

UNIT-III

JAVA DATABASE CONNECTIVITY

JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database. JDBC works with Java on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

Why to Learn JDBC?

JDBC stands for **Java Database Connectivity**, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Applications of JDBC

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as –

- Java Applications
- Java Applets, Swing and JavaFX applications
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

Structured Query Language (SQL) is a standardized language that allows you to perform operations on a database, such as creating entries, reading content, updating content, and deleting entries.

SQL is supported by almost any database you will likely use, and it allows you to write database code independently of the underlying database.

This chapter gives an overview of SQL, which is a prerequisite to understand JDBC concepts. After going through this chapter, you will be able to **Create, Create, Read, Update, and Delete** (often referred to as **CRUD** operations) data from a database.

Create Database

The CREATE DATABASE statement is used for creating a new database. The syntax is –

```
SQL> CREATE DATABASE DATABASE_NAME;
```

Example

The following SQL statement creates a Database named EMP –

```
SQL> CREATE DATABASE EMP;
```

Drop Database

The DROP DATABASE statement is used for deleting an existing database. The syntax is –

```
SQL> DROP DATABASE DATABASE_NAME;
```

Note: To create or drop a database you should have administrator privilege on your database server. Be careful, deleting a database would loss all the data stored in the database.

Create Table

The CREATE TABLE statement is used for creating a new table. The syntax is –

```
SQL> CREATE TABLE table_name  
(  
    column_name column_data_type,  
    column_name column_data_type,  
    column_name column_data_type  
    ...  
);
```

Example

The following SQL statement creates a table named Employees with four columns –

```
SQL> CREATE TABLE Employees  
(  
    id INT NOT NULL,  
    age INT NOT NULL,  
    first VARCHAR(255),  
    last VARCHAR(255),  
    PRIMARY KEY ( id )  
);
```

Drop Table

The DROP TABLE statement is used for deleting an existing table. The syntax is –

```
SQL> DROP TABLE table_name;
```

Example

The following SQL statement deletes a table named Employees –

```
SQL> DROP TABLE Employees;
```

INSERT Data

The syntax for INSERT, looks similar to the following, where column1, column2, and so on represents the new data to appear in the respective columns –

```
SQL> INSERT INTO table_name VALUES (column1, column2, ...);
```

Example

The following SQL INSERT statement inserts a new row in the Employees database created earlier –

```
SQL> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
```

SELECT Data

The SELECT statement is used to retrieve data from a database. The syntax for SELECT is –

```
SQL> SELECT column_name, column_name, ...  
      FROM table_name  
      WHERE conditions;
```

The WHERE clause can use the comparison operators such as =, !=, <, >, <=, and >=, as well as the BETWEEN and LIKE operators.

Example

The following SQL statement selects the age, first and last columns from the Employees table, where id column is 100 –

```
SQL> SELECT first, last, age  
      FROM Employees  
      WHERE id = 100;
```

The following SQL statement selects the age, first and last columns from the Employees table where *first* column contains *Zara* –

```
SQL> SELECT first, last, age  
      FROM Employees  
      WHERE first LIKE '%Zara%';
```

UPDATE Data

The UPDATE statement is used to update data. The syntax for UPDATE is –

```
SQL> UPDATE table_name  
      SET column_name = value, column_name = value, ...  
      WHERE conditions;
```

The WHERE clause can use the comparison operators such as =, !=, <, >, <=, and >=, as well as the BETWEEN and LIKE operators.

Example

The following SQL UPDATE statement changes the age column of the employee whose id is 100 –

```
SQL> UPDATE Employees SET age=20 WHERE id=100;
```

DELETE Data

The DELETE statement is used to delete data from tables. The syntax for DELETE is –

```
SQL> DELETE FROM table_name WHERE conditions;
```

The WHERE clause can use the comparison operators such as =, !=, <, >, <=, and >=, as well as the BETWEEN and LIKE operators.

Example

The following SQL DELETE statement deletes the record of the employee whose id is 100 –

```
SQL> DELETE FROM Employees WHERE id=100;
```

What is JDBC Driver?

JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server.

For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.

The *Java.sql* package that ships with JDK, contains various classes with their behaviors defined and their actual implementations are done in third-party drivers. Third party vendors implement the *java.sql.Driver* interface in their database driver.

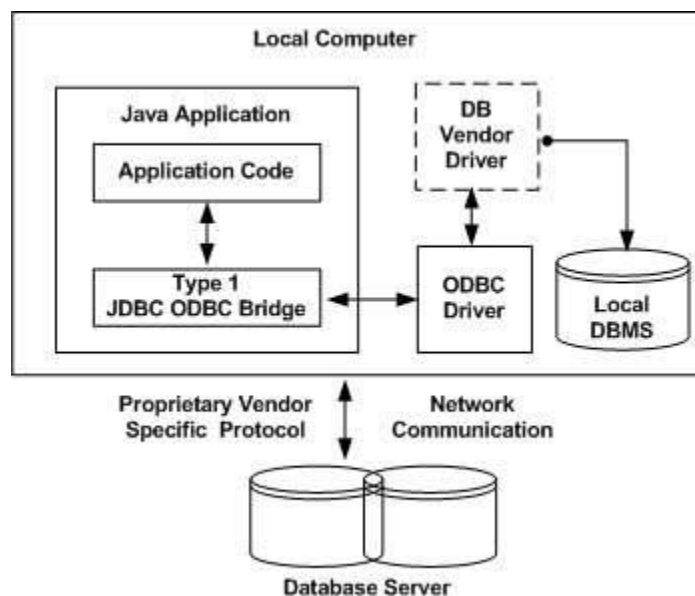
JDBC Drivers Types

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which is explained below –

Type 1: JDBC-ODBC Bridge Driver

In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.

When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.

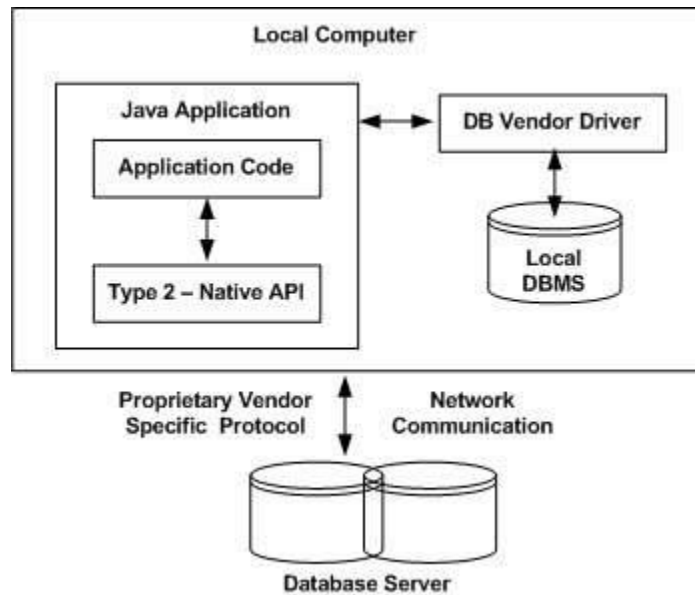


The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

Type 2: JDBC-Native API

In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.

If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.

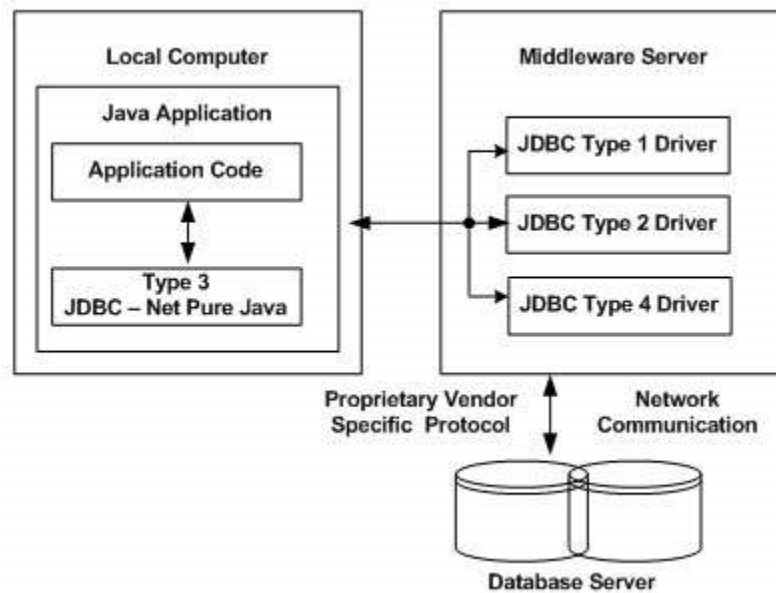


The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

Type 3: JDBC-Net pure Java

In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.

This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.



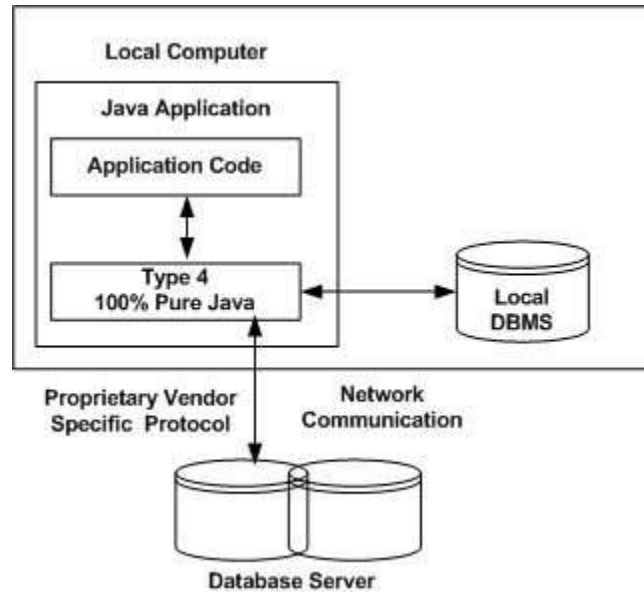
You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.

Your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the nuances will prove helpful.

Type 4: 100% Pure Java

In a Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.

This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.



Which Driver should be Used?

If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.

If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.

Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.

The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

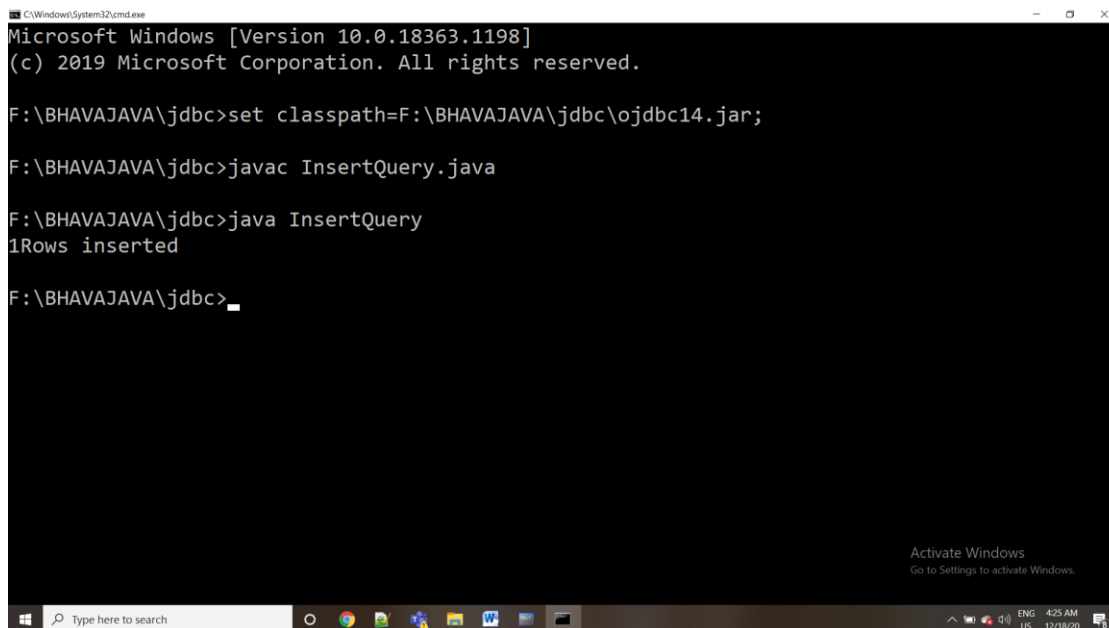
Creating JDBC Application

There are following six steps involved in building a JDBC application –

- **Import the packages:** Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.
- **Register the JDBC driver:** Requires that you initialize a driver so you can open a communication channel with the database.
- **Open a connection:** Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.
- **Execute a query:** Requires using an object of type Statement for building and submitting an SQL statement to the database.
- **Extract data from ResultSet:** Requires that you use the appropriate *ResultSet.getXXX()* method to retrieve the data from the result set.
- **Clean up the environment:** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

Note: To practice JDBC Applications in this Unit,

1. Install oracle10g in your computer with username as “system”, password as “admin”.
2. Set classpath to ojdbc14.jar which is extracted from
“C:\oracle\app\oracle\product\10.2.0\server\jdbc\lib” and copy to working directory.
3. Set classpath using following command



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.18363.1198]
(c) 2019 Microsoft Corporation. All rights reserved.

F:\BHAVAJAVA\jdbc>set classpath=F:\BHAVAJAVA\jdbc\ojdbc14.jar;

F:\BHAVAJAVA\jdbc>javac InsertQuery.java

F:\BHAVAJAVA\jdbc>java InsertQuery
1Rows inserted

F:\BHAVAJAVA\jdbc>
```


Sample Program for JDBC Application Steps and Creating table from JDBC Application:

```
//STEP 1:import the packages
import java.sql.*;
public class JDBCFirst{
    // JDBC driver name and database URL

    static final String jdbcDriver = "oracle.jdbc.driver.OracleDriver";
    static final String dbURL= "jdbc:oracle:thin:@localhost:1521:XE";
    //Database credentials
    static final String user = "system";
    static final String pass = "admin";
    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        try{
            //STEP 2: Register JDBC driver
            Class.forName(jdbcDriver);
            //STEP 3: Open a connection
            System.out.println("Connecting to database...");
            conn = DriverManager.getConnection(dbURL,user,pass);
            //STEP 4: Execute a query
            System.out.println("Creating table");
            stmt = conn.createStatement();
            if(stmt!=null){
                System.out.println("connection established");
                String sqlq = "create table student777
                (\"+\"name varchar(20),\"+\"branch varchar(20))\";
                stmt.executeUpdate(sqlq);

                System.out.println("Table created successfully...");
            }
        }
        catch(ClassNotFoundException|SQLException e){
            e.printStackTrace();
        }
        finally{
            try{
                //finally block used to close resources
                if(stmt!=null)
                    stmt.close();
                if(conn!=null)
                    conn.close();
            }
            catch(SQLException e){
```

```

        e.printStackTrace();
    }
} //end finally try
System.out.println("Goodbye!");
} //end main
} //end class

```

/* Output

F:\BHAVAJAVA\jdbc>javac JDBCFirst.java

F:\BHAVAJAVA\jdbc>java JDBCFirst

Connecting to database...

Creating database...

connection established

Database created successfully...

Goodbye!

*/

Oracle Database URL format:

DBMS: Oracle 10g

Vendor: Oracle Corporation

Driver Type: JDBC driver (Type Driver)

URL format: jdbc:oracle:thin:@<server>:<port>:<DBMSInstance>

(Example Format: jdbc:oracle:thin:@localhost:1521:XE)

Driver class: oracle.jdbc.driver.OracleDriver

To work with Oracle Type 4 driver set classpath to ojdbc14.jar

F:\jdbc>set classpath=F:\jdbc\ojdbc14.jar;

DriverManager class

The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver().

Useful methods of DriverManager class

public static void registerDriver(Driver driver)

Used to register the given driver with DriverManager.

public static void deregisterDriver(Driver driver)

Used to deregister the given driver (drop the driver from the list) with DriverManager.

public static Connection getConnection(String url)

Used to establish the connection with the specified url, username and password.

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

Used to establish the connection with the specified url, username and password.

Connection interface

A Connection is the session between java application and database. The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provide many methods for transaction management like commit(), rollback() etc.

Connection interface methods:

Methods

public Statement createStatement() throws SQLException

creates a statement object that can be used to execute SQL queries.

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

Creates a Statement object that will generate ResultSet objects with the given type and concurrency.
--

public void setAutoCommit(boolean status) throws SQLException
--

is used to set the commit status.By default it is true.

public void rollback() throws SQLException

Drops all changes made since the previous commit/rollback.
--

public void close() throws SQLException
--

closes the connection and Releases a JDBC resources immediately.
--

JDBC uses three different interfaces to represent SQL statements in different formats.

1. Statement
2. PreparedStatement
3. CallableStatement

Statement interface

Executing SQL statement using Statement interface object:

The Statement interface provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

Commonly used methods of Statement interface:

Methods
public ResultSet executeQuery(String sql) Used to execute SELECT query. It returns the object of ResultSet.
public int executeUpdate(String sql) Used to execute specified query, it may be create, drop, insert, update, delete etc.
public boolean execute(String sql) Used to execute queries that may return multiple results.
public int[] executeBatch() Used to execute batch of commands.

Note:

To demonstrate following programs create “studentaits” table in sql prompt(oracle database) with following command.

```
sql>create table studentaits1(roll varchar(20),name varchar(20),branch varchar(20),gender varchar(20),address varchar(20));
```

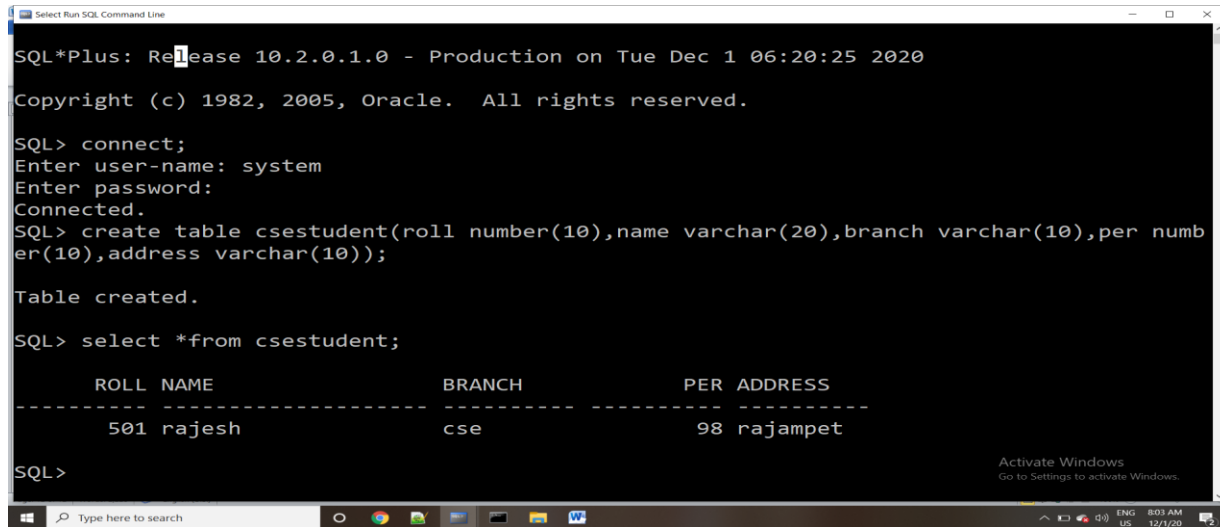
//Example Program to demonstrate executeUpdate method to insert row in database table.

```
import java.sql.*;
public class InsertQuery{
    public static void main(String args[]) throws Exception{
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection con=DriverManager.getConnection(
            "jdbc:oracle:thin:@localhost:1521:XE","system","admin");
        Statement stmt=con.createStatement();
        String iQuery="insert into csestudent(roll,name,branch,per,address)
            values(501,'rajesh','cse',98,'rajampet')";
        int i=stmt.executeUpdate(iQuery);
        if(i>0)
            System.out.println(i+" Rows inserted");
    }
}
```

F:\BHAVAJAVA\jdbc>java InsertQuery

1 Rows inserted

Output in SQL prompt:

A screenshot of a Windows command prompt window titled "Select Run SQL Command Line". The window shows the output of SQL*Plus Release 10.2.0.1.0. The user connects as 'system' and creates a table 'cstudent' with columns 'roll number(10)', 'name varchar(20)', 'branch varchar(10)', 'per number(10)', and 'address varchar(10)'. The table is created successfully. Then, the user runs a select query: 'select *from cstudent;'. The output shows a single row with values: '501', 'rajesh', 'cse', '98', and 'rajampet'.

```
SQL*Plus: Release 10.2.0.1.0 - Production on Tue Dec 1 06:20:25 2020
Copyright (c) 1982, 2005, Oracle. All rights reserved.

SQL> connect;
Enter user-name: system
Enter password:
Connected.
SQL> create table cstudent(roll number(10),name varchar(20),branch varchar(10),per number(10),address varchar(10));

Table created.

SQL> select *from cstudent;

      ROLL NAME          BRANCH          PER ADDRESS
-----
      501 rajesh          cse          98 rajampet

SQL>
```

//Example Program to demonstrate executeUpdate method to update row in database table.

//Step 1:importing java.sql package

import java.sql.*;

public class UpdateQuery{

 public static void main(String args[]) throws Exception{

 //Step 2:Registering with JDBC driver

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

 //Step 3:Getting Connection object

 Connection con=DriverManager.getConnection

 ("jdbc:oracle:thin:@localhost:1521:XE","system","admin");

 //Step 4:Create Statement object

 Statement stmt=con.createStatement();

 String iQuery="update studentaits set address='kadapa' where
 roll='501'";

 //Step 5:SQL Query Execution

 int i=stmt.executeUpdate(iQuery);

 if(i>0)

 System.out.println(i+"Rows updated");

 }

}

Output

F:\BHAVAJAVA\jdbc>javac UpdateQuery.java

F:\BHAVAJAVA\jdbc>java UpdateQuery

1Rows updated

Example Program to demonstrate executeUpdate method to delete row in database table.

```
//Step 1:importing java.sql package
import java.sql.*;
public class DeleteQuery{
    public static void main(String args[]) throws Exception{
        //Step 2:Registering with JDBC driver
        Class.forName("oracle.jdbc.driver.OracleDriver");
        //Step 3:Getting Connection object
        Connection con=DriverManager.getConnection
            ("jdbc:oracle:thin:@localhost:1521:XE","system","admin");
        //Step 4:Create Statement object
        Statement stmt=con.createStatement();
        String iQuery="delete from studentaits where roll='501'";
        //Step 5:SQL Query Execution
        int i=stmt.executeUpdate(iQuery);
        if(i>0)
            System.out.println(i+"Rows deleted");
    }
}
```

Output

```
F:\BHAVAJAVA\jdbc>javac DeleteQuery.java
```

```
F:\BHAVAJAVA\jdbc>java DeleteQuery
```

```
1Rows deleted
```

ResultSet to read from Database table:

The SQL statements that read data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The `java.sql.ResultSet` interface represents the result set of a database query.

A `ResultSet` object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a `ResultSet` object.

Navigating a Result Set

There are several methods in the `ResultSet` interface that involve moving the cursor, including –

S.No	Methods & Description
1	<code>public void beforeFirst() throws SQLException</code> Moves the cursor just before the first row.
2	<code>public void afterLast() throws SQLException</code> Moves the cursor just after the last row.
3	<code>public boolean first() throws SQLException</code> Moves the cursor to the first row.
4	<code>public void last() throws SQLException</code> Moves the cursor to the last row.
5	<code>public boolean absolute(int row) throws SQLException</code> Moves the cursor to the specified row.
6	<code>public boolean relative(int row) throws SQLException</code> Moves the cursor the given number of rows forward or backward, from where it is currently pointing.
7	<code>public boolean previous() throws SQLException</code> Moves the cursor to the previous row. This method returns false if the previous row is off the result set.
8	<code>public boolean next() throws SQLException</code> Moves the cursor to the next row. This method returns false if there are no more rows in the result set.
9	<code>public int getRow() throws SQLException</code> Returns the row number that the cursor is pointing to.

10	public void moveToInsertRow() throws SQLException Moves the cursor to a special row in the result set that can be used to insert a new row into the database. The current cursor location is remembered.
11	public void moveToCurrentRow() throws SQLException Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing

```
//Program to demonstrate executeQuery method to select row in database table.
//Step 1:importing java.sql package
import java.sql.*;
public class SelectQuery{
    public static void main(String args[]) throws Exception{
        //Step 2:Registering with JDBC driver
        Class.forName("oracle.jdbc.driver.OracleDriver");
        //Step 3:Getting Connection object
        Connection con=DriverManager.getConnection
            ("jdbc:oracle:thin:@localhost:1521:XE","system","admin");
        //Step 4:Create Statement object
        Statement stmt=con.createStatement();
        String sQuery="select *from studentaits";
        //Step 5:SQL Query Execution
        ResultSet rs=stmt.executeQuery(sQuery);
        int i=1;
        while(rs.next()){
            System.out.println("-----");
            System.out.println("Student Record:"+i);
            System.out.println("Roll Number:"+rs.getString("roll"));
            System.out.println("Name:"+rs.getString("name"));
            System.out.println("Branch:"+rs.getString("branch"));
            System.out.println("Gender:"+rs.getString("gender"));
            System.out.println("Address:"+rs.getString("address"));
            System.out.println("-----");
            i++;
        }
    }
}
```


Output

```
-----  
F:\BHAVAJAVA\jdbc>javac SelectQuery.java  
F:\BHAVAJAVA\jdbc>java SelectQuery
```

```
-----  
Student Record:1  
Roll Number:502  
Name:ramesh  
Branch:ece  
Gender:Male  
Address:proddatur  
-----
```

PreparedStatement:

The PreparedStatement interface inherited from Statement interface. It is performed over Statement interface to execute SQL statements. The PreparedStatement accepts input parameters at runtime.

PreparedStatement object that represents a precompiled SQL statement. A SQL statement is precompiled and stored in a PreparedStatement object. This object can then be used to efficiently execute this statement multiple times.

Example:

```
PreparedStatement pstmt = con.prepareStatement("UPDATE  
EMPLOYEES SET SALARY = ? WHERE ID = ?");  
pstmt.setInt(1, 15387);  
pstmt.setInt(2, 110592);
```

Methods

public void setInt(int paramIndex, int value)
sets the integer value to the given parameter index.
public void setString(int paramIndex, String value)
sets the String value to the given parameter index.
public void setFloat(int paramIndex, float value)
sets the float value to the given parameter index.
public void setDouble(int paramIndex, double value)
sets the double value to the given parameter index.
public int executeUpdate()
executes the query. It is used for create, drop, insert, update, delete etc.

```
public ResultSet executeQuery()
```

executes the select query. It returns an instance of ResultSet.

```
//Step 1:importing java.sql package
import java.sql.*;
import java.util.*;
public class InsertNQuery{
    public static void main(String args[]) throws Exception{
        //Step 2:Registering with JDBC driver
        Class.forName("oracle.jdbc.driver.OracleDriver");
        //Step 3:Getting Connection object
        Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE",
"system","admin");
        //Step 4:Create Statement object
        PreparedStatement pstmt=con.prepareStatement("insert into
studentaits values(?,?,?,?);");
        Scanner sin=new Scanner(System.in);
        System.out.println("Enter 5 records to insert");
        int i=0;
        while(i<5){
            System.out.println("Enter Record:"+i);
            System.out.println("Enter Roll");
            String sroll=sin.next();
            System.out.println("Enter Name");
            String sname=sin.next();
            System.out.println("Enter Branch");
            String sbranch=sin.next();
            System.out.println("Enter Gender");
            String sgender=sin.next();
            System.out.println("Enter Address");
            String saddress=sin.next();
            pstmt.setString(1,sroll);
            pstmt.setString(2,sname);
            pstmt.setString(3,sbranch);
            pstmt.setString(4,sgender);
            pstmt.setString(5,saddress);
            //Step 5:SQL Query Execution
            int x=pstmt.executeUpdate();
            if(x>0)
                System.out.println(x+"Rows inserted");
            i++;
        }
    }
}
```

Output:

```
F:\BHAVAJAVA\jdbc>javac InsertNQuery.java
F:\BHAVAJAVA\jdbc>java InsertNQuery
Enter 5 records to insert
Enter Record:0
Enter Roll
1201
Enter Name
Abdul
Enter Branch
CSIT
Enter Gender
Male
Enter Address
Rajampet
1Rows inserted
Enter Record:1
Enter Roll
1202
Enter Name
Ashok
Enter Branch
CSIT
Enter Gender
Male
Enter Address
Kadapa
1Rows inserted
Enter Record:2
Enter Roll
1203
Enter Name
Chaitanya
Enter Branch
CSIT
Enter Gender
Male
Enter Address
Kadapa
1Rows inserted
Enter Record:3
Enter Roll
1204
Enter Name
Ganesh
Enter Branch
CSIT
Enter Gender
Male
Enter Address
Chittor
1Rows inserted
Enter Record:4
Enter Roll
1205
Enter Name
Gopi
Enter Branch
CSIT
Enter Gender
Male
Enter Address
Naidupet
1Rows inserted
```

Committing and Rollback Transactions:

When you connect to a database, the auto-commit property for the Connection object is set to true by default. If Connection is not an auto-commit mode, you must call the commit () or rollback () method of the Connection object to commit or rollback the transaction.

For Example,

```
Connection con=DriverManager.getConnection(dbURL,userid,password);
con.setAutoCommit(false);
```

Methods in Connection interface for committing and Rollbacking transactions
public void setAutoCommit(boolean status) throws SQLException is used to set the commit status. By default it is true.
public void commit() throws SQLException Commits the transaction.
public void rollback() throws SQLException Drops all changes made since the previous commit/rollback.
public void close() throws SQLException closes the connection and Releases a JDBC resources immediately.

```
//Step 1:importing java.sql package
import java.sql.*;
public class SetAuto{
    public static void main(String args[]) throws Exception{
        //Step 2:Registering with JDBC driver
        Class.forName("oracle.jdbc.driver.OracleDriver");
        //Step 3:Getting Connection object
        Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","sys
tem","admin");
        con.setAutoCommit(false);
        //Step 4:Create Statement object
        Statement stmt=con.createStatement();
        String uQuery="update studentaits set address='Simla' where
roll='504'";
        //Step 5:SQL Query Execution
        int i=stmt.executeUpdate(uQuery);
        con.commit();
        if(i>0)
            System.out.println(i+"Rows updated");
    }
}
```

```
}
```

Output:

```
F:\BHAVAJAVA\jdbc>javac SetAuto.java
F:\BHAVAJAVA\jdbc>java SetAuto
1Rows updated
```

Batch Updates:

A JDBC batch update is a batch of updates grouped together, and sent to the database in one batch, rather than sending the updates one by one.

Sending a batch of updates to the database is faster than sending them one by one.

Batch processing allows you to group of related SQL statements into a batch and submit them with one call to the database.

Batch processing can be called with Statement object or PreparedStatement object.

For adding SQL statements to the batch we will call addBatch() method of Statement interface.

Syntax:

```
public void addBatch(String query)
```

For executing batch of SQL queries call executeBatch() method of Statement interface.

```
public int[] executeBatch() throws Exception
```

Batch Processing with PreparedStatement object:

1. Create Statement(or) PreparedStatement object using createStatement() (or) prepareStatement () method.
2. Set auto-commit to false using setAutoCommit () method.
3. Add as many as SQL statements you like into batch using addBatch () method on created Statement object.
4. Execute all SQL statements using executeBatch() on created PreparedStatement object.
5. Finally call commit() or rollback() method.

Batching with Statement Object:

```
Statement stmt=con.createStatement();
con.setAutoCommit(false);
String sql1="insert into Employee values (200,'zeus','cse', 30);
stmt.addBatch(sql1);
String sql2="insert into employee values(201,'siva','ece',32);
stmt.addBatch(sql2);
String sql3="update employee set age=35 where id=100";
stmt.addBatch(sql3);
int[] count=stmt.executeBatch();
```

```
//Step 1:importing java.sql package
import java.sql.*;
public class BatchUpdate{
    public static void main(String args[]) throws Exception{
        //Step 2:Registering with JDBC driver
        Class.forName("oracle.jdbc.driver.OracleDriver");
        //Step 3:Getting Connection object
        Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","sys
tem","admin");
        con.setAutoCommit(false);
        //Step 4:Create Statement object
        Statement stmt=con.createStatement();
        String q1="insert into studentaits
values (577,'Sankar','ece',27,'bangalore')";
        stmt.addBatch(q1);
        String q2="insert into studentaits
values (579,'hari','eee',37,'chennai')";
        stmt.addBatch(q2);
        String q3="update studentaits set address='Boyanapalli' where
roll='504'";
        stmt.addBatch(q3);
        stmt.executeBatch();
        con.commit();

    }
}
/*
F:\BHAVAJAVA\jdbc>javac BatchUpdate.java
F:\BHAVAJAVA\jdbc>java BatchUpdate
*/
```

Processing ResultSet:

A set of rows obtained by executing a SQL statement in a database is known as ResultSet. JDBC lets you execute a “select” statement in the database and process the returned Resultset in the java program using an instance of ResultSet interface.

What is ResultSet?

- When you execute a query in a database, it returns the matching records in the form of ResultSet.
- A ResultSet object also contains the information about properties of the columns in the ResultSet such as datatype of columns, name of columns etc.
- A ResultSet object maintains a cursor which points to a row in the ResultSet. It works similar to a cursor object in database program. The cursor object can point one row at a time.
- The row which it point in time called as current row.
- Properties of ResultSet:
 1. Scrollability
 2. Concurrency
 3. Holdability

Scrollability: Determines the ability of the ResultSet to scroll through the rows in forward direction and backward direction. By default a ResultSet is scrollable only in forward direction.

Concurrency: Concurrency refers to its ability of the ResultSet to update data. By default ResultSet is read only and it does not let you update its data.

Holdability: Holdability refers to the state of the ResultSet after a transaction that it is associated with has been committed.

The properties of the ResultSet we may add from createStatement() method (or) prepareStatement() method of Connection interface.

```
public Statement createStatement (int resultSetType,int resultSetConcurrency,  
                                int resultSetScrollability)
```

```
public Statement createStatement (int resultSetType,int resultSetConcurrency)
```

```
public PreparedStatement prepareStatement (int resultSetType,int resultSetConcurrency,  
                                           int resultSetScrollability)
```

```
public PreparedStatement prepareStatement (int resultSetType,int resultSetConcurrency)
```

ResultSet type: Three constants defined in ResultSet interface to define the ResultSet type.

1. **ResultSet.TYPE_FORWARD_ONLY**: Allows a ResultSet object to move only in forward direction only.
2. **ResultSet.TYPE_SCROLL_INSENSITIVE**: Allows a ResultSet to move in forward and backward direction. It makes the changes in the underlying database made by other transactions.
3. **ResultSet.TYPE_SCROLL_SENSITIVE** : Allows ResultSet to move both forward and backward direction.

Concurrency refers to the ability of the ResultSet to update the data.

1. **ResultSet.CONCUR_READ_ONLY**: Makes the ResultSet read only.
2. **ResultSet.CONCUR_UPDATABLE**: Makes a ResultSet updatable.

Holdability refers to the state of the ResultSet after a transaction that is associated with has been committed.

A ResultSet may be closed or kept open when the transaction is committed.

1. **ResultSet. HOLD_CURSORS_OVER_COMMIT**: Keeps the ResultSet open after the transaction is committed.
2. **ResultSet. CLOSE_CURSORS_AT_COMMIT**: Closes the ResultSet after the transaction is committed.

ResultSet interface methods can be broken into three categories.

1. **Navigational methods**: Used to move the cursor around.
2. **Get methods**: Used to view the data in the columns of current row being pointed by cursor.
3. **Update methods**: Used to update data in the columns of current row.

Navigating a ResultSet:

SNO	Method Name
1.	public void beforeFirst() throws SQLException Moves the cursor before first row.
2.	public void afterLast() throws SQLException Moves the cursor just after the last row.
3.	public boolean first() throws SQLException Moves the cursor to the first row.
4.	public boolean last() throws SQLException Moves the cursor to the last row.
5.	public boolean absolute(int row) throws SQLException Moves the cursor to the specified row number. It accepts positive and negative numbers. If a positive row is specified the row is counted from beginning. If a negative row number is specified, the row number is counted from the end.
6.	public boolean relative(int rows) Moves the cursor in forward or backward direction by specified number of rows from its current position.

7.	public boolean previous() throws SQLException Moves the cursor to the previous row.
8.	public boolean next() throws SQLException Moves the cursor to the next row.
9.	public int getRow() throws SQLException Returns the row number that the cursor is pointing to.

Viewing a ResultSet:

SNO	Method Name
1.	public int getInt(String columnName) throws SQLException Returns the integer in the current row in the column named column Name
2.	public int getInt(int columnIndex) throws SQLException Returns the integer in the current row in specified column index.

Making Changes to ResultSet(Updating a ResultSet):

The ResultSet interface contains a collection of update methods for updating the data in a ResultSet.

SNO	Method Name
1.	public void updateString(int columnIndex,String s) throws SQLException Changes the “String” in the specified column to the value of ‘s’.
2.	public void updateRow() Updates the underlying database with the new contents of current row of the ResultSet object.
3.	public void insertRow() Inserts the contents of the insert row into the ResultSet object and into the database object.
4.	public void moveToInsertRow() Moves the cursor to the insert row. The insert row is a special row with an Updatable ResultSet. It is essentially a buffer where new row may be constructed by calling update methods prior to the inserting the row into the ResultSet.
5.	public void deleteRow() Deletes the current row from the ResultSet object from underlying database table.

```

//Inserting Row through ResultSet Object.
//InsertResult.java
import java.sql.*;
public class InsertResult{
    public static void main(String args[]) throws Exception{
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","system","admin");
        con.setAutoCommit(false);
        Statement
stmt=con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_UPDATABLE);
        String sQuery="select roll,name,branch,gender,address from studentaits";
        ResultSet rs=stmt.executeQuery(sQuery);
        int i=1;
        rs.beforeFirst();
        rs.moveToInsertRow();
        rs.updateString("roll","599");
        rs.updateString("name","Siva Kumar");
        rs.updateString("branch","cse");
        rs.updateString("gender","male");
        rs.updateString("address","Siddavatam");
        rs.insertRow();
        con.commit();
    }
}

/*Output
F:\BHAVAJAVA\jdbc>javac InsertResult.java
F:\BHAVAJAVA\jdbc>java InsertResult
*/

//Updating Row through ResultSet object.
//UpdateResult.java
import java.sql.*;
public class UpdateResult{
    public static void main(String args[]) throws Exception{
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","system","admin");
        Statement
stmt=con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_UPDATABLE);
        String sQuery="select roll,name,branch from studentaits";
        ResultSet rs=stmt.executeQuery(sQuery);
        int i=1;
        rs.beforeFirst();
        while(rs.next()){
            if((rs.getString("branch")).equals("cse")){
                String newName=rs.getString("name")+" sir";
                rs.updateString("name",newName);
                rs.updateRow();
            }
        }
    }
}

```

```

        rs.beforeFirst();
        System.out.println("roll \t name \t\t branch");
        while(rs.next()){

System.out.println(rs.getString("roll")+"\t"+rs.getString("name")+"\t"+rs.getString("branch"));
        }
    }
}

/*
F:\BHAVAJAVA\jdbc>javac UpdateResult.java
F:\BHAVAJAVA\jdbc>java UpdateResult
roll      name      branch
501      Sri ravi sir sir      cse
502      Sri chandra sir sir      cse
503      Sri siva sir sir      cse
577      Raghu sir      cse
579      Rahim ece
507      raghu sir      cse
108      ramesh ce
108      ramesh ce
108      ramesh ce
108      ramesh ce
1201     Abdul CSIT
1202     Ashok CSIT
1203     Chaitanya      CSIT
1204     Ganesh CSIT
1205     Gopi CSIT
599     Siva Kumar sir cse
401     eswar ece
402     nisar ece
403     kiran eee
201     mahesh me
202     venkatesh      me
101     Sri lakshmi      ce
*/

```

CallableStatement:

The **CallableStatement** interface provides methods to execute the stored procedures and functions.

CallableStatement is derived interface from PreparedStatement interface which is derived from Statement interface.

The stored procedures are group of statements (SQL query statements) that will precompiled for execution of task.

Creating a CallableStatement

You can create an object of the **CallableStatement** (interface) using the **prepareCall()** method of the **Connection** interface. This method accepts a string variable representing a query to call the stored procedure and returns a **CallableStatement** object.

A Callable statement can have input parameters, output parameters or both. To pass input parameters to the procedure call you can use place holder and set values to these using the setter methods (setInt(), setString(), setFloat()) provided by the CallableStatement interface.

In Connection interface, we have prepareCall() method to get the CallableStatement object.

Syntax:

public CallableStatement prepareCall(String sqlprocedure) throws Exception

Suppose you have a procedure name myProcedure in the database you can prepare a callable statement as:

```
//Preparing a CallableStatement  
String sqlp="{call myProcedure(?,?,?)}";  
CallableStatement cstmt = con.prepareCall(sqlp);
```

Setting values to the input parameters

You can set values to the input parameters of the procedure call using the setter methods.

These accepts two arguments, one is an integer value representing the placement index of the input parameter and, the other is a int or, String or, float etc... representing the value you need to pass as input parameter to the procedure.

Note: Instead of index you can also pass the name of the parameter in String format.

```
cstmt.setString(1, "Raghav");  
cstmt.setInt(2, 3000);  
cstmt.setString(3, "Hyderabad");
```

Executing the Callable Statement

Once you have created the CallableStatement object you can execute it using one of the **execute()** method.

```
cstmt.execute();
```

In brief, the steps for working with CallableStatement is

1. Construct the stored procedure call in a string format using the JDBC standard syntax.

```
String sql = "{call getName(?,?)}";
```

2. Prepare a CallableStatement using the SQL syntax created in the previous step.

```
CallableStatement cstmt = conn.prepareCall(sql);
```

3. Set any IN parameters that need to be passed to the stored procedure.

For example,

```
cstmt.setString(1, "Raghu");
```

```
cstmt.registerOutParameter(2,java.sql.Types.VARCHAR);
```

Set(Register) the OUT parameter as java.sql.Types type.

For example,

```
cstmt.setInt(1, 101);
```

```
cstmt.registerOutParameter(2,java.sql.Types.VARCHAR);
```

4. Call the executeQuery() method or execute() method of the CallableStatement object.

```
cstmt.execute()
```

```
ResultSet rs = cstmt.executeQuery();
```

5. Get the ResultSet object, which is passed back in the second OUT parameter using the getObject() method and cast it as ResultSet.(If any ResultSEt object returned)

```
ResultSet rs = (ResultSet)cstmt.getObject(2);
```

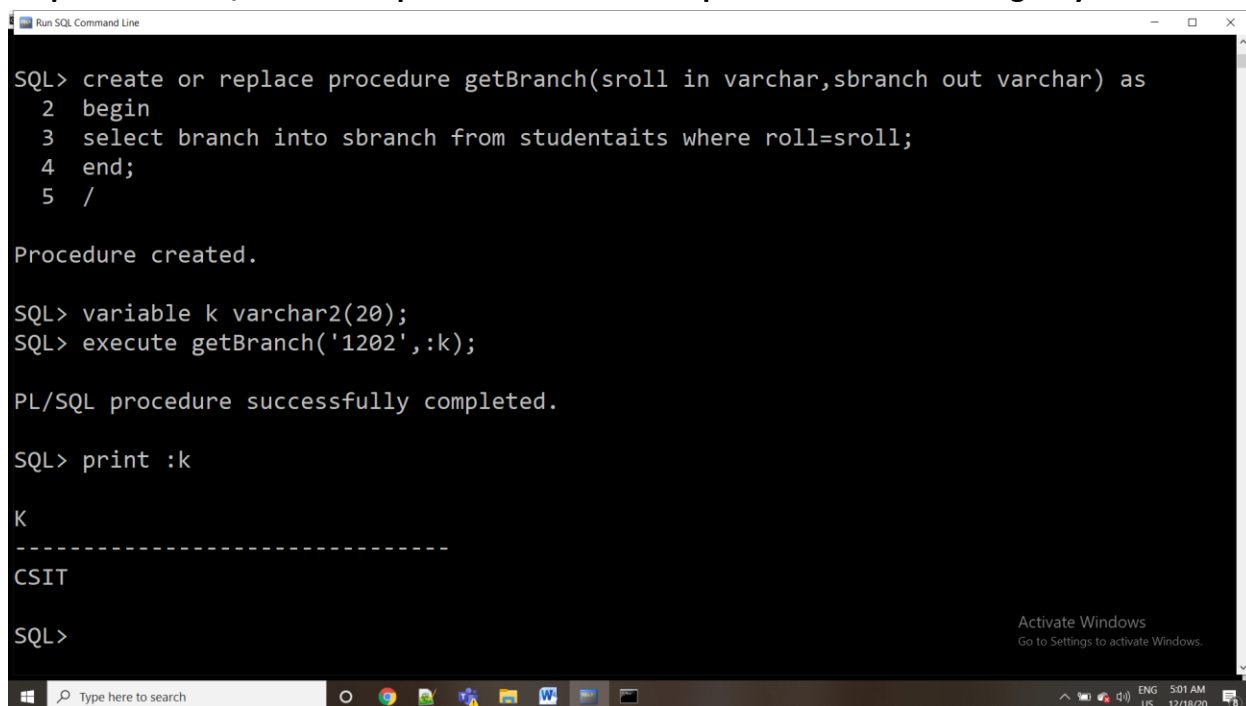
6. Process the ResultSet object as usual by looping through its rows and using its getXxx() methods to read the column values

CallableStatement interface methods

SNO	Method
1.	public void setString(int parameterIndex) throws SQLException This method is used to setting parameters to stored procedure.
2.	public String getString(int parameterIndex) throws SQLException This method is used to get the value of a CHAR, VARCHAR, or LONGVARCHAR parameter as a Java String.
3.	public int getInt(int parameterIndex) throws SQLException Get the value of an INTEGER parameter as a Java int.
4.	public void registerOutParameter(int parameterIndex, int sqlType) throws SQLException Before executing a stored procedure call, you must explicitly call registerOutParameter to register the java.sql.Types of each out parameter. <pre>public class Types extends Object { static int VARCHAR; static int DATE; static int BOOLEAN; static int BLOB; static int ARRAY; ----- ----- }</pre>

//Sample Program

Step1 : Create PL/SQL stored procedure in SQL Prompt as shown in following way.



```
Run SQL Command Line

SQL> create or replace procedure getBranch(sroll in varchar,sbranch out varchar) as
2  begin
3  select branch into sbranch from studentaits where roll=sroll;
4  end;
5  /

Procedure created.

SQL> variable k varchar2(20);
SQL> execute getBranch('1202',:k);

PL/SQL procedure successfully completed.

SQL> print :k

K
-----
CSIT

SQL>
```

Activate Windows
Go to Settings to activate Windows.

Type here to search

ENG 5:01 AM 12/18/20

Step 2:Write JDBC Program to invoke getBranch() stored procedure.

```
//CallableEx.java
import java.sql.*;
public class CallableEx{
    public static void main(String args[]){
        try{
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","sys
tem","admin");

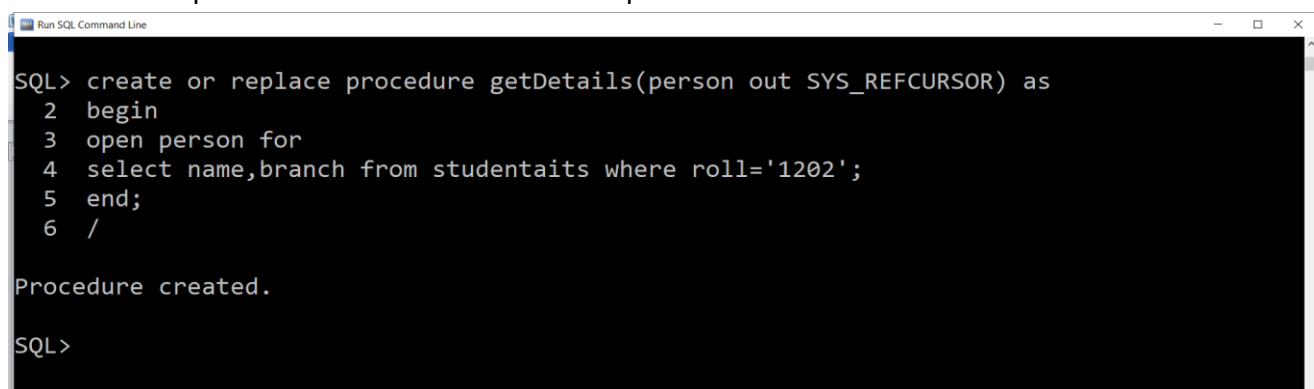
            String sql="{call getBranch(?,?)}";
            CallableStatement cstmt=con.prepareCall(sql);
            cstmt.setString(1,"1201");
            cstmt.registerOutParameter(2,java.sql.Types.VARCHAR);
            cstmt.execute();
            String branch=cstmt.getString(2);
            System.out.println("501 branch is:"+branch);
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
/*
F:\BHAVAJAVA\jdbc>javac CallableEx.java
F:\BHAVAJAVA\jdbc>java CallableEx
501 branch is:CSIT

*/
```

Handling Multiple Results from Statement:

Sometimes executing a SQL statement/Stored procedure may return multiple results. Typically, you get multiple results by executing a stored procedure.

Create stored procedure as shown in below snapshot:

A screenshot of a 'Run SQL Command Line' window. The window has a title bar with a small icon and the text 'Run SQL Command Line'. The main area is a black terminal with white text. The text shows a SQL command to create or replace a procedure named 'getDetails' that takes a 'person' parameter and returns a 'SYS_REFCURSOR'. The command is: 'SQL> create or replace procedure getDetails(person out SYS_REFCURSOR) as', followed by a numbered list: '2 begin', '3 open person for', '4 select name,branch from studentaits where roll='1202';', '5 end;', and '6 /'. Below the command, it says 'Procedure created.' and 'SQL>'.

Using a following method declared in CallableStatement to access multiple results from ResultSet object.

public Object getObject(String parameterName) throws SQLException

Retrieves the value of a parameter as an Object in the Java programming language.

If the value is an SQL NULL, the driver returns a Java null.

This method returns a Java object whose type corresponds to the JDBC type that was registered for this parameter using the method registerOutParameter. By registering the target JDBC type as **oracle.jdbc.OracleTypes.CURSOR**, this method can be used to read database-specific abstract data types.

```
import java.sql.*;
import java.util.*;
class MoreCallable{
    public static void main(String args[]) throws Exception{
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","sys
tem","admin");
        String sp="{call getDetails(?)}";
        CallableStatement cstmt=con.prepareCall(sp);
        cstmt.registerOutParameter(1,oracle.jdbc.OracleTypes.CURSOR);
        cstmt.execute();
        ResultSet rs=(ResultSet)cstmt.getObject(1);
        if(rs!=null){
            while(rs.next()){
                System.out.println("Student Name:"+rs.getString(1));
                System.out.println("Student Branch:"+rs.getString(2));
            }
        }
    }
}
/*
F:\BHAVAJAVA\jdbc>javac MoreCallable.java
F:\BHAVAJAVA\jdbc>java MoreCallable
Student Name:Ashok
Student Branch:CSIT
*/
```

Knowing about Database:

DatabaseMetaData interface provides information about the database. This interface is implemented by driver vendors to let users know the capabilities of a database management system .

In order to get metadata information call the following method from **Connection** interface.

public DatabaseMetaData getMetaData() throws SQLException

DatabaseMetaData methods

1. **public String getDriverName()throws SQLException:** it returns the name of the JDBC driver.
2. **public String getDriverVersion()throws SQLException:** it returns the version number of the JDBC driver.
3. **public String getURL():**It returns the URL of the connected database.
4. **public String getUsername()throws SQLException:** it returns the username of the database.
5. **public String getDatabaseProductName()throws SQLException:** it returns the product name

```
import java.sql.*;
public class DatabaseMeta{
    public static void main(String args[]){
        try{
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE",
                                "system","admin");

            DatabaseMetaData dbmd=con.getMetaData();
            String dname=dbmd.getDriverName();
            String url=dbmd.getURL();
            String dversion=dbmd.getDriverVersion();
            String user=dbmd.getUsername();
            String pname=dbmd.getDatabaseProductName();
            String pversion=dbmd.getDatabaseProductVersion();
            boolean bupdate=dbmd.supportsBatchUpdates();
            System.out.println("Database name="+dname);
            System.out.println("URL="+url);
            System.out.println("Database Version="+dversion);
            System.out.println("Database User Name="+user);
            System.out.println("Database Product name="+pname);
            System.out.println("Database Product Version="+pversion);
            System.out.println("Support Batch Updates?="+bupdate);

        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
/*
F:\BHAVAJAVA\jdbc>javac DatabaseMeta.java
F:\BHAVAJAVA\jdbc>java DatabaseMeta
Database name=Oracle JDBC driver
URL=jdbc:oracle:thin:@localhost:1521:XE
Database Version=10.2.0.1.0XE
Database User Name=SYSTEM
Database Product name=Oracle
Database Product Version=Oracle Database 10g Express Edition Release 10.2.0.1.0 -
Production
Support Batch Updates?=true
*/
```

JDBC-types to Java-Types Mapping:

The JDBC API allows you to access and manipulate data stored in a database in java programming environment. The database uses its own datatypes, whereas java uses its own.

Database type mapping between JDBC and Java **(Learn any 10 mappings from below table)**

JDBC Type	Java Type
ARRAY	java.sql.Array
BIGINT	long
BINARY	byte[]
BIT	boolean
BLOB	java.sql.Blob
BOOLEAN	boolean
CHAR	String
CLOB	java.sql.Clob
DATALINK	java.net.URL
DATE	java.sql.Date
DATE	java.time.LocalDate
DECIMAL	java.math.BigDecimal
DOUBLE	double
FLOAT	double
INTEGER	int
JAVA_OBJECT	underlying Java class
LONGNVARCHAR	String
LONGVARBINARY	byte[]
LONGVARCHAR	String
NCHAR	String
NCLOB	java.sql.NClob
NUMERIC	java.math.BigDecimal
NVARCHAR	String
REAL	float
REF	java.sql.Ref
REF_CURSOR	java.sql.ResultSet
ROWID	java.sql.RowId
SMALLINT	short
SQLXML	java.sql.SQLXML
STRUCT	java.sql.Struct
TIME	java.sql.Time
TIME	java.time.LocalDateTime
TIME_WITH_TIMEZONE	java.time.OffsetTime
TIMESTAMP	java.sql.Timestamp
TIMESTAMP_WITH_TIMEZONE	java.time.OffsetDateTime
TINYINT	byte
VARBINARY	byte[]
VARCHAR	String

Transaction Isolation Level:

In a multi-user database, you will often come across the following two terms:

- Data concurrency
- Data consistency

Data concurrency refers to the ability of multiple users to use the same data concurrently.

Data consistency refers to the accuracy of the data that is maintained when multiple users are manipulating the data concurrently.

As the data concurrency increases (i.e. more users work on the same data), care must be taken to maintain a desired level of data consistency. A database maintains data consistency using locks and by isolating one transaction from another.

Let's look at three phenomena where data consistency may be compromised in a multi-user environment where multiple concurrent transactions are supported.

1. Dirty Read

In a dirty read, a transaction reads uncommitted data from another transaction. Consider the following sequence of steps, which results in inconsistent data because of a dirty read:

- **Transaction A inserts a new row in a table and it has not committed it yet.**
- **Transaction B reads the uncommitted row inserted by the transaction A.**
- **Transaction A rollbacks the changes.**
- At this point, transaction B is left with data for a row that does not exist.

2. Non-Repeatable Read

In a non-repeatable read, when a transaction re-reads the data, it finds that the data has been modified by another transaction that has been already committed. Consider the following sequence of steps, which results in inconsistent data because of a non-repeatable read:

- **Transaction A reads a row.**
- **Transaction B modifies or deletes the same row and commits the changes.**
- **Transaction A re-reads the same row and finds that the row has been modified or deleted.**

3. Phantom Read

In a phantom read, when a transaction re-executes the same query, it finds more data that satisfies the query.

Consider the following sequence of steps, which results in inconsistent data, because of a phantom read:

- **Transaction A executes a query (say Q) and finds X number of rows matching the query.**
- **Transaction B inserts some rows that satisfy the query Q criteria and commits.**
- **Transaction A re-executes the same query (Q) and finds Y number of rows ($Y > X$) matching the query.**

The ANSI SQL-92 standard defines four transaction isolation levels in terms of the above-described three Situations for data consistency. Each isolation level defines what kinds of data inconsistencies are allowed, or not allowed. The four transaction isolation levels are as follows:

- Read Uncommitted • Read Committed
- Repeatable Read • Serializable

Four Isolation Levels Defined by ANSI SQL-92

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
Read Uncommitted	Permitted	Permitted	Permitted
Read Committed	Not Permitted	Permitted	Permitted
Repeatable Read	Not Permitted	Not Permitted	Permitted
Serializable	Not Permitted	Not Permitted	Not Permitted

Java defines the following four constants in the **Connection interface** that correspond to the four isolation levels defined by the ANSI SQL-92 standard:

- **TRANSACTION_READ_UNCOMMITTED**
- **TRANSACTION_READ_COMMITTED**
- **TRANSACTION_REPEATABLE_READ**
- **TRANSACTION_SERIALIZABLE**

You can set the isolation level of a transaction for a database connection using the following method of Connection interface.

```
public void setTransactionIsolation(int isolationLevel)
```

Setting isolation level in the following way.

```
// Get a Connection object
Connection conn = get a connection object...;
// Set the transaction isolation level to read committed
conn.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
```