возвращает массив чисел, каждое из которых характеризует число строк, которые были изменены конкретным запросом из batch-команды.**

DataSource & Connection Pooling
(пул соединений)

Использование Connection в Web-приложениях

Если использовать стандартный подход, т.е. при каждом обращении клиента создавать новое соединение, обмениваться данными с СУБД и закрывать соединение, то при каждом обращении Вы будете тратить драгоценное время на создание соединения с СУБД и его закрытие.

При достаточно большой нагрузке на сервер это может стать ощутимой проблемой.

Если же Вы пользуетесь технологией Servlet, которая позволяет хранить данные между обращениями пользователя, то лучшим выходом будет создание класса - пула соединений (DB Connection Pool).

Пул соединений

- **Пул соединений реализуется согласно шаблону Singleton. В нем необходимо создать свойство-коллекцию соединений (например - ArrayList), в котором будут хранится все свободные соединения с СУБД.**
  В конструкторе класса можно осуществить загрузку JDBC драйвера СУБД. Кроме того, создаются методы создания нового соединения, получения последнего свободного соединения и возвращения соединения обратно в пул.
  При обращении к Вашему сервлету в методе service (doGet/doPost для HttpServlet) следует вызвать статический метод getInstance() пула, который возвращает объект пула (см. шаблон Singleton), после чего у полученного объекта вызвать метод получения свободного соединения.
- **По завершении работы с БД следует вернуть соединение в пул, вызвав соответствующий метод.**

Пул соединений

Для работы с БД сначала необходимо открыть соединение к ней и получить объект типа Connection. Пул соединений представляет собой класс в виде набора объектов JDBC Connection и методов доступа к ним.

По своей сути это – *контейнер с простейшим интерфейсом для контроля и управления над производимыми соединениями к базе данных*.

Реализация пула выполнена в виде класса – DataSource, который даёт возможность:

- **загрузить необходимые драйвера для конкретной базы данных;**
- **получить ссылку на объект типа DataSource;**
- **получить доступное соединение типа Connection из хранилища;**
- **возвратить соединение обратно в хранилище;**
- **уничтожить все ресурсы и закрыть все соединения из хранилища;**

Context.xml under META-INF

```
<Context path="/ dbtest " docBase=" dbtest " debug="5"
    reloadable="true" crossContext="true">
    <logger timestamp="true" suffix=".txt"
    prefix="localhost_realtorApp_log."
    classname="org.apache.catalina.logger.FileLogger"/>
    <Resource name="jdbc/ testDB " auth="Container"
    type="javax.sql.DataSource" maxActive="100" maxIdle="30"
    maxWait="10000" username="javauser" password="javadude"
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost:3306/javatest?autoReconnect=true"/>
 </Context>
```

Web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
    <description>MySQL Test App</description>
    <resource-ref>
        <description>DB Connection</description>
        <res-ref-name>jdbc/ testDB </res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
</web-app>
```

Пул соединений

Класс InitialContext, как часть JNDI API, обеспечивает работу с каталогом именованных объектов. В этом каталоге можно связать объект источника данных DataSource с некоторым именем (не только с именем БД, но и вообще с любым), предварительно создав объект DataSource.

Пул соединений

Шаблон использования:

```
static {
    Context cxt = new InitialContext();
    if ( cxt == null )
    {
     throw new Exception("No context!");
    }

    DataSource ds = (DataSource)
    cxt.lookup("java:/comp/env/jdbc/testDB" );

    if ( ds == null )
    {
     throw new Exception("Data source not found!");
     }
  }
```

Работа с транзакциями в JDBC

Транзакции - Пример

Пример: *перечисление денег с одного счета на другой*.

Если сбой произошел в тот момент, когда операция снятия с одного счета деньги уже произведена, а операция зачисления на другой счет еще не произведена, то система позволяющие такие ситуации должна быть признана не отвечающей требованиям заказчика.

Такие операции должны выполняться обе или не выполняться вовсе.

В этом случае такие две операции трактуют как одну и называют транзакцией**.**

Транзакции

Транзакции должны следовать принципам ACID:

Атомарность **– две или более операций выполняются все или не выполняется ни одна. Успешно завершенные транзакции фиксируются, в случае неудачного завершения происходит откат всей транзакции.**

Согласованность **– если происходит сбой, то система возвращается в состояние до начала неудавшейся транзакции. Если транзакция завершается успешно, то проверка согласованности проверяет успешное завершение всех операций транзакции.**

Изолированность **– во время выполнения транзакции все объекты-сущности, участвующие в ней, должны быть синхронизированы.**

Долговечность **– все изменения, произведенные с данными во время транзакции, записываются в базу данных. Это позволяет восстанавливать систему.**

Транзакции

Чтобы все операции SQL выполняли транзакции, в БД используется ключевое слово COMMIT**.**

В JDBC эта операция выполняется по умолчанию после каждого вызова методов executeQuery() **и** executeUpdate()**.**

Включение режима неавтоматического подтверждения операций: вызывается метод setAutoCommit() **интерфейса** Connection **с параметром** false**.**

Подтверждает выполнение SQL-запросов метод commit() **интерфейса** Connection**, в результате действия которого все изменения таблицы производятся как одно логическое действие.**

Метод rollback() **отменяет действия всех запросов SQL, начиная от последнего вызова** commit()**.**

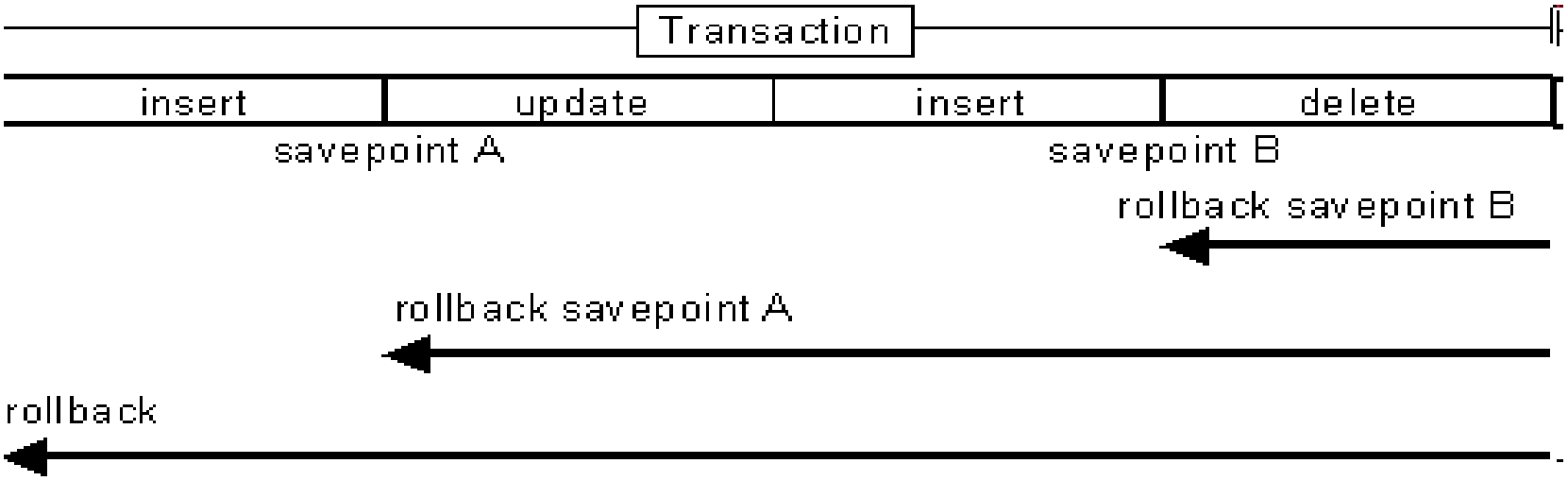
Транзакции

```
cn.setAutoCommit( false );

   ...

   bError = false;
   try
   {
      for( ... )
     {
// validate data, set bError true if error
       if ( bError )
       {
         break;
       }
       st.executeUpdate( ... );
     }
   if ( bError )
   {
     cn.rollback();
   }
   else
   {
     cn.commit();
   }

} // end try
catch ( SQLException SQLe)
{
  cn.rollback();
  ...
} // end catch
catch ( Exception e)
{
  cn.rollback();
  ...
} // end catch
```

Точки сохранения

Начиная с версии 3.0, JDBC поддерживает точки сохранения.

Интерфейс Savepoint **позволяет разделить транзакцию на логические блоки, дающие возможность откатывать совершённые изменения не к последнему вызову** commit(). **, а лишь к заранее установленной точке сохранения.**



Точки сохранения - Пример

```
cn.setAutoCommit(false);

Statement st = cn.createStatement();
int rows = st.executeUpdate("INSERT INTO Employees " +
            "(FirstName, LastName) VALUES " + "('Игорь', 'Цветков')");
// Устанавливаем именованную точку сохранения.
Savepoint svpt = cn.setSavepoint("NewEmp");

// ...
rows = st.executeUpdate("UPDATE Employees
           set Address = 'ул. Седых, 19-34' " +
            "WHERE LastName = 'Цветков'");

// ...
cn.rollback(svpt);
// ...
// Запись о работнике вставлена, но адрес не обновлен.
conn.commit();
```

Транзакции

Для транзакций существует несколько типов чтения:

Грязное чтение (dirty reads) происходит, когда транзакциям разрешено видеть несохраненные изменения данных. Иными словами, изменения, сделанные в одной транзакции, видны вне ее до того, как она была сохранена. Если изменения не будут сохранены, то, вероятно, другие транзакции выполняли работу на основе некорректных данных;

Непроверяющееся чтение (nonrepeatable reads) происходит, когда транзакция А читает строку, транзакция Б изменяет эту строку, транзакция А читает ту же строку и получает обновленные данные;

Фантомное чтение (phantom reads) происходит, когда транзакция А считывает все строки, удовлетворяющие WHERE-условию, транзакция Б вставляет новую или удаляет одну из строк, которая удовлетворяет этому условию, транзакция А еще раз считывает все строки, удовлетворяющие WHERE-условию, уже вместе с новой строкой или недосчитавшись старой.

Транзакции

JDBC удовлетворяет четырем уровням изоляции транзакций, определенным в стандарте SQL:2003.

Уровни изоляции транзакций определены в виде констант интерфейса Connection (по возрастанию уровня ограничения):

TRANSACTION_NONE – информирует о том, что драйвер не поддерживает транзакции;

TRANSACTION_READ_UNCOMMITTED – позволяет транзакциям видеть несохраненные изменения данных, что разрешает грязное, непроверяющееся и фантомное чтения;

TRANSACTION_READ_COMMITTED – означает, что любое изменение, сделанное в транзакции, не видно вне неё, пока она не сохранена. Это предотвращает грязное чтение, но разрешает непроверяющееся и фантомное;

TRANSACTION_REPEATABLE_READ – запрещает грязное и непроверяющееся, но фантомное чтение разрешено;

TRANSACTION_SERIALIZABLE – определяет, что грязное, непроверяющееся и фантомное чтения запрещены.

Уровни изоляции транзакций

Установка уровня изоляции - `setTransactionIsolation(level)`

Transaction Level	Permitted Phenomena			Impact
	Dirty Reads	Non-Repeatable Reads	Phantom Reads	
TRANSACTION_NONE	-	-	-	FASTEST
TRANSACTION_READ_UNCOMMITED	YES	YES	YES	FASTEST
TRANSACTION_READ_COMMITED	NO	YES	YES	FAST
TRANSACTION_REPEATABLE_READ	NO	NO	YES	MEDIUM
TRANSACTION_SERIALIZABLE	NO	NO	NO	SLOW

<epam>

Преимущества JDBC

- **Лёгкость разработки: разработчик может не знать специфики базы данных, с которой работает**
- **Код не меняется, если компания переходит на другую базу данных**
- **Не нужно устанавливать громоздкую клиентскую программу**
- **К любой базе можно подсоединиться через легко описываемый URL**

Delivering Excellence in Software Engineering