<https://www.mkyong.com/spring/spring-jdbctemplate-querying-examples/>

Spring JdbcTemplate Querying examples

## 1. Querying for Single Row

Here’s two ways to query or extract a single row record from database, and convert it into a model class.

## 1.1 Custom RowMapper

In general, It’s always recommended to implement the RowMapper interface to create a custom RowMapper to suit your needs.

package com.mkyong.customer.model;

import java.sql.ResultSet;

import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class CustomerRowMapper implements RowMapper

{

public Object mapRow(ResultSet rs, int rowNum) throws SQLException {

Customer customer = new Customer();

customer.setCustId(rs.getInt("CUST\_ID"));

customer.setName(rs.getString("NAME"));

customer.setAge(rs.getInt("AGE"));

return customer;

}

}

Copy

Pass it to queryForObject() method, the returned result will call your custom mapRow() method to match the value into the properly.

public Customer findByCustomerId(int custId){

String sql = "SELECT \* FROM CUSTOMER WHERE CUST\_ID = ?";

Customer customer = (Customer)getJdbcTemplate().queryForObject(

sql, new Object[] { custId }, new CustomerRowMapper());

return customer;

}

## 1.2 BeanPropertyRowMapper

In Spring 2.5, comes with a handy RowMapper implementation called ‘BeanPropertyRowMapper’, which can maps a row’s column value to a property by matching their names. Just make sure both the property and column has the same name, e.g property ‘custId’ will match to column name ‘CUSTID’ or with underscores ‘CUST\_ID’.

public Customer findByCustomerId2(int custId){

String sql = "SELECT \* FROM CUSTOMER WHERE CUST\_ID = ?";

Customer customer = (Customer)getJdbcTemplate().queryForObject(

sql, new Object[] { custId },

new BeanPropertyRowMapper(Customer.class));

return customer;

}

Copy

## 2. Querying for Multiple Rows

Now, query or extract multiple rows from database, and convert it into a List.

## 2.1 Map it manually

In mutiple return rows, RowMapper is not supported in queryForList() method, you need to map it manually.

public List<Customer> findAll(){

String sql = "SELECT \* FROM CUSTOMER";

List<Customer> customers = new ArrayList<Customer>();

List<Map> rows = getJdbcTemplate().queryForList(sql);

for (Map row : rows) {

Customer customer = new Customer();

customer.setCustId((Long)(row.get("CUST\_ID")));

customer.setName((String)row.get("NAME"));

customer.setAge((Integer)row.get("AGE"));

customers.add(customer);

}

return customers;

}

Copy

## 2.2 BeanPropertyRowMapper

The simplest solution is using the BeanPropertyRowMapper class.

public List<Customer> findAll(){

String sql = "SELECT \* FROM CUSTOMER";

List<Customer> customers = getJdbcTemplate().query(sql,

new BeanPropertyRowMapper(Customer.class));

return customers;

}

Copy

## 3. Querying for a Single Value

In this example, it shows how to query or extract a single column value from database.

## 3.1 Single column name

It shows how to query a single column name as String.

public String findCustomerNameById(int custId){

String sql = "SELECT NAME FROM CUSTOMER WHERE CUST\_ID = ?";

String name = (String)getJdbcTemplate().queryForObject(

sql, new Object[] { custId }, String.class);

return name;

}

Copy

## 3.2 Total number of rows

It shows how to query a total number of rows from database.

public int findTotalCustomer(){

String sql = "SELECT COUNT(\*) FROM CUSTOMER";

int total = getJdbcTemplate().queryForInt(sql);

return total;

}

Copy

Run it

package com.mkyong.common;

import java.util.ArrayList;

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.mkyong.customer.dao.CustomerDAO;

import com.mkyong.customer.model.Customer;

public class JdbcTemplateApp

{

public static void main( String[] args )

{

ApplicationContext context =

new ClassPathXmlApplicationContext("Spring-Customer.xml");

CustomerDAO customerDAO = (CustomerDAO) context.getBean("customerDAO");

Customer customerA = customerDAO.findByCustomerId(1);

System.out.println("Customer A : " + customerA);

Customer customerB = customerDAO.findByCustomerId2(1);

System.out.println("Customer B : " + customerB);

List<Customer> customerAs = customerDAO.findAll();

for(Customer cust: customerAs){

System.out.println("Customer As : " + customerAs);

}

List<Customer> customerBs = customerDAO.findAll2();

for(Customer cust: customerBs){

System.out.println("Customer Bs : " + customerBs);

}

String customerName = customerDAO.findCustomerNameById(1);

System.out.println("Customer Name : " + customerName);

int total = customerDAO.findTotalCustomer();

System.out.println("Total : " + total);

}

}

Copy

## Conclusion

The JdbcTemplate class, comes with many useful overloaded query methods. It’s advise to refer to the existing query method before you create own customize query method, because Spring may done it for you already.

package com.prokarma.pkmst.controller;

import com.prokarma.pkmst.model.ClientCard;

import org.threeten.bp.LocalDate;

import io.swagger.annotations.\*;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestHeader;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.RequestPart;

import org.springframework.web.multipart.MultipartFile;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.jdbc.core.JdbcTemplate;

import java.util.List;

import com.fasterxml.jackson.databind.ObjectMapper;

import java.io.IOException;

import java.sql.ResultSet; // 25.08.2018

import java.sql.SQLException; // 25.08.2018

import org.springframework.jdbc.core.RowMapper; // 25.08.2018

/\*\*

\* Api implemention

\* @author pkmst

\*

\*/

@javax.annotation.Generated(value = "io.swagger.codegen.languages.JavaPKMSTServerCodegen", date = "2018-08-18T07:07:33.250Z")

@Controller

public class ClientApiController implements ClientApi {

private final ObjectMapper objectMapper;

@Autowired //Подключение к БД

private JdbcTemplate jdbcTemplate; // Добавили поле класса

public ClientApiController(ObjectMapper objectMapper) {

this.objectMapper = objectMapper;

}

public ResponseEntity<Void> addClientCard(

@ApiParam(value = "ClientCard that needs to be added" ,required=true ) @RequestBody ClientCard body,

@RequestHeader(value = "Accept", required = false) String accept) throws Exception {

// do some magic!

return new ResponseEntity<Void>(HttpStatus.OK);

}

public ResponseEntity<List<ClientCard>> findClientCardByClientCriteria(@ApiParam(value = "", required = true) @RequestParam(value = "Name", required = true) String name,

@ApiParam(value = "", required = true) @RequestParam(value = "Surname", required = true) String surname,

@ApiParam(value = "") @RequestParam(value = "ParentName", required = false) String parentName,

@ApiParam(value = "", allowableValues = "Passport, Foreign passport, Seaman passport") @RequestParam(value = "TypeDoc", required = false) String typeDoc,

@ApiParam(value = "") @RequestParam(value = "SeriaDoc", required = false) String seriaDoc,

@ApiParam(value = "") @RequestParam(value = "NumDoc", required = false) String numDoc,

@ApiParam(value = "") @RequestParam(value = "Issuer", required = false) String issuer,

@ApiParam(value = "") @RequestParam(value = "IssueDate", required = false) LocalDate issueDate,

@RequestHeader(value = "Accept", required = false) String accept) throws Exception {

// do some magic!

if (accept != null && accept.contains("application/xml")) {

return new ResponseEntity<List<ClientCard>>(objectMapper.readValue("", List.class), HttpStatus.OK);

}

if (accept != null && accept.contains("application/json")) {

return new ResponseEntity<List<ClientCard>>(objectMapper.readValue("", List.class), HttpStatus.OK);

}

return new ResponseEntity<List<ClientCard>>(HttpStatus.OK);

}

public ResponseEntity<ClientCard> getClientCardById(@ApiParam(value = "ID of ClientCard to return",required=true ) @PathVariable("ClientCardId") Long clientCardId,

@RequestHeader(value = "Accept", required = false) String accept) throws Exception {

// do some magic!

string docTypeQuery = "select " + //24.08.2018

"DocTypeID" +

", DocName" +

" FROM DocTypes";

List<Map<String, Object>> docTypes = jdbcTemplate.queryForList(docTypeQuery);

Map<String,String> doctypesMap = new HashMap<>();

for (Map<String, Object> row: docTypes) {

String id = (String) row.get("DocTypeId");

String Name = (String) row.get("DocName");

doctypesMap.put(id,name);

}

string personDocQuery = "select " + //24.08.2018

"PersonDocID" +

", ClientCardID" +

" , DocTypeID" +

" , SeriaDoc" +

" , NumDoc" +

" , Issuer" +

" , IssueDate" +

" FROM PersonDoc" +

" WHERE ClientCardID = ?;";

Object[] args = new Object[] { clientCardID };

List<Map<String,Object>> rows = jdbcTemplate.queryForList(personDocQuery, args);

List<PersonDoc> docs = new ArrayList<>(); //Объект типа List<PersonDoc>

for (Map<String, Object> row: rows) {

PersonDoc doc = new PersonDoc();

Object idObj = row.get("PersonDocId"); // преобразование данных

String idStr = (String) idObj;

Long id = Long.ParseLong(idstr);

doc.setId(id);

String docTypeId = (String) row.get("DocTypeId");

String docTypeName = doctypesMap.get(docTypeId);

PersonDoc.TypeDocEnum type = PersonDoc.TypeDocEnum.fromValue(docTypeName);

doc.setTypeDoc(type);

Long seriaDoc = Long.parseLong((String) row.get("SeriaDoc")));

doc.setSeriaDoc(seriaDoc);

Long numDoc = Long.parseLong((String) row.get("NumDoc")));

doc.setNumDoc(numDoc);

Long issuer = Long.parseLong((String) row.get("Issuer")));

doc.setIssuer(issuer);

Long issueDate = Long.parseLong((String) row.get("IssuerDate")));

doc.setIssuer(issuerDate);

docs.add(doc);

}

clientCard = new clientCard();

for (PersonDoc doc: docs) {

card.addPersonDocItems(doc);

}

//выполняем запрос к БД

//обрабатывам результат по аналогии с PersonDoc

return new ResponseEntity<ClientCard> (card, HttpStatus.OK); // 24.08.2018

PersonDoc Class

package com.prokarma.pkmst.model;

import java.util.Objects;

import com.fasterxml.jackson.annotation.JsonProperty;

import com.fasterxml.jackson.annotation.JsonCreator;

import com.fasterxml.jackson.annotation.JsonValue;

import io.swagger.annotations.ApiModel;

import io.swagger.annotations.ApiModelProperty;

import org.threeten.bp.LocalDate;

/\*\*

\* Response class to be returned by Api

\* @author pkmst

\*

\*/

/\*\*

\* PersonDoc

\*/

@javax.annotation.Generated(value = "io.swagger.codegen.languages.JavaPKMSTServerCodegen", date = "2018-08-18T07:07:33.250Z")

public class PersonDoc {

@JsonProperty("id")

private Long id = null;

/\*\*

\* PersonDoc type

\*/

public enum TypeDocEnum {

PASSPORT("Passport"),

FOREIGN\_PASSPORT("Foreign passport"),

SEAMAN\_PASSPORT("Seaman passport");

private String value;

TypeDocEnum(String value) {

this.value = value;

}

@Override

@JsonValue

public String toString() {

return String.valueOf(value);

}

@JsonCreator

public static TypeDocEnum fromValue(String text) {

for (TypeDocEnum b : TypeDocEnum.values()) {

if (String.valueOf(b.value).equals(text)) {

return b;

}

}

return null;

}

}

@JsonProperty("TypeDoc")

private TypeDocEnum typeDoc = null;

@JsonProperty("SeriaDoc")

private String seriaDoc = null;

@JsonProperty("NumDoc")

private String numDoc = null;

@JsonProperty("Issuer")

private String issuer = null;

@JsonProperty("IssueDate")

private LocalDate issueDate = null;

public PersonDoc id(Long id) {

this.id = id;

return this;

}

/\*\*

\* Get id

\* @return id

\*\*/

@ApiModelProperty(value = "")

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public PersonDoc typeDoc(TypeDocEnum typeDoc) {

this.typeDoc = typeDoc;

return this;

}

/\*\*

\* PersonDoc type

\* @return typeDoc

\*\*/

@ApiModelProperty(required = true, value = "PersonDoc type")

public TypeDocEnum getTypeDoc() {

return typeDoc;

}

public void setTypeDoc(TypeDocEnum typeDoc) {

this.typeDoc = typeDoc;

}

public PersonDoc seriaDoc(String seriaDoc) {

this.seriaDoc = seriaDoc;

return this;

}

/\*\*

\* Get seriaDoc

\* @return seriaDoc

\*\*/

@ApiModelProperty(value = "")

public String getSeriaDoc() {

return seriaDoc;

}

public void setSeriaDoc(String seriaDoc) {

this.seriaDoc = seriaDoc;

}

public PersonDoc numDoc(String numDoc) {

this.numDoc = numDoc;

return this;

}

/\*\*

\* Get numDoc

\* @return numDoc

\*\*/

@ApiModelProperty(required = true, value = "")

public String getNumDoc() {

return numDoc;

}

public void setNumDoc(String numDoc) {

this.numDoc = numDoc;

}

public PersonDoc issuer(String issuer) {

this.issuer = issuer;

return this;

}

/\*\*

\* Get issuer

\* @return issuer

\*\*/

@ApiModelProperty(required = true, value = "")

public String getIssuer() {

return issuer;

}

public void setIssuer(String issuer) {

this.issuer = issuer;

}

public PersonDoc issueDate(LocalDate issueDate) {

this.issueDate = issueDate;

return this;

}

/\*\*

\* Get issueDate

\* @return issueDate

\*\*/

@ApiModelProperty(required = true, value = "")

public LocalDate getIssueDate() {

return issueDate;

}

public void setIssueDate(LocalDate issueDate) {

this.issueDate = issueDate;

}

@Override

public boolean equals(java.lang.Object o) {

if (this == o) {

return true;

}

if (o == null || getClass() != o.getClass()) {

return false;

}

PersonDoc personDoc = (PersonDoc) o;

return Objects.equals(this.id, personDoc.id) &&

Objects.equals(this.typeDoc, personDoc.typeDoc) &&

Objects.equals(this.seriaDoc, personDoc.seriaDoc) &&

Objects.equals(this.numDoc, personDoc.numDoc) &&

Objects.equals(this.issuer, personDoc.issuer) &&

Objects.equals(this.issueDate, personDoc.issueDate);

}

@Override

public int hashCode() {

return Objects.hash(id, typeDoc, seriaDoc, numDoc, issuer, issueDate);

}

@Override

public String toString() {

StringBuilder sb = new StringBuilder();

sb.append("class PersonDoc {\n");

sb.append(" id: ").append(toIndentedString(id)).append("\n");

sb.append(" typeDoc: ").append(toIndentedString(typeDoc)).append("\n");

sb.append(" seriaDoc: ").append(toIndentedString(seriaDoc)).append("\n");

sb.append(" numDoc: ").append(toIndentedString(numDoc)).append("\n");

sb.append(" issuer: ").append(toIndentedString(issuer)).append("\n");

sb.append(" issueDate: ").append(toIndentedString(issueDate)).append("\n");

sb.append("}");

return sb.toString();

}

/\*\*

\* Convert the given object to string with each line indented by 4 spaces

\* (except the first line).

\*/

private String toIndentedString(java.lang.Object o) {

if (o == null) {

return "null";

}

return o.toString().replace("\n", "\n ");

}

}

# <http://java-online.ru/jdbc-resultset.xhtml>

# Результаты запроса ResultSet

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

Доступ к данным **ResultSet** обеспечивает посредством набора get-методов, которые организуют доступ к колонкам текущей строки. Метод ResultSet.next используется для перемещения к следующей строке ResultSet, делая ее текущей.

## Методы ResultSet

Список наиболее часто используемых методов класса **ResultSet** представлен в таблице :

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| public boolean absolute(int row) throws SQLException | Метод перемещает курсор на заданное число строк от начала, если число положительно, и от конца - если отрицательно |
| public void afterLast() throws SQLException | Этот метод перемещает курсор в конец результирующего набора за последнюю строку |
| public void beforeFirst() throws SQLException | Этот метод перемещает курсор в начало результирующего набора перед первой строкой |
| public void deleteRow() throws SQLException | Удаляет текущую строку из результирующего набора и базы данных |
| public ResultSetMetaData getMetaData() throws SQLException | Предоставляет объект метаданных для данного ResultSet. Класс **ResultSetMetaData** содержит информацию о результирующие таблице, такую как количество столбцов, их заголовок и т.д. |
| public int getRow() throws SQLException | Возвращает номер текущей строки |
| public Statement getStatement() throws SQLException | Возвращает экземпляр Statement, который произвел данный результирующий набор |
| public boolean next() throws SQLException public boolean previous() throws SQLException | Эти методы позволяют переместиться в результирующем наборе на одну строку вперед или назад. Во вновь созданном результирующем наборе курсор устанавливается перед первой строкой, поэтому первое обращение к методу next() влечет позиционирование на первую строку. Эти методы возвращают true, если остается строка для дальнейшего перемещения. Если строк для обработки больше нет, возвращается false. Если открыт поток InputStream для предыдущей строки, он закрывается. Также очищается цепочка предупреждений SQLWarning |
| public void close() throws SQLException | Осуществляет немедленное закрытие ResultSet вручную. Обычно этого не требуется, так как закрытие Statement, связанного с ResultSet, автоматически закрывает ResultSet. К сожалению, не все разработчики JDBC-драйверов придерживаются этих концепций, например, драйвер Oracle самостоятельно не закрывает ResultSet'ы, так что настоятельно рекомендуется закрывать вручную |

## Пример использования ResultSet

package mssql;

import java.sql.\*;

public class Main

{

private static Connection con = null;

private static String username = "name";

private static String password = "pass";

private static String URL = "jdbc:jtds:sqlserver://localhost:1433";

public static void main(String[] args) throws SQLException

{

// Загрузка драйвера

DriverManager.registerDriver(new net.sourceforge.jtds.jdbc.Driver());

// Подключение к БД

con = DriverManager.getConnection(URL, username, password);

if(con != null)

System.out.println("Connection Successful !\n");

else

System.exit(0);

// Создание Statement для отправки запроса базе данных

Statement st = con.createStatement();

// Результирующий запрос

ResultSet rs = st.executeQuery("select \* from users");

// Количество колонок в результирующем запросе

int columns = rs.getMetaData().getColumnCount();

// Перебор строк с данными

while(rs.next()){

for (int i = 1; i <= columns; i++){

System.out.print(rs.getString(i) + "\t");

}

System.out.println();

}

System.out.println();

if(rs != null)

rs.close();

if(st != null)

st.close();

if(con != null)

con.close();

}

}

*Результирующий набор данных****ResultSet****можно не закрывать. Это делается автоматически родительским объектом [Statement](http://java-online.ru/jdbc-statement.xhtml), когда он закрывается, начинает выполняться повторно или используется для извлечения следующего результата в последовательности нескольких результатов. Но лучше все же закрывать и не надеяться, что это сделает разработчик драйвера JDBC.*

### Значение NULL в ResultSet

Чтобы определить, равно ли значение определенной колонки NULL или нет, необходимо сначала прочитать значение колонки, а затем использовать метод wasNull класса ResultSet для выяснения данного факта. Если wasNull возвращает значение true, то это означает, что считанное значение равно NULL.

В случае, если возвращаемое значение NULL, то методы ResultSet.getXXX, равны:

* значение будет null для тех методов getXXX, которые возвращают объекты (getString, getBigDecimal, getBytes, getDate, getTime, getTimestamp, getAsciiStream, getUnicodeStream, getBinaryStream, getObject)
* нулевое значение для методов, возвращающих целочисленное или вещественное значения (getByte, getShort, getInt, getLong, getFloat, and getDouble)
* false для метода getBoolean

## Строки, колонки и курсоры ResultSet

**ResultSet** содержит так называемый курсор, который позиционируется на текущей строке данных. При вызове метода **next**, курсор перемещается на следующую строку.

При открытии набора данных **ResultSet** курсор расположен перед первой строкой, и первый вызов next передвигает его на первую строку.

**ResultSet** хранит курсор до самого закрытия или пока не закроется родительский объект Statement.

Курсор для результирующей таблицы имеет имя. Если БД поддерживает позиционированные обновления или позиционированные удаления, то командам обновления или удаления можно передать в качестве параметра имя курсора, которое можно получить с помощью вызова метода getCursorName()

Statement stmt = connection.createStatement();

ResultSet rset = stmt.executeQuery ("select \* from users");

String cursorName = rs.getCursorName();

Но не все СУБД могут поддерживать позиционированные обновления или удаления. Чтобы узнать, поддерживает ли данное соединение Connection эти операции или нет, можно вызвать методы DatabaseMetaData.supportsPositionedDelete и supportsPositionedUpdate.

Методы **ResultSet.getXXX** предоставляют доступ к значениям в колонках в текущей строке. В пределах одной строки значения могут быть считаны в любом порядке. Для обеспечения бо́льшей совместимости рекомендуется считывать их подряд слева направо и делать это только один раз. Для указания колонки можно использовать либо ее имя, либо ее номер. Например, если вторая колонка объекта ResultSet rs называется "title" и хранит строковое значение, то извлечь его можно одним из двух способов:

String s = rs.getString("title");

String s = rs.getString(2);

При обращении к колонке по номеру следует помнить, что колонки нумеруются слева направо, начиная с 1, а имена колонок в вызове методов getXXX нечувствительны к регистру букв.

Наименования колонок совпадает с соответствующими наименованиями колонок в запросе. Если же в выражении select не указываются имена колонок (например "select \* from users"), то необходимо либо использовать номера колонок, либо "подключать" метаданные. Информацию о колонках в ResultSet можно получить с помощью вызова ResultSet.getMetaData. Возвращаемый объект ResultSetMetaData содержит информацию о количестве, типах и свойствах колонок объекта ResultSet.

В некоторых случаях имена двух колонок могут совпадать. Тогда при использовании имен колонок в методах getXXX возвращается значение первой подходящей колонки. Таким образом, чтобы считать значение других колонок с таким же именем, надо использовать индексы колонок. Кроме того, использование индексов немного эффективнее.

Если имя колонки известно, а индекс нет, то для поиска номера колонки можно использовать метод **findColumn()**.

## Типы данных и их преобразование

Различные методы чтения записей типа getXXX конвертируют низкоуровневые данные в типы данных Java. Например, если в таблице БД тип данных VARCHAR, то при использовании метода getString, драйвер JDBC конвертирует VARCHAR в объект String. Т.е. возвращаемым из метода getString значением будет объект String.

Следующая таблица показывает, какие типы данных различные методы getXXX могут считывать и какие JDBC-типы (SQL-типы) рекомендуются для этих методов.

Условные обозначения :

* "x" означает, что метод getXXX может быть использован,
* "X" означает, что данный метод рекомендуется использовать для этого типа данных.

Например, для типа данных LONGVARCHAR значение можно извлечь любым из методов getXXX кроме getBytes и getBinaryStream, но рекомендуется использовать методы getAsciiStream и getUnicodeStream.

Метод getObject возвращает значение как Object и может быть использован в тех случаях, когда соответствующий низкоуровневый тип данных является специфичным для данной СУБД, или когда приложению необходимо принять любой тип данных.

Таблица соответствия методов ResultSet.getXXX при чтении значений различных типам данных SQL.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **T I N Y I N T** | **S M A L L I N T** | **I N T E G E R** | **B I G I N T** | **R E A L** | **F L O A T** | **D O U B L E** | **D E C I M A L** | **N U M E R I C** | **B I T** | **C H A R** | **V A R C H A R** | **L O N G V A R C H A R** | **B I N A R Y** | **V A R B I N A R Y** | **L O N G V A R B I N A R Y** | **D A T E** | **T I M E** | **T I M E S T A M P** |
| getByte | **X** | x | x | x | x | x | x | x | x | x | x | x | x |  |  |  |  |  |  |
| getShort | x | **X** | x | x | x | x | x | x | x | x | x | x | x |  |  |  |  |  |  |
| getInt | x | x | **X** | x | x | x | x | x | x | x | x | x | x |  |  |  |  |  |  |
| getLong | x | x | x | **X** | x | x | x | x | x | x | x | x | x |  |  |  |  |  |  |
| getFloat | x | x | x | x | **X** | x | x | x | x | x | x | x | x |  |  |  |  |  |  |
| getDouble | x | x | x | x | x | **X** | **X** | x | x | x | x | x | x |  |  |  |  |  |  |
| getBigDecimal | x | x | x | x | x | x | x | **X** | **X** | x | x | x | x |  |  |  |  |  |  |
| getBoolean | x | x | x | x | x | x | x | x | x | **X** | x | x | x |  |  |  |  |  |  |
| getString | x | x | x | x | x | x | x | x | x | x | **X** | **X** | x | x | x | x | x | x | x |
| getBytes |  |  |  |  |  |  |  |  |  |  |  |  |  | **X** | **X** | x |  |  |  |
| getDate |  |  |  |  |  |  |  |  |  |  | x | x | x |  |  |  | **X** |  | x |
| getTime |  |  |  |  |  |  |  |  |  |  | x | x | x |  |  |  |  | **X** | x |
| getTimestamp |  |  |  |  |  |  |  |  |  |  | x | x | x |  |  |  | x |  | **X** |
| getAsciiStream |  |  |  |  |  |  |  |  |  |  | x | x | **X** | x | x | x |  |  |  |
| getUnicodeStream |  |  |  |  |  |  |  |  |  |  | x | x | **X** | x | x | x |  |  |  |
| getBinaryStream |  |  |  |  |  |  |  |  |  |  |  |  |  | x | x | **X** |  |  |  |
| getObject | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x |

## Чтение больших LOB объектов

Для чтения больших объектов **LOB** (Large Object Bynary) также используется ResultSet. Методы getBytes и getString возвращают эти данные в виде одного большого массива (байт, символов). Можно узнать размер объекта с помощью метода Statement.getMaxFieldSize.

byte[] binaryBuffer = null;

try {

Statement stmt = connection.createStatement();

ResultSet rset = stmt.executeQuery(sql);

rset.next();

binaryBuffer = rset.getBytes(1);

rset.close();

stmt.close();

} catch (SQLException e) {

System.err.println (e.getMessage());

}

Можно большие объекты **LOB** читать с помощью потоков (java.io.InputStream), которые возвращаются некоторыми методами ResultSet. Следует обратить внимание на то, что к этим потокам надо обращаться сразу, так как они будут закрыты при следующем вызове getXXX объекта ResultSet. Такое поведение диктуется низкоуровневой реализацией доступа к большим двоичным объектам.

JDBC API включает три отдельных метода для чтения данных в поток :

* getBinaryStream возвращает поток байтов "как есть", без какого-либо предварительного преобразования
* getAsciiStream возвращает поток, состоящий из однобайтовых ASCII-символов
* getUnicodeStream возвращает поток двухбайтных символов Unicode

Эти потоки отличаются от обычных потоков Java, которые возвращают нетипизированные байты.

Следующий пример демонстрирует использование getAsciiStream :

String sql = "select book from lib where name = 'Золушка'";

Statement stmt = connection.createStatement();

ResultSet rs = st.executeQuery(sql);

rs.next();

Clob clob = rs.getClob(1);

InputStream is = clob.getAsciiStream();

Пример записи файлов в бинарные (BLOB) и символьные (CLOB/TEXT) поля баз данных Oracle и MySQL можно увидеть [здесь](http://java-online.ru/jdbc-lob.xhtml).

## Множественные наборы : getResultSet, getUpdateCount, getMoreResults

Обычно при выполнении SQL-запросов используют либо метод **executeQuery**, возвращающий единственный **ResultSet**, либо **executeUpdate**, который может быть использован для изменения значения в таблице БД и который возвращают количество измененных строк. Тем не менее, в отдельных случаях приложению заранее может быть неизвестно, возвратит ли данный запрос результат или нет. Кроме этого, некоторые хранимые процедры могут возвратить несколько наборов данных и/или счетчиков обновления.

Для этого случая в JDBC есть механизм, когда приложение может обрабатывать произвольную коллекцию наборов результатов или счетчиков обновления. Данный механизм основан на вызове метода **execute** и последующем вызове трех других методов **getResultSet**, getUpdateCount и **getMoreResults**.

Методы **getResultSet**, **getUpdateCount** и getMoreResults позволяют приложению получать результаты запроса по-очереди и для каждого результата определять, является ли он набором данных или счетчиком обновлений.