UNIT 3 JDBC PART 1

Structure

3.0 INTRODUCTION

In this unit, we will discuss the Database Connectivity concept for Java applications. We know, that we can save our data in the file system, but it doesn't offer any capability to querying on data efficiently. At the same time, we know that the various databases like Oracle, MySQL, Sybase etc., provide file-processing capabilities and facilitates efficient query processing. We will explore the Java Database Connectivity (JDBC) Application Program Interface (API) and the interaction with a database using the JDBC feature of Java.

We can write Java applications using JDBC to manage different programming activities, starting from establishing a database connection and then sending a query and retrieving result updates from the database. In general, using JDBC API, we can create the table, update and insert values, fetch records, create prepared statements, perform transactions and catch exceptions.

We can visualize the working of the JDBC API where we are writing a typical Java program and using standard library routines. First, a database connection has been established; subsequently, we use JDBC to query the database. After processing, the result is returned to the application, and finally, we close the connection.

In this unit, we will also explore various JDBC components like Driver, Driver Manager, Connection, Statement etc., which are used for database interaction. A JDBC Driver is used to open the database connection for query execution and receiving results with the Java application. The Driver Manager ensures the transparent access of each database using the correct Driver, which is specific to that database. The Connection interface creates the session between the application and the database and, finally, the SQL query execution and subsequent fetching of results through some pre-defined libraries stored as Statement object, PreparedStatement object and CallableStatement object.

3.1 OBJECTIVES

After going through this unit, you will be able to:

- write the steps for database connectivity within the Java application using JDBC,
- write Java application to execute SQL query and fetch the results using JDBC API,
- explain various JDBC components required for database interactions,
- use drivers to open the connection for the execution of database commands ,
- use Driver manager, which loads all the system drivers and select the most appropriate Driver, and
- use Connection interface, which offers different methods for database communication

3.2 JDBC INTRODUCTION

JDBC API used provide database connectivity between Java applications and various databases. It defines how to access Relational Database (RDBMS) through standard interfaces and classes. Precisely JDBC API helps us to write Java applications that can access tabular data.

JDBC was created by Sun Microsystems and is SQL-based API to access databases, but now it is own by Oracle. JDBC has been developed as an alternative to the Open Database Connectivity (ODBC), a C-based API created by Microsoft and is used to access databases via SQL.

JDBC API allows the Java application to access various types of databases like Oracle, MS Access, Sybase, MySQL and SQL Server etc. It offers the platform-independent interface between the Java application and the RDBMS and allows the application to execute SQL statements. Various tasks are associated with database usage, such as establishing a connection, query creations, query executions, and viewing and modifying the resulting records. Primarily, JDBC allows mobile access to an underlying database, and for seamless performance, it provides a complete set of interfaces.

To better understand the concept, let us first discuss the Java database connectivity architecture. The JDBC Architecture provisions both 2-tier and 3-tier models but in general, it consists of two layers where through JDBC API, the first layer manages the application-to-JDBC Manager connection and the other layer JDBC Manager-to-Driver Connection.

The two-tier architecture provides direct communication of Java applications with the database where the JDBC driver facilitates communication with a specific database. The Driver receives a request from the application and transforms it into a vendor-specific database call, which is passed to the database. This configuration is termed the client-server architecture, where the client is the user machine, and the database acts as a server. This architecture facilitates the user to send the query to the database, and results are retrieved back by the user.

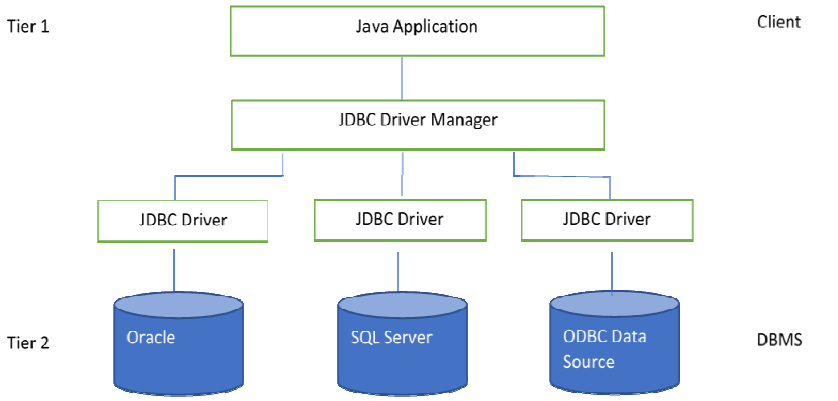

Figure 3.1 Two-tier architecture

In the three-tier architecture, the middle tier is used to take care of the business logic. In this architecture, the user machine sends a command to the middle-tier, which directs it for processing to the database. After processing the commands, database sends back the results to the middle tier, and subsequently, middle-tier sends it to the user machine. The JDBC API is used in the Application business layer of a three-tier architecture.

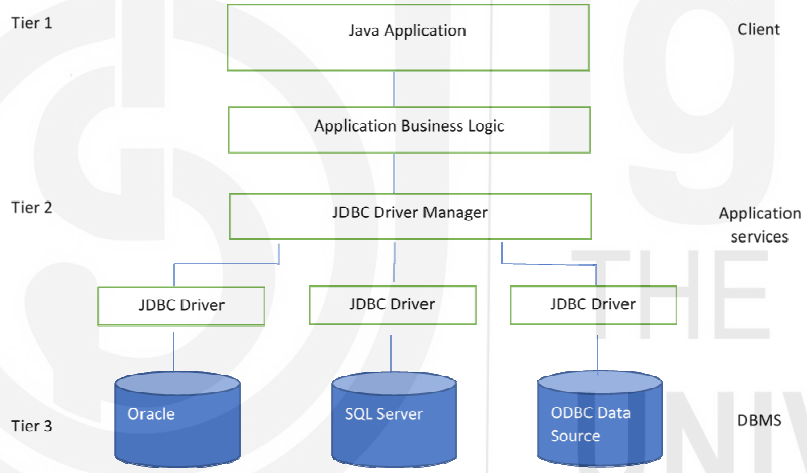

Figure 3.2 Three-tier architecture

The Java application communicates with a database, and the JDBC API facilitates the Java application to execute the SQL queries. The application then retrieves the processed results in a consistent and database independent manner. It uses a driver manager and database-specific Driver and ensures that the driver manager uses the correct Driver to access each database. The JDBC API provides various classes and interfaces that represent objects for this purpose. It includes Connection interface, Statements, PreparedStatement, CallableStatements Interface, ResultSet, Metadata Interface, BLOBs, CLOBs, Database drivers, DriverManager etc.

The JDBC API uses a Driver Manager class to ensure the transparent access of each database. The transparent access to a database by facilitating the correct Driver specific to that database. The JDBC drivers enable the Java applications to make use of different operating systems and hardware platforms.

The API technology provides the standards for Java applications to use the database independently due to its WORA capabilities (“Write Once, Run Anywhere”). Database independence is achieved by using various methods and interfaces that set

up an accessible communication with the database. The application can thus execute queries, retrieve results and update data. JDBC API is part of the Java Standard Edition (SE) and Java Enterprise Edition (EE) platform and provides Java.sql.* and Javax.sql.* packages.

3.2.1 Core JDBC Components

JDBC consists of the following core components through which it can interact with a database:

1. JDBC Driver Manager
2. JDBC Driver
3. Connection Interface
4. Statement Interface
5. ResultSet Interface

JDBC Driver manager: It is the main class that defines an object which is used to connect the application to a JDBC driver. The Driver Manager manages various types of drivers running on an application. The primary responsibility is to correctly loads all the system drivers and select the suitable Driver for opening a database connection. There are Standard Extension packages Javax.naming and Javax.sql which are used to create the database connection. We can make use of any package for the connection, but it is recommended to use a Data Source object registered with a Java Naming and Directory Interface™ (JNDI) naming service.

Driver: A JDBC driver opens the database connection for the execution of database commands and receiving results with the Java application. A JDBC driver establishes a connection with a database and subsequently send an update and retrieve queries. The JDBC API requires drivers for each database and ensures database independence.

Connection Interface: The connection interface offers different methods for database communication, and all the database communication is possible only through the connection object.

Statement Interface: The statement interface object submits a SQL query to the database. It provides two sub-interfaces PreparedStatement and CallableStatement. These interfaces execute queries and fetch the processed result of the executed SQL queries.

ResultSet Interface: The ResultSet interface provides the outcome of a query. The ResultSet object maintains a cursor that points at the current row in the result set data. The result set refers to the row and column data possessed by a ResultSet object of the underlying database.

3.3 JDBC Driver

A JDBC driver is a software component that translates standard JDBC calls into a database library API call and enables interaction of the Java application with the database. The JDBC drivers open the database connection to execute database commands and receive results with the Java application. The advantage of using these drivers is to ensure database independence, which implies that we can easily change the backend database by only changing the database driver and a few lines of code.

There are four different types of JDBC drivers having diverse implementations. Let us discuss these drivers:

3.3.1 Type 1/JDBC-ODBC Bridge

The JDBC-ODBC bridge driver acts as a bridge where ODBC drivers are used for accessing JDBC. This bridge driver connects to the database by converting JDBC method calls into ODBC function calls. First, the Java statement transformed into a JDBC statement which subsequently calls the ODBC. This transformation is done by using the Type-I Driver and, finally, the query execution. Oracle does not support this Driver, and this JDBC-ODBC bridge driver is removed from Java 8 onwards.

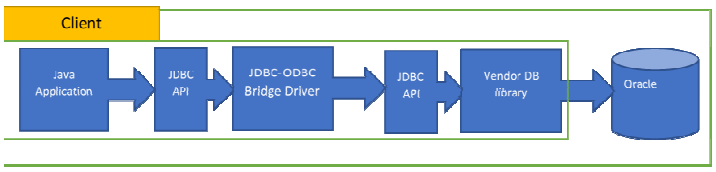

Figure 3.3 JDBC-ODBC bridge driver

Advantage: The JDBC-ODBC Bridge driver was easy to use and offered easy connectivity with any database.

Disadvantage: The Driver was required to install on the client machine.

3.3.2 Type 2/Java to Native API

The Java to Native API driver uses the Java Native Interface (JNI) to make calls to a local database library API. The Type 2 driver connects with the database by converting the JDBC method calls into a native call of the database API. It requires some native code to communicate directly with the database server, but not entirely written in Java.

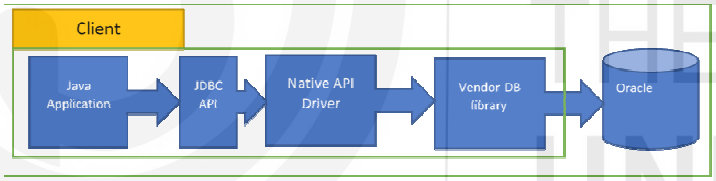

Figure 3.4 Java to Native API driver

Advantage: The Type 2 driver offers better performance than the JDBC-ODBC Driver.

Disadvantage: The native Driver and the vendor library need to install on every client machine. Also, we need to change the native API if we change the database.

3.3.3 Type 3/ Java to Network Protocol

The Network Protocol driver connect with the middleware using a three-tier approach. The Driver connects with the database by converting the JDBC method calls into the vendor-specific database protocol. It is also called an All-Java Driver as entirely written in Java (pure Java drivers).

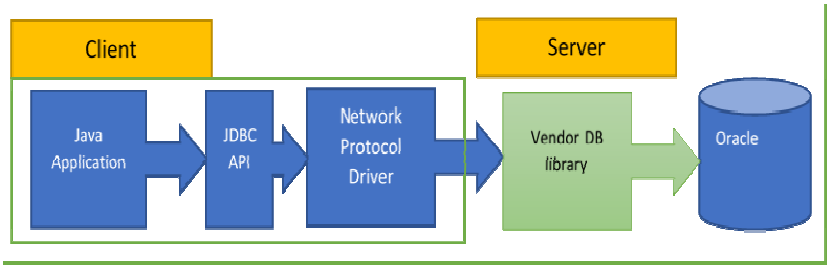

Figure 3.5 Java to Network Protocol driver

Advantage: The Type 3 driver offers a flexible JDBC solution as the native database libraries are not required to install on the client-side. Another advantage of this Driver is that a single driver can be used to access multiple databases.

Disadvantage: The maintenance of the Type 3 driver is costly as the coding at the middleware is database-specific. The server performs various tasks like load balancing, logging etc., but it will require to have network support on the client machine.

3.3.4 Type 4/ Java to Database Protocol

The Thin Driver is the fastest pure Java driver and communicates directly through socket connections without requiring any middleware or native library. The Driver connects with the database by converting the JDBC method calls into the vendor-specific database-protocol directly. This Driver converts the Java statements to SQL statements directly without the need of any intermediate library, and so it is also called a thin driver.

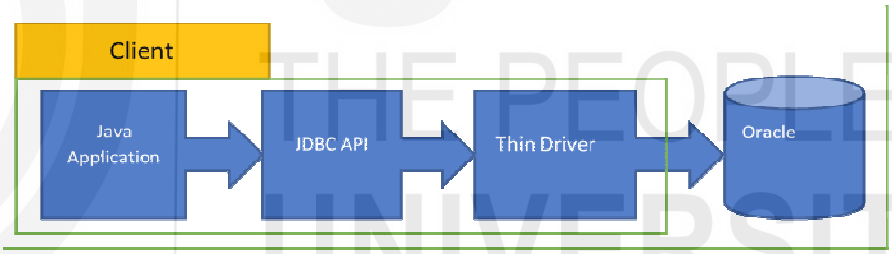

Figure 3.6 Thin deriver

Advantage: These drivers are deployed online without requiring any software installation on the client or server.

Disadvantage: The Type 4 driver is database dependent which means if our backend database changes, we need to deploy a new driver compatible with the database.

Generally, the choice of the Driver depends on many factors like the requirements of the project, cost, flexibility, interoperability, performance etc. Type 1 and Type 2 driver are available mostly free of charge, but they require software installation and software configuration on each client. Type 3 and Type 4 are not open-source. Still, the Type 3 driver offers access to different types of databases, i.e. it is suitable for the application which requires access to multiple databases at the same time. In contrast, Type 4 drivers usually provide better performance.

☞ Check Your Progress 1

1. Briefly explain the necessary steps to create a JDBC application?

2. Which JDBC driver is the fastest Driver, and why?

3. How many packages are available in JDBC API?

3.4 JDBC DATABASE CONNECTION STEPS

Let us first discuss and explore the various components used in the JDBC Database Connection steps. The JDBC API provides a set of interfaces and classes to establish a connection with the database, create and execute SQL queries, to fetch the results and processing that ResultSet.

JDBC Steps for connecting a Java application with the database can be summarized as follows:

- Load the Driver
- Create a connection
- Create Statement
- Query execution
- ResultSet processing
- Close connection

The JDBC provides driver interface, a Connection interface, Statement interface, and ResultSet interface for the execution.

Step 1: Loading the Driver: The JDBC drivers are acting as a gateway to a database. The first thing we need to do is to initialize a driver before its usage in the program. We need to load the Driver or register the Driver once to open a communication channel with the database. For MYSQL JDBC Driver, we can download MySQL-connector-Java-8.0.21-bin.jar (current), extract the .jar file and add its path into $CLASSPATH.

There are two ways through which we can register the Driver.

1. **DriverManager.registerDriver():** This is an inbuild Java class with register as its static member and will call the driver class constructor at the compile time.

To register the MYSQL Driver, we can use the following syntax

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

2. **Class.forName():** This method will dynamically load the Driver's class into memory at the runtime.

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

Generally, the JVM automatically loads the classes used in the program But, when the driver class is not used explicitly, we need to explicitly inform the JVM to load the driver class to be used in the program. Since Java version 6 onwards, the JDBC driver is not required to explicitly load; instead, we just need to have the appropriate jar in the classpath. For this reason, that driver loading step is very much non-compulsory from Java 6 onwards. However, it is highly recommended that you check whether automatic loading is available with the Driver used in the application.

Step 2: Create the connections: After we load the Driver, it is required to open a connection which is done by creating a connection object through the static getConnection() method. The connection object establishes a physical connection with the database of the DriverManager class.

Connection jdbcconn = DriverManager.getConnection(URL, user, password)

jdbcconn is a reference to Connection interface, "user" is the username used to access the SQL command prompt, "password" is the password used to access the SQL command prompt, and URL is Uniform Resource Locator. We can define it as follows:

URL = "jdbc:mysql://localhost:3306/emp", user = "root" and password = "admin"

We can connect with the database using the credentials as user "root" with password "admin", and the schema is emp through the port number 3306 of host localhost (or we may use the address of database server).

Step 3: Create a statement: After establishing a connection to interact with the database, the createStatement() method is used to send commands to the database. The Statement object subsequently used to execute the query.

The JDBC Statement used is as follows:

Statement jdbcstmt = jdbcconn.createStatement();

jdbcconn is a reference to the Connection interface

Step 4: Query Execution: The query execution to retrieve the result from the database, we can use the executeQuery() method of the Statement class. There are multiple types of queries in the database like an update, insert or retrieval of data. This executeQuery() method returns the ResultSet object, which is used to fetch the records of a table.

The executeUpdate(SQL query) method of Statement class is used to execute update or insert query. It returns an integer that represents the number of affected rows by the SQL statement.

The JDBC Statement to create a table is as follows:

jdbcstmt.executeUpdate("CREATE TABLE EMP (id INTEGER not NULL, first VARCHAR(30)," last VARCHAR(30), age INTEGER, city VARCHAR(30), salary INTEGER, PRIMARY KEY (id))");

The JDBC Statement to get the data is as follows:

ResultSet jdbcresultset = jdbcstmt.executeQuery("Select * from Emp");

Step 5: ResultSet Processing: After query execution, we have the ResultSet, and the data access through the ResultSet object over a cursor. The cursor is a pointer and initially placed before the first row. Now, using the next() method, it can advance to the subsequent rows of the ResultSet. There are different getter methods of the ResultSet are available to access the values of the current row, depending on datatypes.

For example, to access the records of all the Employees the ResultSet can be iterated like:

while(jdbcresultset.next())

{

System.out.println("id : " + jdbcresultset.getInt("id") + " First : " + jdbcresultset.getString("first") + " Last : " + jdbcresultset.getString("last") + " Age : " + jdbcresultset.getInt("age") + " City : " + jdbcresultset.getString("city") + " Salary : " + jdbcresultset.getInt("salary"));

}

Step 6: Close the connections: After fetching the records, we will close the connection. The Connection interface close() method used to close the connection and the Statement and ResultSet objects will be closed automatically.

Syntax :

jdbcconn.close();

Let us now take an example, in which we create a JDBC application which establishes a database connection to consolidate our understanding of the concept.

Example: 3.5.1:

```
import Java.sql.*;
import Java.sql.DriverManager;
public class connection
{
 public static void main(String[] args)
 {
  Connection jdbcconn = null;
  try
  {
   //STEP 1: The JDBC driver is registered
   Class.forName("com.mysql.cj.jdbc.Driver");
```

```
// Alternatively, DriverManager.registerDriver(new
com.mysql.cj.jdbc.Driver());

//STEP 2: Open connection
System.out.println("Connecting to database...");
jdbcconn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/emp",
"root", "admin");

//STEP 3: Create the Statement object
Statement jdbcstmt = jdbcconn.createStatement();

//STEP 4: Execute query to show table
System.out.println("Executing statement...");
ResultSet jdbcresultset = jdbcstmt.executeQuery("Select * from
Emp");

//STEP 5: The ResultSet is processed
while(jdbcresultset.next())
{
 //Retrieve data
 System.out.println("id : " + jdbcresultset.getInt("id") + " First
 : " + jdbcresultset.getString("first")  + " Last : " +
 jdbcresultset.getString("last") +  " Age : " +
 jdbcresultset.getInt("age") + " City : " +
 jdbcresultset.getString("city") + " Salary : " +
 jdbcresultset.getInt("salary"));
}
//STEP 6: Clean-up environment
jdbcconn.close();
}
catch(SQLException sqlex)
{
 //This will take care of JDBC errors
 sqlex.printStackTrace();
}
catch (Exception ex)
{
 // This will take care of Class.forName errors
 ex.printStackTrace();
}
finally
{
 // This will take care of closing all the resources
 Try
 {
  if (jdbcconn !=null)
  jdbcconn.close();
 }
 catch (SQLException sqlex)
 {
  sqlex.printStackTrace();
 }
 //  end finally
}
// end try
System.out.println("Executed!");
}
//end main
}
// end Example
```

Output:

```
Connecting to database...
Executing statement...
id : 501 First : Aman Last : Sharma Age : 40 City : Delhi Salary : 8000
id : 502 First : Ajay Last : Tandon Age : 50 City : Mumbai Salary : 9000
id : 505 First : Suraj Last : Gupta Age : 45 City : Pune Salary : 8000
id : 521 First : Aman Last : Sharma Age : 40 City : Delhi Salary : 8000
id : 522 First : Ajay Last : Tandon Age : 50 City : Mumbai Salary : 9000
id : 525 First : Suraj Last : Gupta Age : 45 City : Pune Salary : 8000

Executed!
```

In this example: first, the import statements are used to import required classes in the code.

In step 1, We registered the JDBC driver (MySQL's Connector/J driver /Type 4 driver), and the Java virtual machine has loaded the desired driver implementation.

In step 2, we have used the getconnection() method. It opens a connection using database URL, username and password.

In step 3, we have created the statement object.

In step 4, We executed the query, where we have accessed an existing database through the connection object, and then data is inserted and accessed through the JDBC driver.

In step 5, we have extracted the data through the result set object and

In step 6, we have closed all the database connections to end the sessions.

3.5 JDBC SQLEXCEPTION

Exception handling is advantageous as it separates the Error-Handling Code from the main logic of the program. In the programming constructs, an exception occurs whenever we come across an abnormal condition, which interrupts the normal flow of the main logic code. The exceptions are handled in a controlled manner, and this stops the normal flow of execution and the control redirects to the possible catch clause. In JDBC, the exception instance generally thrown is SQLException. An SQLException means when a connection object is unable to find an appropriate driver to connect with the database, an exception is thrown. An SQLException thrown when we encounter an error during database interaction may be an inappropriate Driver or URL.
Following are the commonly used methods for JDBC SQLException:

getMessage( ): This method used to get the error message. For database error, it will provide Oracle error number and message, and for the driver error, it will provide JDBC driver error message.

getErrorCode( ): This method is used to get the exception error code, and generally, it is a vendor-specific exception.

setNextException(SQLException sqlex): This method is used to add another SQL exception in the chain.

getNextException( ): This method gets the next exception object from the exception chain.

printStackTrace( ): This method is used to print the current exception, throwable, and backtrace to a standard error stream.

printStackTrace(PrintStream s): This method is used to print the throwable and its backtrace to the specified print stream.

printStackTrace(PrintWriter w): This method is used to print the throwable and backtrace to the specified print writer.

The try-catch block utilized the information from the exception object, and it will catch the exception to continue the program execution.

```
try
{
  // Error prone code
}
catch(Exception ex)
{
  // exception handling code
}
finally
{
  // must be executed code for e.g. jdbcconn.close();
}
```

In the try-catch block, we write the code which has potential chances of failure. If the code fails, the try block should throw an exception object, and the catch block will take care of that exceptions. The exception in the catch block handles the exception object and may print a statement or perform a pre-defined action to handles the issue. We may use multiple catch blocks, which allows taking more than one exception. The finally-block execute all other necessary components of the program despite the exception. JDBC Exceptions are an efficient way of handling exceptions and should always be used.
Now to consolidate our understanding of Exception handling, let us take the previous example 3.5.1 and pass the incorrect credentials.

In the first case, we pass the wrong driver, and the result will be class not found exception at exceptionhandling.main(exceptionhandling.Java:8)

Output:

```
Java.lang.ClassNotFoundException: com1.mysql.cj.jdbc.Driver
        at
Java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader
.Java:602)
        at
Java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoad
ers.Java:178)
        at Java.base/Java.lang.ClassLoader.loadClass(ClassLoader.Java:522)
        at Java.base/Java.lang.Class.forName0(Native Method)
        at Java.base/Java.lang.Class.forName(Class.Java:340)
        at exceptionhandling.main(exceptionhandling.Java:8)
Executed!
```

3.6 JDBC DRIVERMANAGER CLASS

In the second case, let we pass the incorrect database name, and the program will show the unknown database error at exceptionhandling.main(exceptionhandling.Java:13). The outcome of the program as follows:

Output:

```
Connecting to database...
Java.sql.SQLSyntaxErrorException: Unknown database 'emp1'
        at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.Java:120)
        at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.Java:97)
        at
com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLExceptionsMappin
g.Java:122)
        at com.mysql.cj.jdbc.ConnectionImpl.createNewIO(ConnectionImpl.Java:836)
        at com.mysql.cj.jdbc.ConnectionImpl.<init>(ConnectionImpl.Java:456)
        at com.mysql.cj.jdbc.ConnectionImpl.getInstance(ConnectionImpl.Java:246)
        at com.mysql.cj.jdbc.NonRegisteringDriver.connect(NonRegisteringDriver.Java:197)
        at Java.sql/Java.sql.DriverManager.getConnection(DriverManager.Java:677)
        at Java.sql/Java.sql.DriverManager.getConnection(DriverManager.Java:228)
        at exceptionhandling.main(exceptionhandling.Java:13)
Executed!
```

The JDBC DriverManager is a concrete Java class that provides an interface between the user and drivers. The other JDBC components are Java interfaces implemented by the various driver packages. The JDBC driver can create the database connections, but generally, the DriverManager is used to get the connection. The JDBC DriverManager provides elementary services to maintain JDBC drivers and defines an object that connects the application to the Driver.

The DriverManager manages the various drivers running on an application. It tracks the available drivers and appropriately chooses the one which connects the application with the database.

It also tracks the login time limits of the driver and recording logs and tracing messages. The DriverManager.getConnection() method is the most commonly used method of this class to establish the database connection.

It maintains driver classes registered through the DriverManager.registerDriver() method and a list of all the available drivers preserved with the DriverManager class. The Class.forName() method is used to load the Driver. The DriverManager is being informed by the JDBC drivers whenever their implementation class is loaded.

Syntax:

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

There are many associated methods with this class. Let us summarize a few of the important methods:

registerDriver(Driver driver): This method is used to register the Driver with the DriverManager class.

deregisterDriver(Driver driver): This method is used to deregister or drop from the list of registered drivers in the DriverManager class.

getConnection(String URL): This method is used to establish a connection with the given database URL.

getConnection(String URL, String username, String password): This method is used to establish the connection with the given URL, username and password.

getConnection(String URL, Properties info): This method is used to establish a connection with the given URL.

getDrivers(String URL): This method attempts to locate the Driver by the given string.

getDrivers(): This method retrieves the information of the registered drivers of the DriverManager class.

☞ Check Your Progress 2

1. What are two approaches for registering the Driver?

2. Write down the importance of the JDBC DriverManager class?

3. getConnection()method belongs to which class?

4. Why do we get the following exception Java.sql.SQLException: No suitable driver found?

5. Can we use multiple catch and finally blocks with a single try statement?

.....

.....

.....

.....

3.7 JDBC CONNECTION INTERFACE

A Connection is an interface used for creating the session between the application and the database. A Java application may require to connect with a single database or sometimes, depending on the application, may require many connections with the different databases. The Connection interface of the Java.sql package characterizes a session with the connected database.

After a connection establishment, the execution of the SQL statements takes place. This process of query execution and result fetching is within the context of the connection. The implementation of these SQL queries requires the usage of some pre-defined libraries stored as Statement object, PreparedStatement object and CallableStatement object. Based on the application requirements, we can use any one of them.

The Connection interface contains Statement interface, PreparedStatement interface and CallableStatement interface. The getConnection() method of the class DriverManager is used to get a Connection object which is subsequently used to get the Statement object of Statement, PreparedStatement and CallableStatement. The connection interface also supports various methods for transaction management, another critical concept while using databases.

The DriverManager.getConnection() method is used to establish a connection and the DriverManager class searches and locate the most appropriate driver from the list of all registered Driver classes that can connect with the database specified in the URL.

syntax:

```
Connection jdbcconn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/emp", "root", "admin");
```

The commonly used methods for the Connection Interface are as follows:

createStatement(): This method creates a statement object for executing the SQL queries.

createStatement(int resultSetType, int resultSetConcurrency): This method creates a statement object for the ResultSet

PreparedStatement(String SQL): This method creates a PreparedStatement object for executing parameterized queries.

getMetaData(): This method returns a Database MetaData object containing the connected database information.

setAutoCommit(boolean status): This method is used to set the commit status, which is by default always true.

commit(): This method saves the changes which have been committed or are permanent rollback.

rollback(): This method drops all changes when commit or permanent rollback

close(): This method closes the connection and immediately releases the JDBC resource.

3.7.1 Statement Interface

The Statement interface is used for the general-purpose access of the database. It is generally used when working with static SQL queries at runtime. We connect the JDBC to interact with the database, the Java.sql package, the two interfaces Java.sql.Statement and Java.sql.PreparedStatement. are used for efficiently executing the SQL queries.

For query execution, we may use any interface, but they have a different implementation. The Statement interface is used to execute the static SQL statement, and the PreparedStatement interface is used to execute pre-compiled queries.

The Statement interface provides various methods for query execution. It executes SQL queries using the basic methods like Statement.executeQuery(String) or Statement.executeUpdate(String) for DDL operations. This interface provides the Object of ResultSet. The Connection.createStatement() method is used to get a Statement object.

syntax:

Statement jdbcstmt = jdbcconn.createStatement();

The PreparedStatement and CallableStatement are the two sub-interfaces offered by the Statement interface.

The PreparedStatement object stores the pre-compiled query, which allows multiple runs of the same SQL query with different parameters efficiently. On the other hand, the CallableStatements are used to execute the SQL stored procedure. We will discuss these sub-interfaces in the subsequent section.

Following are the commonly used methods for Statement interface.

ResultSet executeQuery(String SQL): The executeQuery method is used to execute the SELECT statement, and will return the ResultSet object.

int executeUpdate(String SQL): The executeUpdate method is used to execute a specific query like insert, update, delete etc. and will return the number of affected rows.

Boolean execute(String SQL): The execute method used to execute queries that returns a boolean value and will return true if a ResultSet object fetched the result else, it would return false.

executeBatch(): The executeBatch method executes batch statements.

Let us write a simple example of Statement interface to perform various basic operations like insert, update and delete.

Example 3.7.1:

```
import Java.sql.*;
import Java.sql.DriverManager;
class statementInterface
{
  public static void main(String args[])
  {
   Connection jdbcconn = null;
   try
   {
    Class.forName("com.mysql.cj.jdbc.Driver");
    jdbcconn =
    DriverManager.getConnection("jdbc:mysql://localhost:3306/emp",
    "root", "admin");
    Statement jdbcstmt = jdbcconn.createStatement();

    // Update the record using execute method
    boolean status = jdbcstmt.execute("Update Emp set age = 40 where
    id =502");
    if(status == false)
    {
     System.out.println("Number of Updated rows " +
     jdbcstmt.getUpdateCount() );
    }
    //  Insert the record using execute method
    int count1 = jdbcstmt.executeUpdate("Insert into Emp(id, First,
    Last, Age, City, Salary) values(512,'Gaurav', 'Mehta', 44,
    'Delhi', 5000)");
    System.out.println("Number of Rows Inserted " + count1);

    // Update the record using executeUpdate method
    int count2 = jdbcstmt.executeUpdate("Update Emp set age = 35 where
    id = 521");
    System.out.println("Number of  Updated rows" + count2);

    // Delete the record using executeUpdate method
    //count = jdbcstmt.executeUpdate("Delete from Emp where id = 41");
    //System.out.println("Number of Rows Deleted " + count);

    // Executing Query using executeQuery method
    ResultSet jdbcresultset = jdbcstmt.executeQuery("Select * from
    Emp");

    // Retrieve data using ResultSet
    while(jdbcresultset.next())
    {
     System.out.println("id : " + jdbcresultset.getInt("id") + " First
     : " + jdbcresultset.getString("first")  + " Last : " +
     jdbcresultset.getString("last") +  " Age : " +
     jdbcresultset.getInt("age") + " City : " +
     jdbcresultset.getString("city") + " Salary : " +
     jdbcresultset.getInt("salary"));
    }
    jdbcconn.close();
   }
   catch(Exception e)
   {
    //Handle errors for Class.forName
    System.out.println(e);
   }
   //end try
```

```
   System.out.println("Executed!");
  }
  //end main
}
// end Example
```

Output:

```
Number of Updated rows 1
Number of Rows Inserted 1
Number of  Updated rows1
id : 501 First : Aman Last : Sharma Age : 40 City : Delhi Salary : 8000
id : 502 First : Ajay Last : Tandon Age : 40 City : Mumbai Salary : 9000
id : 505 First : Suraj Last : Gupta Age : 45 City : Pune Salary : 8000
id : 512 First : Gaurav Last : Mehta Age : 44 City : Delhi Salary : 5000
id : 521 First : Aman Last : Sharma Age : 35 City : Delhi Salary : 8000
id : 522 First : Ajay Last : Tandon Age : 50 City : Mumbai Salary : 9000
id : 525 First : Suraj Last : Gupta Age : 45 City : Pune Salary : 8000

Executed!
```

3.7.2 PreparedStatement Interface

As we have discussed that for query execution, we can use any of the interfaces Java.sql.Statement and Java.sql.PreparedStatement. The Java.sql.Statement is generally used for simple DDL queries but for better efficiency where repeating SQL queries or parameterization is required, we use the Java.sql.PreparedStatement. PreparedStatement interface inherits the primary statement interface mentioned earlier and offers parameterization, and also it is much safer against SQL injection attacks.

PreparedStatement interface is generally used when we need to execute the SQL statements multiple times, and the interface accepts input parameters at the runtime. The interface corresponds to pre-compiled statements, offers the better performance of the application and will fetch the result faster as the query is compiled only once. These statements are first compiled into the database and subsequently stored as a PreparedStatement object. This object facilitates the SQL statement execution several times.

The PreparedStatement interface, when used for the parameterized queries a question mark (?) symbol as a placeholder known as parameter-marker, is passed for the values. The question mark values are set by the PreparedStatement and values supplied for every parameter before query execution. The advantage of using PreparedStatement is that we can use the same query multiple times with different parameter values.

For example, String SQL="insert into Emp values(?, ?, ?, ?, ?, ?)"

Here, we pass these (?) parameter used as a placeholder in the query and the values set at the runtime by calling the PreparedStatement setter methods. The various setter methods are used to provide values for these placeholders based on the data-types like setInt(), setFloat(), setString() etc. The setInt(int parameterIndex, value) is the standard form of the setter method where “Int” part in setInt is varying based on the datatype float, string etc. and the parameter index position is defined as parameterIndex in the statement. The parameterIndex starts from 1, unlike Java array indices, which starts at 0.

We can use the PreparedStatement method of the Connection Interface to get the PreparedStatement object.

Syntax

PreparedStatement jdbcprepstmt = connection.prepareStatement(SQL);

The PreparedStatement is more efficient as it directly sent the compiled SQL to the database, thus is more efficient for repeated executions.

Following are the commonly used methods for PreparedStatement interface:

setInt(int parameterIndex, int value): The setInt method used for setting Int value as per the parameter index.

setString(int parameterIndex, String value): The setString method is used to set the String value as per the parameter index.

setFloat(int parameterIndex, float value): The setFloat method is used to set the Float value as per the parameter index.

setDouble(int parameterIndex, double value): The setDouble method is used to set the Double value as per the parameter index.

executeUpdate(): The executeUpdate method used for executing a query like create, drop, insert, update, delete etc.

ResultSetexecuteQuery(): This method is used for executing the select query, and it will return an instance of ResultSet.

Let us write a simple example of the parameterized interface to perform the basic operations of insert, update and delete

Example 3.7.2:

```
import Java.sql.*;
import Java.sql.DriverManager;
public class preparedStmt
{
  public static void main(String[] args)
  {
    Connection jdbcconn = null;
    try
    {
      Class.forName("com.mysql.cj.jdbc.Driver");
      System.out.println("Connecting to database...");
      jdbcconn =
      DriverManager.getConnection("jdbc:mysql://localhost:3306/emp",
      "root", "admin");
      preparedStmt jdbcprepstmt  = new preparedStmt();
      jdbcprepstmt.insertEmp(jdbcconn, 621, "Kushagr", "Mahajan", 31,
      "Pune", 6000);
      jdbcprepstmt.updateEmp(jdbcconn, 521, 33);
      jdbcprepstmt.displayEmp(jdbcconn, 502);
      jdbcprepstmt.deleteEmp(jdbcconn, 522);

      jdbcconn.close();
    }
    catch(Exception e)
    {
      //Handle errors for Class.forName
      System.out.println(e);
    }
    //end try
    System.out.println("Executed!");
```

```
}
//end main
/*
 * PreparedStatement to Insert the record
 * @parameter jdbcconn
 */
private void insertEmp(Connection jdbcconn, int id, String first,
String last, int age, String city, int salary)
throws SQLException
{
  String insertSQL = "insert into Emp values( ?, ?, ?, ?, ?, ?)";
  PreparedStatement jdbcprepstmt  = null;
  try
  {
    jdbcprepstmt  = jdbcconn.prepareStatement(insertSQL);
    jdbcprepstmt.setInt(1, id);// here 1 is to specify that is it
    the first parameter
    jdbcprepstmt.setString(2, first);
    jdbcprepstmt.setString(3, last);
    jdbcprepstmt.setInt(4, age);
    jdbcprepstmt.setString(5, city);
    jdbcprepstmt.setInt(6, salary);

    int rowcount = jdbcprepstmt.executeUpdate();
    System.out.println("Count of rows inserted " + rowcount);
  }

  finally
  {
    if (jdbcprepstmt  != null)
    {
      jdbcprepstmt.close();
    }
  }
}

/*
 * PreparedStatement to Update the record
 * @parameter jdbcconn, id, age
 */
private void updateEmp(Connection jdbcconn, int id, int age) throws
SQLException
{
  String updateSQL = "Update emp set age = ? where id = ?";
  PreparedStatement jdbcprepstmt  = null;
  try
  {
    jdbcprepstmt  = jdbcconn.prepareStatement(updateSQL);
    jdbcprepstmt.setInt(1, age);
    jdbcprepstmt.setInt(2, id);
    int rowcount = jdbcprepstmt.executeUpdate();
    System.out.println("Count of rows updated " + rowcount);
  }
  finally
  {
    if (jdbcprepstmt  != null)
    {
      jdbcprepstmt.close();
    }
  }
}
/*
```

```
    * PreparedStatement to Delete the record
    * @parameter jdbcconn, id
    */
   private void deleteEmp(Connection jdbcconn, int id) throws
   SQLException
   {
     String deleteSQL = "Delete from emp where id = ?";
     PreparedStatement jdbcprepstmt  = null;
     try
     {
       jdbcprepstmt  = jdbcconn.prepareStatement(deleteSQL);
       jdbcprepstmt.setInt(1, id);
       int rowcount = jdbcprepstmt.executeUpdate();
       System.out.println("Count of rows deleted " + rowcount);
     }
     finally
     {
       if (jdbcprepstmt  != null)
       {
         jdbcprepstmt.close();
       }
     }
   }
   /*
    * PreparedStatement to Display the record
    * @parameter jdbcconn, id
    */
   private void displayEmp(Connection jdbcconn, int id) throws
   SQLException
   {
     String selectSQL = "Select * from emp where id = ?";
     PreparedStatement jdbcprepstmt  = null;
     try
     {
       jdbcprepstmt  = jdbcconn.prepareStatement(selectSQL);
       jdbcprepstmt .setInt(1, id);
       ResultSet jdbcresultset = jdbcprepstmt.executeQuery();
       while (jdbcresultset.next())
       {
         System.out.println("id : " + jdbcresultset.getInt("id") + "
         First : " + jdbcresultset.getString("first") + " Last : "+
         jdbcresultset.getString("last") + " Age : " +
         jdbcresultset.getInt("age") + " City : " +
         jdbcresultset.getString("city") + " Salary : " +
         jdbcresultset.getInt("salary"));
       }
     }

     finally
     {
       if (jdbcprepstmt  != null)
       {
         jdbcprepstmt.close();
       }
     }
   }
}
```

Output:

```
Connecting to database...
Count of rows inserted 1
Count of rows updated 1
id : 502 First : Ajay Last : Tandon Age : 40 City : Mumbai Salary : 9000
Count of rows deleted 1
Executed!
```

The query execution in the relational database goes through the steps of parsing, compilation, optimization and finally, implementation of an optimized query. The Statement interface is required to go through all the steps, but with the PreparedStatement the first three steps are executed while creating the preparedStatement itself. This makes this execution quicker than Statement.

3.7.3 CallableStatement Interface

We have already seen that we can execute the simple SQL query using Statement interface and pre-compiled parameterized SQL query using PreparedStatement. The JDBC API also provides a CallableStatement interface used to implement store procedures and an extension of the PreparedStatement.

The Stored procedure is a set of SQL statements that is stored as a group and resided within the database and can be shared by multiple programs. These stored procedures make the performance better because these are pre-compiled. The pre-compiled procedure is executed with just one call to the database server being in the same database server space, so reduce the network traffic. Further, we can apply any business logic to the database through the concept of stored procedures and functions. For example, we can evaluate the age of the employee using the date of birth. To get the result, we can create a function that can take input as the date of birth and get the output as the age of the employee.

CallableStatement interface is used to call the stored procedures and functions. We can use the prepareCall() method of the Connection interface to get the CallableStatement object.

Syntax

CallableStatement jdbccallstmt = connection.prepareCall("{call PROCEDURE_NAME(?, ?, ?)}");

The symbol '?' is a placeholder they can use to register IN, OUT, and INOUT parameters.

There are three types of parameters used in the stored procedure. The PreparedStatement object generally uses only the IN parameter, but the CallableStatement object can use all three parameters IN, OUT, and INOUT.

The In parameter is used when the SQL statement is created, and the value is unknown, i.e. the IN parameter is used to pass the values to the stored procedure. The OUT parameter value returned after the execution of the query, i.e. OUT parameter used to hold the result returned by the stored procedure. The INOUT parameter provides both input and output values, i.e. IN OUT acts as both IN and OUT parameter. We use the setter method to bind values to IN parameters, and we use the getter method to retrieve values from the OUT parameters.

In our program, we use CallableStatement method registerOutParamter() to register OUT parameter before calling the stored procedure.

Following are the commonly used methods for CallableStatement interface:

execute(): This method is used to execute a query and return a Boolean value. The value is true if a result is a ResultSet object, and it will be false if there are no results or it's an updated count.

executeUpdate(): This method is for DML statements or DDL statements like Insert, Update Create etc.

executeQuery(): This method is for SQL statement that returns ResultSet.

Let us write a simple example of a callable interface. In this example, we are using a stored procedure "employee_details" that receives "id" as the parameter and provide all other details of the employee.

Example 3.7.3:

```
delimiter $$$
DROP PROCEDURE IF EXISTS emp.employee_details $$$
CREATE PROCEDURE emp.employee_details
(IN p_id int,
OUT p_first varchar(30), OUT p_last varchar(30), OUT p_age int, OUT p_city varchar(30), OUT p_salary int
)
BEGIN
 SELECT id, first, last, age, city, salary
 INTO
p_id, p_first, p_last,  p_age, p_city, p_salary
 from EMP
where id = p_id;
END
$$$
```

```
import Java.sql.*;
import Java.sql.DriverManager;
public class callableStmt
{
 public static void main(String[] args)
 {
  Connection jdbcconn = null;
  CallableStatement jdbccallstmt = null;
  try
  {
   Class.forName("com.mysql.cj.jdbc.Driver");
   System.out.println("Connecting to database...");
   jdbcconn =
   DriverManager.getConnection("jdbc:mysql://localhost:3306/emp",
   "root", "admin");
   System.out.println("Creating statement...");
   String SQL = "{call employee_details ( ?,?,?,?,?,?)}";
   jdbccallstmt = jdbcconn.prepareCall(SQL);
   //First Bind the IN parameter and after that bind the OUT parameter
   int p_id = 521;
   jdbccallstmt.setInt(1, p_id); // This would set ID as 21
   // We need to register the OUT parameters
   jdbccallstmt.registerOutParameter(2, Java.sql.Types.VARCHAR);
   jdbccallstmt.registerOutParameter(3, Java.sql.Types.VARCHAR);
   jdbccallstmt.registerOutParameter(4, Java.sql.Types.BIGINT);
   jdbccallstmt.registerOutParameter(5, Java.sql.Types.VARCHAR);
```

```
  jdbccallstmt.registerOutParameter(6, Java.sql.Types.BIGINT);
  // Stored procedure is run using the execute method
  jdbccallstmt .execute();
  //Fetch employee details with getter method
  String p_first = jdbccallstmt.getString(2);
  String p_last = jdbccallstmt.getString(3);
  Integer p_age = jdbccallstmt.getInt(4);
  String p_city = jdbccallstmt.getString(5);
  Integer p_salary = jdbccallstmt.getInt(6);
  System.out.println("Employee Details with ID:" + p_id + " are " +
  p_first + " " + p_last + " " + p_age + " " +  p_city + " " +
  p_salary);
  jdbcconn.close();
 }
 catch (Exception e)
 {
  System.out.println(e);
 }
 }
}
```

Output:

```
Connecting to database...
Creating statement...
Employee Details with ID:521 are Aman Sharma 33 Delhi 8000
```

The JDBC API offers three different interfaces viz. The statement interface executes routine SQL queries, PreparedStatement to execute dynamic or parameterized SQL queries and CallableStatement to implement the stored procedures. The three interfaces significantly differ in terms of functionality and performance. Let us summarize the differences between these three interfaces.

Table: the differences between Statement, PreparedStatement, and CallableStatement interfaces

Statement	PreparedStatement	CallableStatement
execute Normal SQL queries	execute parameterized or dynamic queries	call the stored procedure
Preferred to use when query execution only once like DDL statements	preferred to use when a query executed multiple times	preferred to use when the stored procedure executed
we cannot pass the parameters	accepts the parameters at run time	accepts at run time and uses three different types of parameters IN, OUT, IN OUT
offers low performance	offers better performance when multiple queries are executed	offers very high performance

☞ Check Your Progress 3

1. Why is Statement object required to perform SQL query execution?

2. What is the fundamental difference between the Statement and PreparedStatement interface?

3. What is the return type of execute(), executeQuery() and executeUpdate() and Which method executes any kind of SQL statement?

4. What executeUpdate() method of PreparedStatement interface does?

3.8 SUMMARY

In this unit, we have seen how the JDBC API is used for Database Connectivity for Java applications. JDBC API provides various methods and interfaces for accessible communication with the database, which facilitates an application to execute queries, retrieve results and update data. We have discussed the JDBC drivers, which ensure the database independence and only changing the database driver, and a few lines of code allows us to change the backend database. The JDBC Database Connection Steps to understand the process starting from loading a driver, which opens a communication channel with the database to create a connection through DriverManager class—subsequently creating and executing the statements to fetch the result and finally closing the connection.

The various types of JDBC drivers are available, and there are several factors that may be critical in choosing an appropriate driver for the application. The DriverManager class provides vital services to maintain JDBC drivers and defines an object that connects Java applications to a JDBC driver. Registering and deregistering drivers, locating and storing the list of available drivers, and establishing connections are critical tasks handled by the DriverManager class.

The connection interface is used to represent a session with the connected database and offer various objects for efficient database query execution. The Statement interface generally executes a simple query, the PreparedStatement interface is used to manage repeating queries, and the CallableStatement interface is further improvising on implementing stored procedures.

3.9 SOLUTIONS/ANSWERS TO CHECK YOUR PROGRESS

☞ Check Your Progress 1

1. Following are the necessary steps to create a JDBC application:

   a. The first step is to import all the required packages which contain the JDBC classes.
   b. The second step is to register the appropriate JDBC driver, which can perform the task of opening the communications channel with the database.
   c. The third step is to use the Driver Manager class getConnection() method, which can open a connection.
   d. The fourth step is to use the object of Statement Interface to execute a query.
   e. The fifth step is to extract the data using the ResultSet method.
   f. The sixth step is to close all the connections and thus clean up the environment.

2. The Thin Driver is the fastest pure Java driver which communicates directly through socket connections. It doesn't require any middleware or native library. It thus connects with the database by converting the JDBC method calls directly into the vendor database protocol without requiring any intermediate library.

3. It provides two packages Java.sql.* and Javax.sql.*

☞ Check Your Progress 2

1. The two approaches for registering the Driver are as follows:

   a. The first approach uses the class.forName() method for dynamically loading the Driver's class into the memory, and JVM registers the Driver automatically.
   b. The second approach uses the DriverManager.registerDriver() method, which statically loads the Driver and is generally used in non-JDK compliant JVM.

2. The JDBC DriverManager is a concrete Java class that provides an interface between the user and drivers. The other JDBC components are Java interfaces and

are implemented by the various driver packages. The JDBC driver can create the database connections, but generally, the DriverManager is used to get the connection. The JDBC DriverManager provides elementary services to maintain JDBC drivers and defines an object that connects Java applications to a JDBC driver.

3. The getConnection() method is used to establish a connection with the database and is belongs to DriverManager class.

4. The "exception Java.sql.SQLException: No suitable driver found" exception is due to the unformatted SQL URL string.

5. Yes, we can use multiple catch blocks with a single try statement, and also, the try-catch blocks can be nested similar to if-else statements, but there can only be one finally block with a try-catch statement.

☞ Check Your Progress 3

1. The execution of the SQL queries requires some pre-defined library defined in the form of Statement object, PreparedStatement object and CallableStatement object. Based on the application requirement, we need to create either Statement, PreparedStatement or CallableStatement object.

2. The fundamental difference between the Statement interface and the PreparedStatement interface is that the PreparedStatement interface uses pre-compiled statements for execution and thus offers better performance and safeguards from SQL injection attacks.

3. The execute() return type is Boolean, executeQuery() return type is ResultSet object and executeUpdate() return type is int. The boolean execute() can execute any kind of SQL statement.

4. This method is used for executing a query like create, drop, insert, update, delete etc.

3.10 REFERENCES/FURTHER READING

- Herbert Schildt "Java The Complete Reference", McGraw-Hill,2017
- Horstmann, Cay S., and Gary Cornell, " *Core Java: Advanced Features*" Vol. 2. Pearson Education, 2013.
- Prasanalakshmi, "*Advanced Java Programming*", CBS Publishers 2015
- Sagayaraj, Denis, Karthik and Gajalakshmi ,"*Java Programming – for Core and Advanced Users*", Universities Press 2018
- Sharan, Kishori, "Beginning Java 8 APIs, Extensions and Libraries: Swing, JavaFX, JavaScript, JDBC and Network Programming APIs", Apress, 2014.
- Parsian, Mahmoud, "*DBC metadata, MySQL, and Oracle recipes: a problem-solution approach"* Apress, 2006.
- https://docs.oracle.com/Javase/tutorial/jdbc/basics/index.html
- https://www.roseindia.net/jdbc/