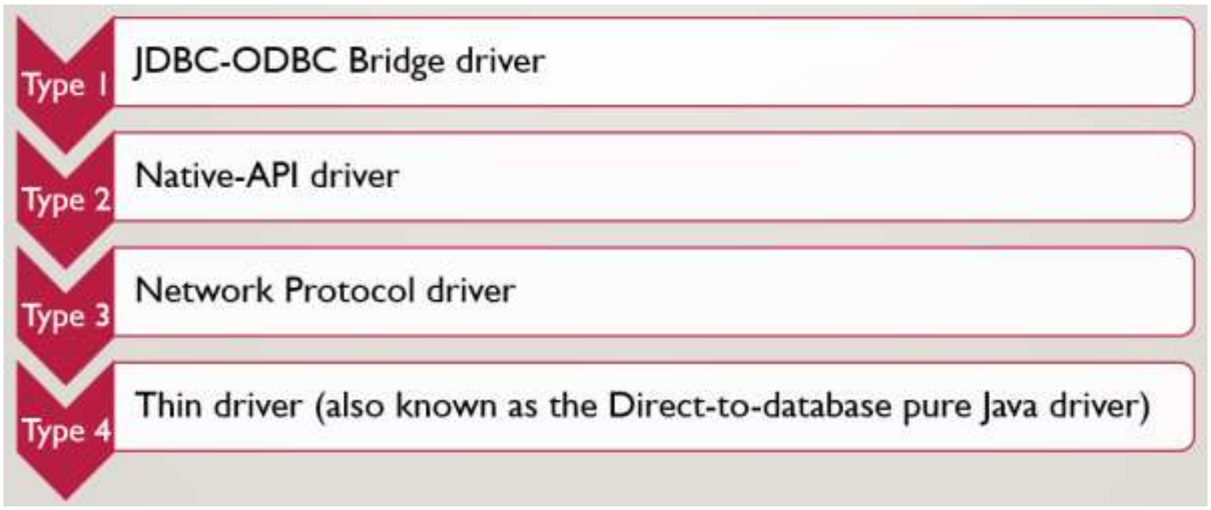
Introduction to JDBC

- JDBC stands for Java Database Connectivity.
- It's a Java API (Application Programming Interface) that enables Java applications to interact with databases.
- It allows them to perform various database operations like querying, updating, inserting, and deleting data.
- Java API provides a set of interfaces and classes for Java developers to work with when dealing with databases.

JDBC Driver

- JDBC requires a driver to connect to different types of databases.
- A JDBC driver is a software component that provides an interface for Java applications to interact with a specific type of database.

Types of JDBC drivers



Drivers (Type I)

Type I driver provides mapping between JDBC and access API of a database

- The access API calls the native API of the database to establish communication

A common Type I driver defines a JDBC to ODBC bridge

- ODBC is the database connectivity for databases
- JDBC driver translates JDBC calls to corresponding ODBC calls
- Thus if ODBC driver exists for a database this bridge can be used to communicate with the database from a Java application

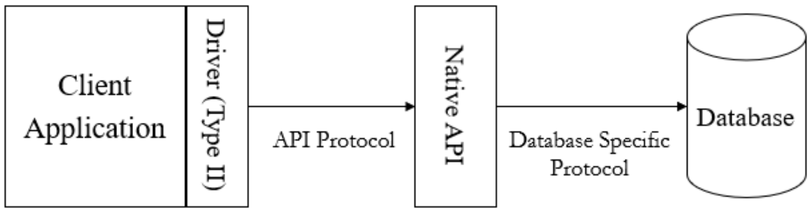
Inefficient and narrow solution

- Inefficient, because it goes through multiple layers
- Narrow, since functionality of JDBC code limited to whatever ODBC supports

Drivers (Type II)

Type II driver communicates directly with native API

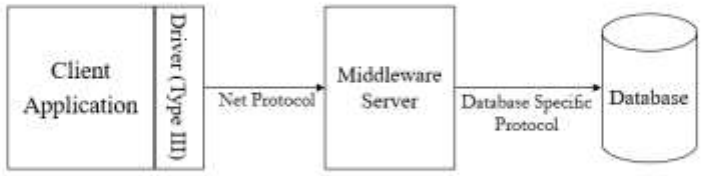
- Type II makes calls directly to the native API calls
- More efficient since there is one less layer to contend with (i.e. no ODBC)
- It is dependent on the existence of a native API for a database



Drivers (Type III)

Type III driver make calls to a middleware component running on another server

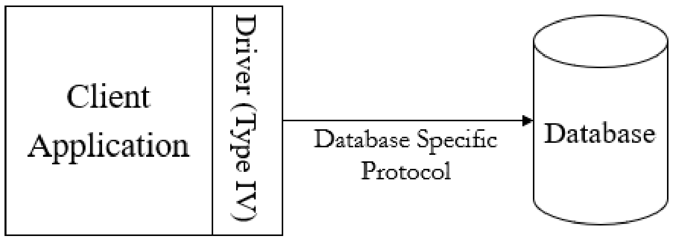
- This communication uses a database independent net protocol
- Middleware server then makes calls to the database using database-specific protocol
- The program sends JDBC call through the JDBC driver to the middle tier
- Middle-tier may use Type I or II JDBC driver to communicate with the database.



Drivers (Type IV)

Type IV driver is an all-Java driver that is also called a thin driver

- It issues requests directly to the database using its native protocol
- It can be used directly on platform with a JVM
- Most efficient since requests only go through one layer
- Simplest to deploy since no additional libraries or middle-ware



Introduction to JDBC-Part 2

Establishing Connection

- To establish a connection to a database using JDBC, you typically need to provide connection parameters such as database URL, username, and password.

Database URL:

- This is a string that specifies the location and name of the database.
- It typically includes information such as the database type (e.g., MySQL, PostgreSQL), hostname or IP address of the database server, port number, and database name.

Username:

- The username is the identifier used to authenticate the user's access to the database.

- It represents the user's identity and determines the permissions and privileges they have within the database management system.

Password:

- The password is a secret string of characters associated with the username.
- It serves as a form of authentication to verify the identity of the user attempting to connect to the database.

Creating Statements

After establishing a connection, you can create SQL statements using JDBC. There are mainly two types of statements:

Statement:

- Used for executing static SQL queries.

PreparedStatement:

- Used for executing parameterized SQL queries. It's precompiled and allows you to reuse the same SQL statement with different parameters.

Executing Queries

- Once you have created a statement, you can execute SQL queries to retrieve data from the database.
- JDBC supports various types of SQL queries like SELECT, INSERT, UPDATE, DELETE, etc.

Processing Results

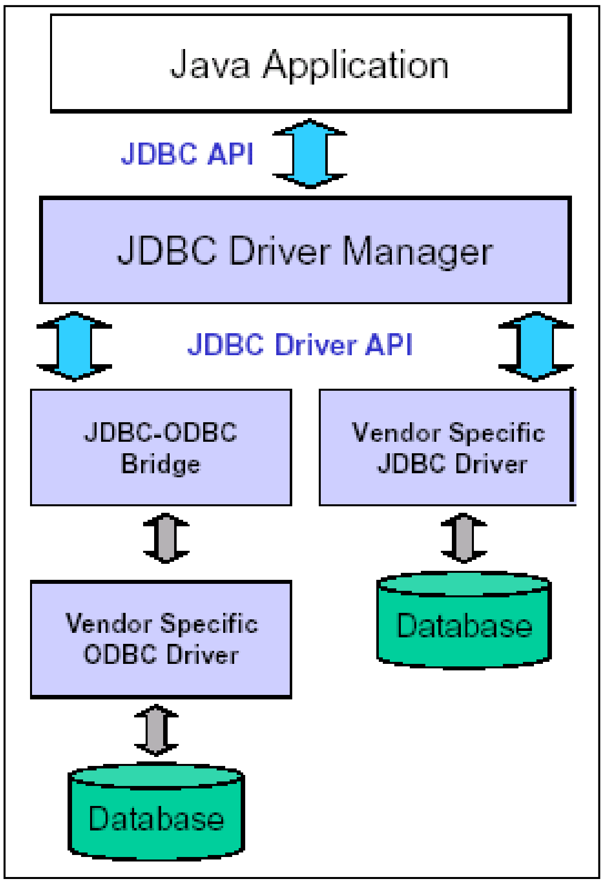
- After executing a query, you can retrieve the results using ResultSet object. ResultSet provides methods to iterate over the results and extract data row by row.
- JDBC methods can throw SQLExceptions, so it's essential to handle exceptions properly in your code to deal with errors gracefully.

Closing Resources

It's crucial to close the database resources like Connection, Statement, and ResultSet after you're done using them to release database and JDBC resources properly.

JDBC Architecture

JDBC (Java Database Connectivity) architecture provides a standard interface for Java applications to interact with relational databases.



JDBC Consists of two parts:

- JDBC API, a purely Java-based API
- JDBC Driver Manager, which communicates with vendor-specific drivers that perform the real communication with the database

Translation to the vendor format occurs on the client

- No changes needed to the server
- Driver (translator) needed on client

JDBC architecture Components

JDBC API:

- The JDBC API provides a set of classes and interfaces for Java applications to interact with databases.
- These classes and interfaces are part of the java.sql and javax.sql packages.

Driver Manager:

- The Driver Manager is responsible for managing the JDBC drivers.
- It maintains a list of available JDBC drivers and selects an appropriate driver based on the connection URL provided by the application.

JDBC Driver:

- JDBC drivers are software components that implement the JDBC API to provide connectivity between Java applications and databases. Connection

Pooling:

- In many enterprise applications, managing connections to the database efficiently is crucial for performance.

Connection pooling libraries or frameworks are often used to create and manage a pool of reusable database connections, reducing the overhead of establishing and tearing down connections for each database operation

Data Source:

- A data source is an object that provides a connection to a database.
- It is typically configured in a Java EE environment using a naming and directory service, such as JNDI (Java Naming and Directory Interface).

Connection:

- A Connection object represents a session with a specific database.
- It is used to create Statement, PreparedStatement, and CallableStatement objects for executing SQL queries and commands.

Statement:

- A Statement object is used to execute SQL queries and commands against the database.
- There are three types of statements Used for executing simple SQL queries without parameters.

PreparedStatement:

- Used for executing parameterized SQL queries, which helps prevent SQL injection attacks and improves performance by precompiling the SQL statement

CallableStatement:

- Used for executing stored procedures or functions.

ResultSet:

- A ResultSet object represents the result of a database query.
- It provides methods for traversing the result set and retrieving data from the query result.

Transaction Management:

- JDBC supports transaction management through the Connection object.
- Applications can start, commit, or rollback transactions to ensure data consistency and integrity.

JDBC classes

DriverManager

Load JDBC Driver:

- The first step is to load the JDBC driver class using **Class.forName("com.mysql.jdbc.Driver").**
- This dynamically loads the driver class into memory, allowing JDBC to recognize and use it.

Establish Connection:

- After loading the driver, the following method is called to establish a connection to the database specified by the URL (url), using the provided username and password.

DriverManager.getConnection(url, username, password)

Connection

Create Statement:

Once a connection (conn) is established, a statement object (stmt) is created using conn.createStatement().

This statement is used to execute SQL queries against the database.

Execute Query:

The stmt.executeQuery(sql) method executes the SQL query specified by sql and returns a ResultSet object containing the results.

Process ResultSet:

- The ResultSet object (rs) allows iterating over the rows returned by the query.
- The **while (rs.next())** loop iterates over each row in the result set.

Retrieve Data:

- Within the loop, data from each row can be retrieved using methods like **getString() or getInt()** by specifying the column name or index.

PreparedStatement

Prepare Statement:

- Instead of directly executing a SQL query, a prepared statement is used in scenarios where the same SQL statement will be executed multiple times with different parameter values.
- The SQL statement is provided with placeholders (?) for parameters.

Set Parameters:

- Parameters for the prepared statement are set using methods like setString() or setInt() to replace the placeholders with actual values.

Execute Update:

- The executeUpdate() method is used to execute SQL statements that modify the database, such as INSERT, UPDATE, or DELETE operations.

ResultSet

Retrieve Data:

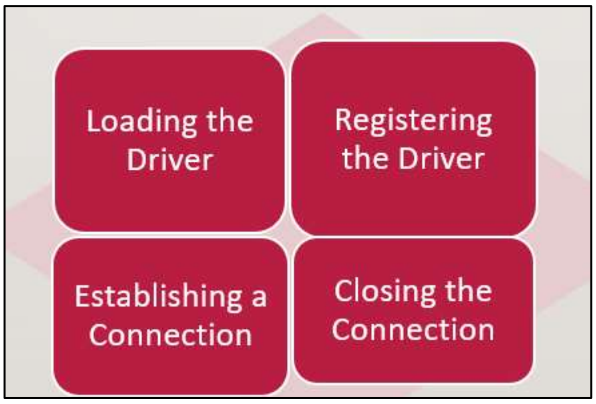
- Similar to the previous example, a ResultSet object is obtained by executing a query.
- The while (rs.next()) loop iterates over each row in the result set.

Retrieve Column Values:

- Inside the loop, column values can be retrieved using methods like getString() or getInt() by specifying the column name or index.

JDBC driver management

JDBC (Java Database Connectivity) driver management involves the process of handling JDBC drivers, which are software components that enable Java applications to interact with databases.



Loading the Driver

- The Class.forName() method is used to dynamically load the JDBC driver class into memory.

- This is necessary because JDBC drivers are typically packaged as JAR (Java ARchive) files containing the compiled Java classes.
- Loading the driver class allows the JVM to instantiate the driver and register it with the DriverManager.
- For example, to load the MySQL JDBC driver, you would use:

  Class.forName("com.mysql.jdbc.Driver");

- This line tells the JVM to load the class named com.mysql.jdbc.Driver into memory.

Registering the Driver

In earlier versions of JDBC, you had to explicitly register the JDBC driver with the DriverManager using the registerDriver() method.

- For example, to register the MySQL JDBC driver, you would use:

  DriverManager.registerDriver(new com.mysql.jdbc.Driver());

- This line registers an instance of the MySQL JDBC driver with the DriverManager.
- Since JDBC 4.0, drivers can be automatically loaded and registered when they are found in the classpath. However, explicit registration may still be necessary for certain drivers or compatibility reasons.

Establishing a Connection

- Once the driver is loaded and registered (if needed), you can establish a connection to the database using the DriverManager.getConnection() method.
- The connection URL is a string that specifies the location of the database, along with any additional connection parameters.
- For example, to connect to a MySQL database named mydatabase running on localhost with the default port (3306), you would use:

String url = "jdbc:mysql://localhost:3306/mydatabase";

String username = "username";

String password = "password";

Connection connection = DriverManager.getConnection(url, username, password);

- This line establishes a connection to the MySQL database using the provided URL, username, and password.

Closing the Connection

After you've finished using the database connection, it's important to close it to release any resources held by the connection.

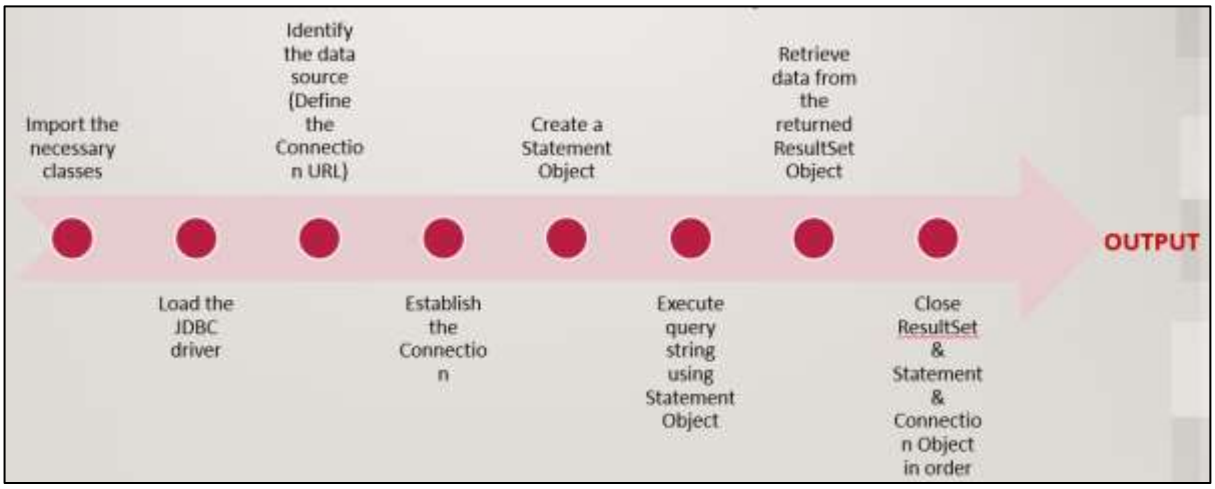
- Closing the connection is typically done using the close() method of the Connection interface.
- For example:

  connection.close();

- Closing the connection releases database resources, such as network connections and database cursors, and ensures proper cleanup.
- Failure to close connections can lead to resource leaks and may cause performance issues, especially in applications with heavy database usage.

JDBC Connection

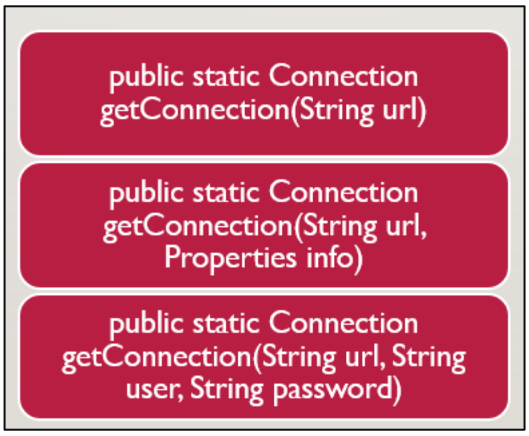
JDBC CONNECTIONS- Basic Steps


Import the necessary classes
Load the JDBC driver
Identify the data source (Define the Connection URL)
Establish the Connection
Create a Statement Object
Execute query string using Statement Object
Retrieve data from the returned ResultSet Object
Close ResultSet & Statement & Connection Object in order
OUTPUT

Connection-Creation

Required to communicate with a database via JDBC

Three separate methods:



public static Connection getConnection(String url)

public static Connection getConnection(String url, Properties info)

public static Connection getConnection(String url, String user, String password)

Code Example (Access)

```
try {// Load the driver class

    System.out.println("Loading Class driver");

    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

    // Define the data source for the driver

    String sourceURL = "jdbc:odbc:music";

    // Create a connection through the DriverManager class

    System.out.println("Getting Connection");

    Connection   databaseConnection   =   DriverManager.getConnection(sourceURL);


    }

catch (ClassNotFoundException cnfe) {

          System.err.println(cnfe); }

       catch (SQLException sqle) {
            System.err.println(sqle);}
```

Code Example (Oracle)

```
    try {

            Class.forName("oracle.jdbc.driver.OracleDriver");

            String sourceURL =
"jdbc:oracle:thin:@delilah.bus.albany.edu:1521:databasename";

            String user = "goel";

            String password = "password";

            Connection
databaseConnection=DriverManager.getConnection(sourceURL,user, password );

            System.out.println("Connected Connection"); }

            catch (ClassNotFoundException cnfe) {

            System.err.println(cnfe); }

            catch (SQLException sqle) {

            System.err.println(sqle);}
```

Connection-CLOSING

Each machine has a limited number of connections (separate thread)

If connections are not closed the system will run out of resources and freeze

Syntax: public void close() throws SQLException

Naïve Way:

```
         try {

         Connection conn

         = DriverManager.getConnection(url);

          // Jdbc Code
          ...
} catch (SQLException sqle) {
            sqle.printStackTrace();
          }
          conn.close();
```

SQL exception in the Jdbc code will prevent execution to reach conn.close()

Correct way (Use the finally clause)

```
try{
Connection conn = Driver.Manager.getConnection(url);
          // JDBC Code
          } catch (SQLException sqle) {
                    sqle.printStackTrace();
          } finally {
                    try {
                        conn.close();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
          }
```

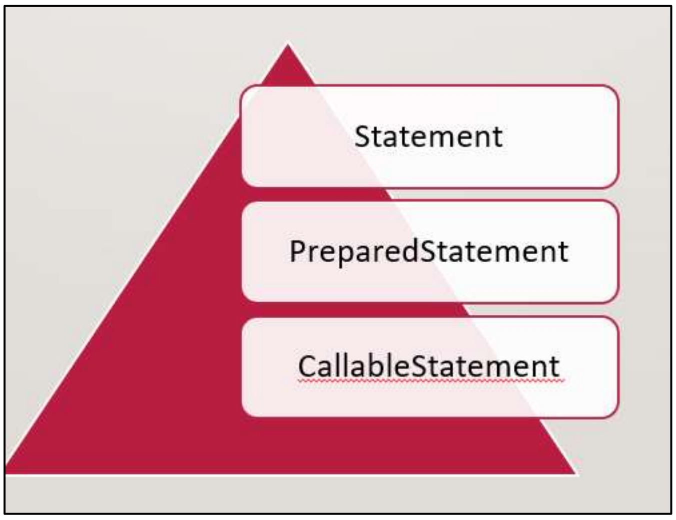
JDBC statements

Statement Types

- Statements in JDBC abstract the SQL statements
- Primary interface to the tables in the database
- Used to create, retrieve, update & delete data (CRUD) from a table

Syntax: Statement statement = connection.createStatement();

Three types of statements each reflecting a specific SQL statements



Statement Syntax

Statement used to send SQL commands to the database

Case 1: ResultSet is non-scrollable and non-updateable

public Statement createStatement() throws SQLException

Statement statement = connection.createStatement();

Case 2: ResultSet is non-scrollable and/or non-updateable

public Statement createStatement(int, int) throws SQLException

Statement statement = connection.createStatement();

Case 3: ResultSet is non-scrollable and/or non-updateable and/or holdable

public Statement createStatement(int, int, int) throws SQLException

Statement statement = connection.createStatement();

PreparedStatement

public PreparedStatement prepareStatement(String sql) throws SQLException

PreparedStatement pstatement = prepareStatement(sqlString);

CallableStatement used to call stored procedures public CallableStatement prepareCall(String sql) throws SQLException

Statement Release

- Statement can be used multiple times for sending a query
- It should be released when it is no longer required
    - Statement.close():
    - It releases the JDBC resources immediately instead of waiting for the statement to close automatically via garbage collection
- Garbage collection is done when an object is unreachable
    - An object is reachable if there is a chain of reference that reaches the object from some root reference
- Closing of the statement should be in the finally clause

```
try{
    Connection conn = Driver.Manager.getConnection(url);
    Statement stmt = conn.getStatement();
    // JDBC Code
    } catch (SQLException sqle) {
            sqle.printStackTrace();
    } finally {
             try {stmt.close();
                   conn.close();
      } catch (Exception e) {
                    e.printStackTrace();
              }
    }
```