JDBC

Java™ Database Connectivity (JDBC) es la especificación JavaSoft de una interfaz de programación de aplicaciones (API) estándar que permite que los programas Java accedan a sistemas de gestión de bases de datos. La API JDBC consiste en un conjunto de interfaces y clases escribas en el lenguaje de programación Java.
Con estas interfaces y clases estándar, los programadores pueden escribir aplicaciones que conecten con bases de datos, envíen consultas escritas en el lenguaje de consulta estructurada (SQL) y procesen los resultados.
Puesto que JDBC es una especificación estándar, un programa Java que utilice la API JDBC puede conectar con cualquier sistema de gestión de bases de datos (DBMS), siempre y cuando haya un controlador para dicho DBMS en concreto.

¿POR QUÉ JDBC?

La API JDBC es una API de Java que puede acceder a cualquier tipo de datos tabulares, especialmente a los datos almacenados en una base de datos relacional.
JDBC le ayuda a escribir aplicaciones Java que gestionan estas tres actividades de programación:

1. Conectarse a una fuente de datos, como una base de datos
2. Enviar consultas y actualizar declaraciones a la base de datos.
3. Recuperar y procesar los resultados recibidos de la base de datos en respuesta a su consulta

DRIVER MANAGER

La API JDBC define las interfaces y clases Java™ que utilizan los programadores para conectar con bases de datos y enviar consultas. Un controlador JDBC implementa dichas interfaces y clases para un determinado proveedor de DBMS.
Un programa Java que utiliza la API JDBC carga el controlador especificado para el DBMS particular antes de conectar realmente con una base de datos. Luego la clase JDBC **DriverManager** envía todas las llamadas de la API JDBC al controlador cargado.
Hay cuatro tipos de controladores JDBC:

- **Puente JDBC-ODBC más controlador ODBC, también denominado controlador de Tipo 1**

Convierte las llamadas de la API JDBC en llamadas de Microsoft ODBC que luego se pasan al controlador ODBC
El código binario ODBC se debe cargar en cada sistema cliente que utilice este tipo de controlador.
ODBC es el acrónimo de Open Database Connectivity.

- **API nativa, en parte controlador Java, también denominado controlador de Tipo 2**

Convierte las llamadas de la API JDBC en llamadas de API de cliente específicas de DBMS. Al igual que el controlador puente, este tipo de controlador necesita que se cargue cierto código binario en cada sistema cliente.

- **JDBC-Net, controlador Java puro, también denominado controlador de Tipo 3**

Envía las llamadas de la API JDBC a un servidor de nivel medio que convierte las llamadas al protocolo de red específico de DBMS
Luego las llamadas convertidas se envían a un determinado DBMS.

- **Protocolo nativo, controlador Java puro, también denominado controlador de Tipo 4**

Convierte las llamadas de la API JDBC directamente al protocolo de red específico de DBMS sin un nivel medio. Este controlador permite a las aplicaciones cliente conectar directamente con el servidor de bases de datos.

CONNECTIONS

Establishing a Connection

First, you need to establish a connection with the data source you want to use. A data source can be a DBMS, a legacy file system, or some other source of data with a corresponding JDBC driver. Typically, a JDBC application connects to a target data source using one of two classes:

- `DriverManager`: This fully implemented class connects an application to a data source, which is specified by a database URL. When this class first attempts to establish a connection, it automatically loads any JDBC 4.0 drivers found within the class path. Note that your application must manually load any JDBC drivers prior to version 4.0.
- `DataSource`: This interface is preferred over `DriverManager` because it allows details about the underlying data source to be transparent to your application. A `DataSource` object's properties are set so that it represents a particular data source.

Using the DriverManager Class

Connecting to your DBMS with the `DriverManager` class involves calling the method `DriverManager.getConnection`. The following method, `JDBCTutorialUtilities.getConnection`, establishes a database connection:

```
public Connection getConnection() throws SQLException {

    Connection conn = null;
    Properties connectionProps = new Properties();
    connectionProps.put("user", this.userName);
    connectionProps.put("password", this.password);

    if (this.dbms.equals("mysql")) {
        conn = DriverManager.getConnection(
                   "jdbc:" + this.dbms + "://" +
                   this.serverName +
                   ":" + this.portNumber + "/",
                   connectionProps);
    } else if (this.dbms.equals("derby")) {
        conn = DriverManager.getConnection(
                   "jdbc:" + this.dbms + ":" +
                   this.dbName +
                   ";create=true",
                   connectionProps);
    }
    System.out.println("Connected to database");
    return conn;
}
```

The method `DriverManager.getConnection` establishes a database connection. This method requires a database URL, which varies depending on your DBMS. The following are some examples of database URLs:

1. MySQL: `jdbc:mysql://localhost:3306/`, where `localhost` is the name of the server hosting your database, and `3306` is the port number

2. Java DB: `jdbc:derby:`*`testdb`*`;create=true`, where *`testdb`* is the name of the database to connect to, and `create=true` instructs the DBMS to create the database.

   Note: This URL establishes a database connection with the Java DB Embedded Driver. Java DB also includes a Network Client Driver, which uses a different URL.

This method specifies the user name and password required to access the DBMS with a `Properties` object.

Note:

- Typically, in the database URL, you also specify the name of an existing database to which you want to connect. For example, the URL `jdbc:mysql://localhost:3306/mysql` represents the database URL for the MySQL database named `mysql`. The samples in this tutorial use a URL that does not specify a specific database because the samples create a new database.
- In previous versions of JDBC, to obtain a connection, you first had to initialize your JDBC driver by calling the method `Class.forName`. This methods required an object of type `java.sql.Driver`. Each JDBC driver contains one or more classes that implements the interface `java.sql.Driver`. The drivers for Java DB are `org.apache.derby.jdbc.EmbeddedDriver` and `org.apache.derby.jdbc.ClientDriver`, and the one for MySQL Connector/J is `com.mysql.cj.jdbc.Driver`. See the documentation of your DBMS driver to obtain the name of the class that implements the interface `java.sql.Driver`.

  Any JDBC 4.0 drivers that are found in your class path are automatically loaded. (However, you must manually load any drivers prior to JDBC 4.0 with the method `Class.forName`.)

The method returns a `Connection` object, which represents a connection with the DBMS or a specific database. Query the database through this object.

Specifying Database Connection URLs

A database connection URL is a string that your DBMS JDBC driver uses to connect to a database. It can contain information such as where to search for the database, the name of the database to connect to, and configuration properties. The exact syntax of a database connection URL is specified by your DBMS.

Java DB Database Connection URLs

The following is the database connection URL syntax for Java DB:

```
jdbc:derby:[subsubprotocol:][databaseName][;attribute=value]*
```

- *`subsubprotocol`* specifies where Java DB should search for the database, either in a directory, in memory, in a class path, or in a JAR file. It is typically omitted.
- *`databaseName`* is the name of the database to connect to.
- *`attribute=value`* represents an optional, semicolon-separated list of attributes. These attributes enable you to instruct Java DB to perform various tasks, including the following:
  - Create the database specified in the connection URL.
  - Encrypt the database specified in the connection URL.
  - Specify directories to store logging and trace information.
  - Specify a user name and password to connect to the database.

MySQL Connector/J Database URL

The following is the database connection URL syntax for MySQL Connector/J:

```
jdbc:mysql://[host][,failoverhost...]
     [:port]/[database]
     [?propertyName1][=propertyValue1]
     [&propertyName2][=propertyValue2]...
```

- `host:port` is the host name and port number of the computer hosting your database. If not specified, the default values of `host` and `port` are 127.0.0.1 and 3306, respectively.
- `database` is the name of the database to connect to. If not specified, a connection is made with no default database.
- `failover` is the name of a standby database (MySQL Connector/J supports failover).
- `propertyName=propertyValue` represents an optional, ampersand-separated list of properties. These attributes enable you to instruct MySQL Connector/J to perform various tasks.

Examples of connections

DB	URL de conexión JDBC
Firebird	`jdbc:firebirdsql://`**`<host>`**`[:`**`<puerto>`**`]/`**`<ruta de acceso o alias de la BD>`**
IBM DB2	`jdbc:db2://`**`nombreHost`**`:`**`puerto`**`/`**`nombreBaseDatos`**
IBM DB2 for i	`jdbc:as400://`**`[host]`**
IBM Informix	`jdbc:informix-sqli://`**`nombreHost`**`:`**`puerto`**`/`**`nombreBaseDatos`**`:INFORMIXSERVER=`**`miservidor`**
MariaDB	`jdbc:mariadb://`**`nombreHost`**`:`**`puerto`**`/`**`nombreBaseDatos`**
Microsoft SQL Server	`jdbc:sqlserver://`**`nombreHost`**`:`**`puerto`**`;`**`nombreBaseDatos`**`=name`
MySQL	`jdbc:mysql://`**`nombreHost`**`:`**`puerto`**`/`**`nombreBaseDatos`**

Oracle	`jdbc:oracle:thin:@`**`nombreHost`**`:`**`puerto`**`:`**`SID`**
	`jdbc:oracle:thin:@//`**`nombreHost`**`:`**`puerto`**`:`**`servicio`**

Oracle XML DB	`jdbc:oracle:oci:@//`**`nombreHost`**`:`**`puerto`**`:`**`servicio`**
PostgreSQL	`jdbc:postgresql://`**`nombreHost`**`:`**`puerto`**`/`**`nombreBaseDatos`**
Progress OpenEdge	`jdbc:datadirect:openedge://`**`host`**`:`**`puerto`**`;databaseName=`**`nombre_bd`**
Sybase	`jdbc:sybase:Tds:`**`nombreHost`**`:`**`puerto`**`/`**`nombreBaseDatos`**
Teradata	`jdbc:teradata://`**`nombreServidorBaseDatos`**

Processing SQL Statements with JDBC

In general, to process any SQL statement with JDBC, you follow these steps:

1. Establishing a connection.
2. Create a statement.
3. Execute the query.
4. Process the `ResultSet` object.
5. Close the connection.

This method outputs the contents of the table `COFFEES`. This method will be discussed in more detail later in this tutorial:

```
  public static void viewTable(Connection con) throws SQLException {
    String query = "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from
COFFEES";
    try (Statement stmt = con.createStatement()) {
      ResultSet rs = stmt.executeQuery(query);
      while (rs.next()) {
        String coffeeName = rs.getString("COF_NAME");
        int supplierID = rs.getInt("SUP_ID");
        float price = rs.getFloat("PRICE");
        int sales = rs.getInt("SALES");
        int total = rs.getInt("TOTAL");
        System.out.println(coffeeName + ", " + supplierID + ", " + price
+ ", " + sales + ", " + total);
      }
    } catch (SQLException e) {
      JDBCTutorialUtilities.printSQLException(e);
    }
  }
```

Establishing Connections

First, establish a connection with the data source you want to use. A data source can be a DBMS, a legacy file system, or some other source of data with a corresponding JDBC driver. This connection is represented by a `Connection` object.

Creating Statements

A `Statement` is an interface that represents a SQL statement. You execute `Statement` objects, and they generate `ResultSet` objects, which is a table of data representing a database result set. You need a `Connection` object to create a `Statement` object.

For example, `CoffeesTable.viewTable` creates a `Statement` object with the following code:

```
stmt = con.createStatement();
```

There are three different kinds of statements:

- `Statement`: Used to implement simple SQL statements with no parameters.
- `PreparedStatement`: (Extends `Statement`.) Used for precompiling SQL statements that might contain input parameters. See Using Prepared Statements for more information.

- `CallableStatement`: (Extends `PreparedStatement`.) Used to execute stored procedures that may contain both input and output parameters. See Stored Procedures for more information.

Executing Queries

To execute a query, call an `execute` method from `Statement` such as the following:

- `execute`: Returns `true` if the first object that the query returns is a `ResultSet` object. Use this method if the query could return one or more `ResultSet` objects. Retrieve the `ResultSet` objects returned from the query by repeatedly calling `Statement.getResultSet`.
- `executeQuery`: Returns one `ResultSet` object.
- `executeUpdate`: Returns an integer representing the number of rows affected by the SQL statement. Use this method if you are using `INSERT`, `DELETE`, or `UPDATE` SQL statements.

For example, `CoffeesTable.viewTable` executed a `Statement` object with the following code:

```
ResultSet rs = stmt.executeQuery(query);
```

Processing ResultSet Objects

You access the data in a `ResultSet` object through a cursor. Note that this cursor is not a database cursor. This cursor is a pointer that points to one row of data in the `ResultSet` object. Initially, the cursor is positioned before the first row. You call various methods defined in the `ResultSet` object to move the cursor. For example, `CoffeesTable.viewTable` repeatedly calls the method `ResultSet.next` to move the cursor forward by one row. Every time it calls `next`, the method outputs the data in the row where the cursor is currently positioned:

```
      ResultSet rs = stmt.executeQuery(query);
      while (rs.next()) {
        String coffeeName = rs.getString("COF_NAME");
        int supplierID = rs.getInt("SUP_ID");
        float price = rs.getFloat("PRICE");
        int sales = rs.getInt("SALES");
        int total = rs.getInt("TOTAL");
        System.out.println(coffeeName + ", " + supplierID + ", " + price
+ ", " + sales + ", " + total);
      }
      // ...
```

Closing Connections

When you are finished using a `Connection`, `Statement`, or `ResultSet` object, call its `close` method to immediately release the resources it's using.

Alternatively, use a `try`-with-resources statement to automatically close `Connection`, `Statement`, and `ResultSet` objects, regardless of whether an `SQLException` has been thrown. (JDBC throws an `SQLException` when it encounters an error during an interaction with a data source. See Handling SQL Exceptions for more information.) An automatic resource statement consists of a `try` statement and one or more declared resources. For example, the `CoffeesTable.viewTable` method automatically closes its `Statement` object, as follows:

```
public static void viewTable(Connection con) throws SQLException {
    String query = "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from
COFFEES";
    try (Statement stmt = con.createStatement()) {
      ResultSet rs = stmt.executeQuery(query);
      while (rs.next()) {
        String coffeeName = rs.getString("COF_NAME");
        int supplierID = rs.getInt("SUP_ID");
        float price = rs.getFloat("PRICE");
        int sales = rs.getInt("SALES");
        int total = rs.getInt("TOTAL");
        System.out.println(coffeeName + ", " + supplierID + ", " + price
+ ", " + sales + ", " + total);
      }
    } catch (SQLException e) {
      JDBCTutorialUtilities.printSQLException(e);
    }
  }
```

The following statement is a `try`-with-resources statement, which declares one resource, `stmt`, that will be automatically closed when the `try` block terminates:

```
    try (Statement stmt = con.createStatement()) {
      // ...
    }
```

INSERTING ROWS

Creating, reading, updating, and deleting data in a database is a common task in many applications, and JDBC (Java Database Connectivity) is a Java API that allows you to connect to a database and perform these operations.g

The first step is to establish a connection to the database. You can do this by loading the JDBC driver and creating a connection object.

```
try {
      Class.forName("com.mysql.jdbc.Driver");
      Connection con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/mydb", "username",
            "password");
      System.out.println("Connection established.");
}
catch (Exception e) {
      e.printStackTrace();
}
```

Once you have a connection to the database, you can use the connection object to create a new record in the database. To do this, you will need to use an SQL INSERT statement and execute it using the connection object.

```
try {
      String sql = "INSERT INTO table_name (column1, column2, column3)
      VALUES (?, ?, ?)";
      PreparedStatement statement = con.prepareStatement(sql);
      statement.setString(1, "value1");
      statement.setString(2, "value2");
      statement.setInt(3, 123);
      statement.executeUpdate();
      System.out.println("Record created.");
} catch (SQLException e) {
      e.printStackTrace();
}
```

To read a record from the database, you will need to use an SQL SELECT statement and execute it using the connection object. The result of the query will be a ResultSet object that you can use to access the data in the record.

```
try {
      String sql = "SELECT column1, column2, column3 FROM table_name
      WHERE id = ?";
      PreparedStatement statement = con.prepareStatement(sql);
      statement.setInt(1, 1);
      ResultSet result = statement.executeQuery();
if (result.next()) {
      String column1 = result.getString("column1");
      String column2 = result.getString("column2");
      int column3 = result.getInt("column3");
      System.out.println("Column 1: " + column1);
      System.out.println("Column 2: " + column2);
      System.out.println("Column 3: " + column3);
}
} catch (SQLException e) {
      e.printStackTrace();
}
```

UPDATING ROWS

To update a record in the database, you will need to use an SQL UPDATE statement and execute it using the connection object.

```
try {
      String sql = "UPDATE table_name SET column1 = ?, column2 = ?,
      column3 = ? WHERE id = ?";
      PreparedStatement statement = con.prepareStatement(sql);
      statement.setString(1, "new_value1");
      statement.setString(2, "new_value2");
      statement.setInt(3, 456);
      statement.setInt(4, 1);
      statement.executeUpdate();
      System.out.println("Record updated.");
} catch (SQLException e) {
      e.printStackTrace();
}
```

DELETING ROWS

To delete a record from the database, you will need to use an SQL DELETE statement and execute it using the connection object.

```
try {
      String sql = "DELETE FROM table_name WHERE id = ?";
      PreparedStatement statement = con.prepareStatement(sql);
      statement.setInt(1, 1);
      statement.executeUpdate();
      System.out.println("Record deleted.");
} catch (SQLException e) {
      e.printStackTrace(); }
```

MAPPING BETWEEN SQL & JAVA DATA TYPES

Constraints

We need to provide reasonable Java mappings for the common SQL data types. We also need to make sure that we have enough type information so that we can correctly store and retrieve parameters and recover results from SQL statements.
However, there is no particular reason that the Java data type needs to be exactly isomorphic to the SQL data type. For example, since Java has no fixed length arrays, we can represent both fixed length and variable length SQL arrays as variable length Java arrays. We also felt free to use Java Strings even though they don't precisely match any of the SQL CHAR types.
Table 2 shows the default Java mapping for various common SQL data types. Not all of these types will necessarily be supported by all databases. The various mappings are described more fully in the following sections.
Table 2: Standard mapping from SQL types to Java types.

SQL type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Similarly table 3 shows the reverse mapping from Java types to SQL types.

Table 3: Standard mapping from Java types to SQL types.

The mapping for String will normally be VARCHAR but will turn into LONGVARCHAR if the given value exceeds the drivers limit on VARCHAR values. Similarly for byte[] and VARBINARY and LONGVARBINARY.

Java Type	SQL type
String	VARCHAR or LONGVARCHAR
java.math.BigDecimal	NUMERIC
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
byte[]	VARBINARY or LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP

Dynamic data access

This chapter focuses on access to results or parameters whose types are known at compile time. However, some applications, for example generic browsers or query tools, are not compiled with knowledge of the database schema they will access, so JDBC also provides support for fully dynamically typed data access. See Section 14.2.

CHAR, VARCHAR, and LONGVARCHAR

There is no need for Java programmers to distinguish among the three different flavours of SQL strings CHAR, VARCHAR, and LONGVARCHAR. These can all be expressed identically in Java. It is possible to read and write the SQL correctly without needing to know the exact data type that was expected.

These types could be mapped to either String or char[]. After considerable discussion we decided to use String, as this seemed the more appropriate type for normal use. Note that the Java String class provides a method for converting a String to a char[] and a constructor for turning a char[] into a String.

For fixed length SQL strings of type CHAR(n), the JDBC drivers will perform appropriate padding with spaces. Thus when a CHAR(n) field is retrieved from the database the resulting String will always be of length "n" and may include some padding spaces at the end. When a String is sent to a CHAR(n) field, the driver and/or the database will add any necessary padding spaces to the end of the String to bring it up to length "n".

The ResultSet.getString method allocates and returns a new String. This is suitable for retrieving normal data, but the LONGVARCHAR SQL type can be used to store multi-megabyte strings. We therefore needed to provide a way for Java programmers to retrieve a LONGVARCHAR value in chunks. We handle this by allowing programmers to retrieve a LONGVARCHAR as a Java input stream from which they can subsequently read data in whatever chunks they prefer. Java streams can be used for either Unicode or Ascii data, so the programmer may chose to use either getAsciiStream or getUnicodeStream.

DECIMAL and NUMERIC

The SQL DECIMAL and NUMERIC data types are used to express fixed point numbers where absolute precision is required. They are often used for currency values.

These two types can be expressed identically in Java. The most convenient mapping uses the java.math.BigDecimal extended precision number type provided in JDK1.1

We also allow access to DECIMAL and NUMERIC as simple Strings and arrays of chars. Thus Java programmers can use getString to receive a NUMERIC or DECIMAL result.

BINARY, VARBINARY, and LONGVARBINARY

There is no need for Java programmers to distinguish among the three different flavours of SQL byte arrays BINARY, VARBINARY, and LONGVARBINARY. These can all be expressed identically as byte arrays in Java. (It is possible to read and write the SQL correctly without needing to know the exact BINARY data type that was expected.)

As with the LONGVARCHAR SQL type, the LONGVARBINARY SQL type can sometimes be used to return multi-megabyte data values. We therefore allow a LONGVARBINARY value to be retrieved as a Java input stream, from which programmers can subsequently read data in whatever chunks they prefer.

BIT

The SQL BIT type can be mapped directly to the Java boolean type.

TINYINT, SMALLINT, INTEGER, and BIGINT

The SQL TINYINT, SMALLINT, INTEGER, and BIGINT types represent 8 bit, 16 bit, 32 bit, and 64 bit values. These therefore can be mapped to Java's byte, short, int, and long data types.

REAL, FLOAT, and DOUBLE

SQL defines three floating point data types, REAL, FLOAT, and DOUBLE.

We map REAL to Java float, and FLOAT and DOUBLE to Java double.

REAL is required to support 7 digits of mantissa precision. FLOAT and DOUBLE are required to support 15 digits of mantissa precision.

DATE, TIME, and TIMESTAMP

SQL defines three time related data types. DATE consists of day, month, and year. TIME consists of hours, minutes and seconds. TIMESTAMP combines DATE and TIME and also adds in a nanosecond field.

There is a standard Java class java.util.Date that provides date and time information. However, this class doesn't perfectly match any of the three SQL types, as it includes both DATE and TIME information, but lacks the nanosecond granularity required for TIMESTAMP.

We therefore define three subclasses of java.util.Date. These are:

- java.sql.Date for SQL DATE information
- java.sql.Time for SQL TIME information
- java.sql.Timestamp for SQL TIMESTAMP information

In the case of java.sql.Date the hour, minute, second, and milli-second fields of the java.util.Date base class are set to zero.

In the case of java.sql.Time the year, month, and day fields of the java.util.Date base class are set to 1970, January, and 1, respectively. This is the "zero" date in the Java epoch.

The java.sql.Timestamp class extends java.util.Date by adding a nanosecond field.

SQLException

Overview of SQLException

When JDBC encounters an error during an interaction with a data source, it throws an instance of `SQLException` as opposed to `Exception`. (A data source in this context represents the database to which a `Connection` object is connected.) The `SQLException` instance contains the following information that can help you determine the cause of the error:

- A description of the error. Retrieve the `String` object that contains this description by calling the method `SQLException.getMessage`.
- A SQLState code. These codes and their respective meanings have been standardized by ISO/ANSI and Open Group (X/Open), although some codes have been reserved for database vendors to define for themselves. This `String` object consists of five alphanumeric characters. Retrieve this code by calling the method `SQLException.getSQLState`.
- An error code. This is an integer value identifying the error that caused the `SQLException` instance to be thrown. Its value and meaning are implementation-specific and might be the actual error code returned by the underlying data source. Retrieve the error by calling the method `SQLException.getErrorCode`.
- A cause. A `SQLException` instance might have a causal relationship, which consists of one or more `Throwable` objects that caused the `SQLException` instance to be thrown. To navigate this chain of causes, recursively call the method `SQLException.getCause` until a `null` value is returned.
- A reference to any *chained* exceptions. If more than one error occurs, the exceptions are referenced through this chain. Retrieve these exceptions by calling the method `SQLException.getNextException` on the exception that was thrown.

Retrieving Exceptions

The following method, `JDBCTutorialUtilities.printSQLException`, outputs the SQLState, error code, error description, and cause (if there is one) contained in the `SQLException` as well as any other exception chained to it:

```
public static void printSQLException(SQLException ex) {

    for (Throwable e : ex) {
        if (e instanceof SQLException) {
            if (ignoreSQLException(
                ((SQLException)e).
                getSQLState()) == false) {

                e.printStackTrace(System.err);
                System.err.println("SQLState: " +
                    ((SQLException)e).getSQLState());

                System.err.println("Error Code: " +
                    ((SQLException)e).getErrorCode());

                System.err.println("Message: " + e.getMessage());

                Throwable t = ex.getCause();
                while(t != null) {
                    System.out.println("Cause: " + t);
                    t = t.getCause();
                    }
                }
            }
        }
    }
```

For example, if you call the method `CoffeesTable.dropTable` with Java DB as your DBMS, the table `COFFEES` does not exist, *and* you remove the call to `JDBCTutorialUtilities.ignoreSQLException`, the output will be similar to the following:

```
SQLState: 42Y55
Error Code: 30000
Message: 'DROP TABLE' cannot be performed on
'TESTDB.COFFEES' because it does not exist.
```

Instead of printing `SQLException` information, you could instead first retrieve the `SQLState` then process the `SQLException` accordingly. For example, the method `JDBCTutorialUtilities.ignoreSQLException` returns `true` if the `SQLState` is equal to code `42Y55` (and you are using Java DB as your DBMS), which causes `JDBCTutorialUtilities.printSQLException` to ignore the `SQLException`:

```
public static boolean ignoreSQLException(String sqlState) {

    if (sqlState == null) {
        System.out.println("The SQL state is not defined!");
        return false;
    }

    // X0Y32: Jar file already exists in schema
    if (sqlState.equalsIgnoreCase("X0Y32"))
        return true;

    // 42Y55: Table already exists in schema
    if (sqlState.equalsIgnoreCase("42Y55"))
        return true;

    return false;
}
```

Retrieving Warnings

`SQLWarning` objects are a subclass of `SQLException` that deal with database access warnings. Warnings do not stop the execution of an application, as exceptions do; they simply alert the user that something did not happen as planned. For example, a warning might let you know that a privilege you attempted to revoke was not revoked. Or a warning might tell you that an error occurred during a requested disconnection.

A warning can be reported on a `Connection` object, a `Statement` object (including `PreparedStatement` and `CallableStatement` objects), or a `ResultSet` object. Each of these classes has a `getWarnings` method, which you must invoke in order to see the first warning reported on the calling object. If `getWarnings` returns a warning, you can call the `SQLWarning` method `getNextWarning` on it to get any additional warnings. Executing a statement automatically clears the warnings from a previous statement, so they do not build up. This means, however, that if you want to retrieve warnings reported on a statement, you must do so before you execute another statement.

The following methods from `JDBCTutorialUtilities.java` illustrate how to get complete information about any warnings reported on `Statement` or `ResultSet` objects:

```
public static void getWarningsFromResultSet(ResultSet rs)
    throws SQLException {
    JDBCTutorialUtilities.printWarnings(rs.getWarnings());
}

public static void getWarningsFromStatement(Statement stmt)
    throws SQLException {
    JDBCTutorialUtilities.printWarnings(stmt.getWarnings());
}

public static void printWarnings(SQLWarning warning)
    throws SQLException {

    if (warning != null) {
        System.out.println("\n---Warning---\n");

    while (warning != null) {
        System.out.println("Message: " + warning.getMessage());
        System.out.println("SQLState: " + warning.getSQLState());
        System.out.print("Vendor error code: ");
        System.out.println(warning.getErrorCode());
        System.out.println("");
        warning = warning.getNextWarning();
    }
}
```

The most common warning is a `DataTruncation` warning, a subclass of `SQLWarning`. All `DataTruncation` objects have a SQLState of `01004`, indicating that there was a problem with reading or writing data. `DataTruncation` methods let you find out in which column or parameter data was truncated, whether the truncation was on a read or write operation, how many bytes should have been transferred, and how many bytes were actually transferred.

Categorized SQLExceptions

Your JDBC driver might throw a subclass of `SQLException` that corresponds to a common SQLState or a common error state that is not associated with a specific SQLState class value. This enables you to write more portable error-handling code. These exceptions are subclasses of one of the following classes:

- `SQLNonTransientException`
- `SQLTransientException`
- `SQLRecoverableException`

See the latest Javadoc of the `java.sql` package or the documentation of your JDBC driver for more information about these subclasses.

Other Subclasses of SQLException

The following subclasses of `SQLException` can also be thrown:

- `BatchUpdateException` is thrown when an error occurs during a batch update operation. In addition to the information provided by `SQLException`, `BatchUpdateException` provides the update counts for all statements that were executed before the error occurred.
- `SQLClientInfoException` is thrown when one or more client information properties could not be set on a Connection. In addition to the information provided by `SQLException`, `SQLClientInfoException` provides a list of client information properties that were not set.

Referencias

https://www.altova.com/manual/es/DatabaseSpy/databasespyprofessional/dbc_creating_a_jdbc_connection.html#:~:text=JDBC%20(Java%20Database%20Connectivity)%20es,pero%20pueden%20ofrecer%20m%C3%A1s%20caracter%C3%ADsticas.

https://docs.oracle.com/javase/tutorial/jdbc/basics/connecting.html

https://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html

https://www.geeksforgeeks.org/simplifying-crud-operation-with-jdbc/

https://nick-lab.gs.washington.edu/java/jdk1.3.1/guide/jdbc/spec/jdbc-spec.frame8.html

https://docs.oracle.com/javase/tutorial/jdbc/basics/sqlexception.html