# DATABASE PROGRAMMING IN JAVA

## Session: 1

# JDBC Concepts

# Objectives

- Explain JDBC architecture and database connectivity
- Explain creation of a JDBC application
- Explain creation of queries and different types of queries
- Explain exception handling in a JDBC application
- Explain the use of Stored Procedures and Transactions
- Describe RowSets
- Explain JDBCRowSet and CachedRowSet

# Database Connectivity

- To connect the Java applications with the databases, the Application Programming Interfaces (APIs) software for database connectivity, known as JDBC, is used.
- Open Database Connectivity (ODBC) and JDBC are two widely used APIs for such activities.

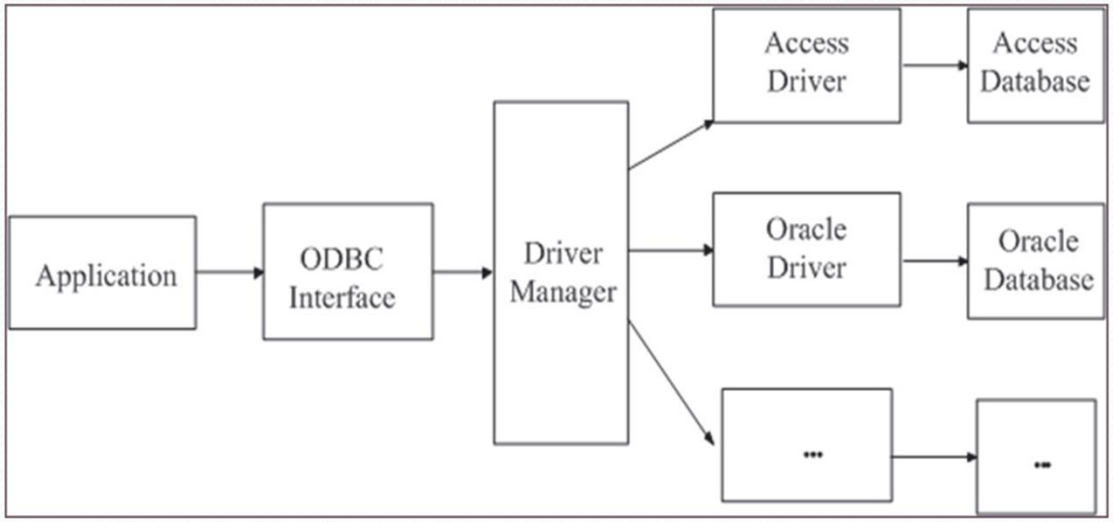
# Open Database Connectivity (ODBC)

ODBC is an API provided by Microsoft for accessing the database.

It uses Structured Query Language (SQL) as its database language.

The driver manager is part of Microsoft ODBC API and is used to manage various drivers in the system.

The following figure shows how an ODBC connection interacts with various components in the application:



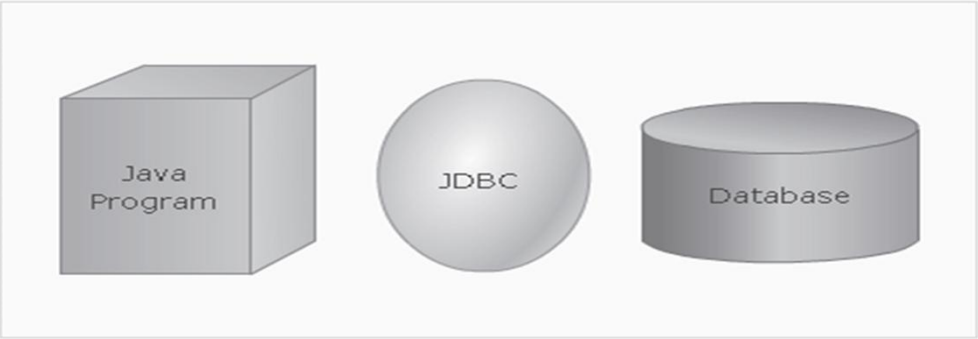
# Java Database Connectivity (JDBC)

JDBC is a Java-based API, which provides a set of classes and interfaces written in Java to access and manipulate different kinds of databases.

The JDBC API has a set of classes and interfaces used for accessing tabular data.

The advantage of using JDBC API is that an application can access any database and run on any platform having Java Virtual Machine (JVM).

Following figure shows JDBC connectivity with the database:



# Need for JDBC

ODBC is written in native C language whereas JDBC is based on Java.

ODBC makes use of pointers whereas JDBC does not use pointers as Java does not allow pointer operations.

Since JDBC is built on ODBC, it retains the basic features of ODBC, it builds additional features on ODBC and is a natural interface for Java applications.

# Product Components of JDBC

## JDBC API

- JDBC API allows applications to execute SQL statements, retrieve results, and make changes to the underlying database.

## JDBC Driver Manager

- A Java application is connected to a JDBC driver using an object of the `DriverManager` class.

## JDBC Test Suite

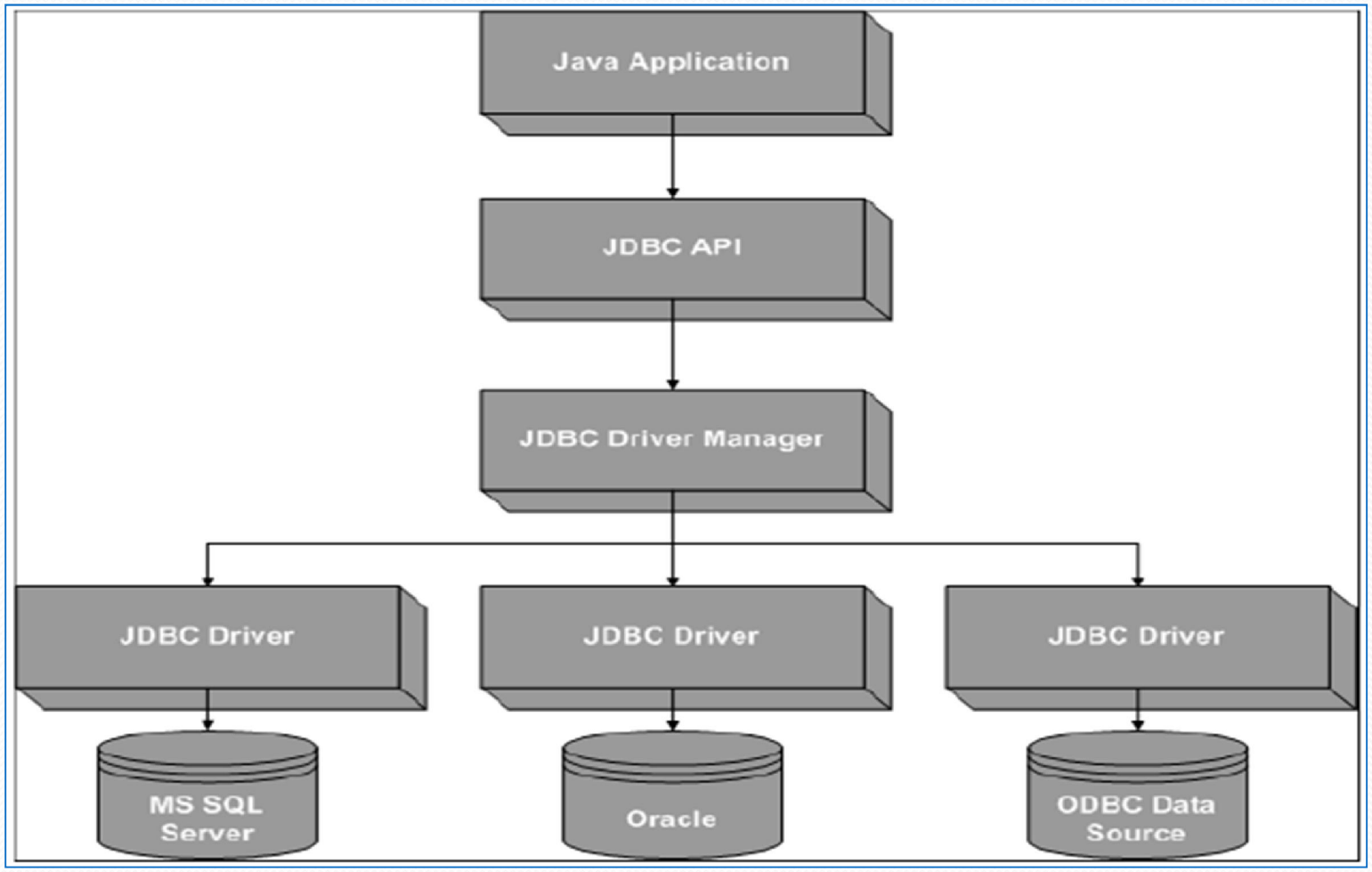
- JDBC driver test suite helps to determine that JDBC drivers will run the program.

## JDBC-ODBC Bridge

- JDBC access is provided using ODBC drivers.

# JDBC Architecture 1-2

Following figure shows the JDBC architecture:



# JDBC Architecture 2-2

## Two-Tier Model

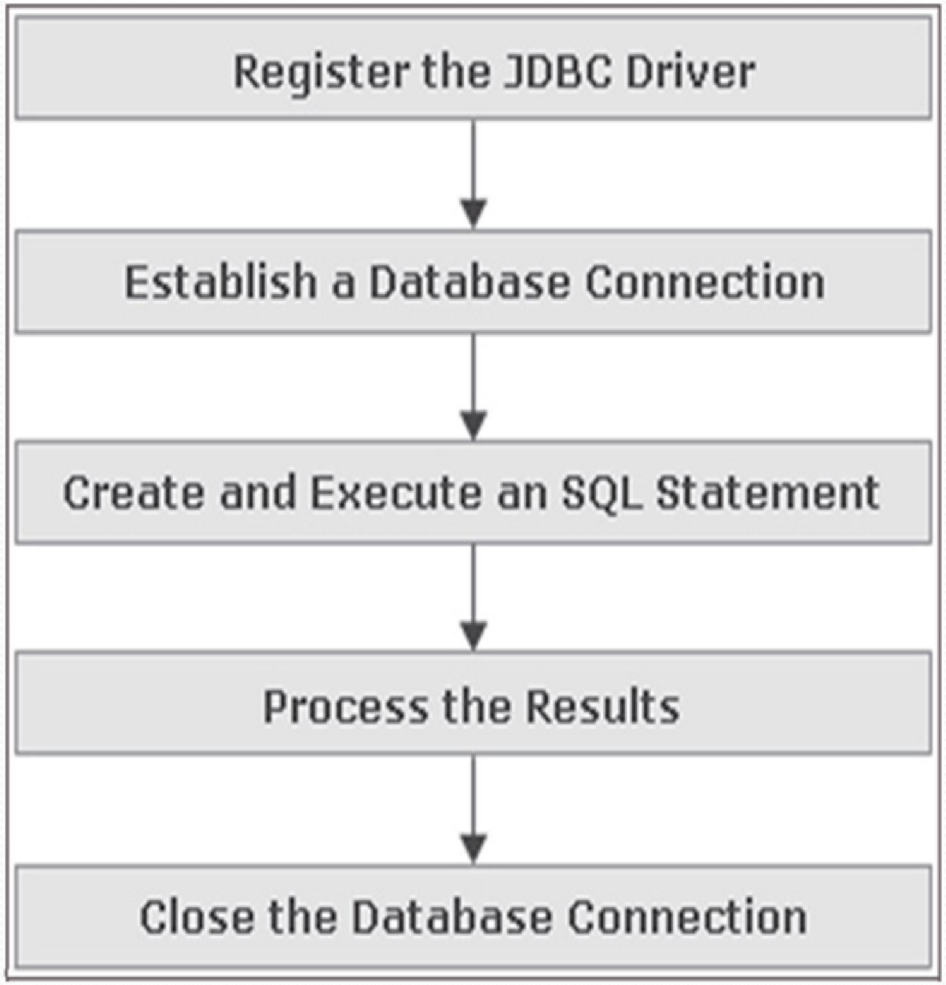
In two-tier model the application program initiates a database access, following which appropriate driver accesses the database and returns the required results to the application program.

## Three-Tier Model

In three-tier model there is an additional middle layer between the application program and the database. The middle layer enforces access control of the data in the database.

# Steps to Develop a JDBC Application

The following figure shows various steps in creating a JDBC connection:



# Registering the JDBC Driver

Database drivers can be registered using different methods:

- `Class.forName()` method

  The parameter for the `Class.forName()` method is the driver location

- `DriverManager.getInstance()` method

  The parameter of the `getInstance()` method is driver location

# Establishing a Connection

- In order to establish a database connection a `Connection` object is created with appropriate URL of the database.

**Syntax**:

```
protocol:<subprotocol>:<subname>
```

- The following Code Snippet shows an example of the URL:

**Code Snippet**:

```
jdbc:odbc:demo
```

- `DriverManager.getConnection()` method is used to create the connection along with appropriate credentials.

**Syntax**:

```
Connection cn = DriverManager.getConnection(<connection url>,
<username>, <password>);
```

- The following Code Snippet shows the creation of a `Connection` object:

**Code Snippet**:

```
Connection cn = DriverManager.getConnection("jdbc:odbc:demo", "sa",
"playware");
```

# Creating Statements and Queries

- The SQL statements are created through Statement interface. A statement is prepared through `createStatement()` method.

**Syntax**:

```
public Statement createStatement() throws SQLException
```

- The following Code Snippet demonstrates how to create a Connection object and create appropriate SQL statement:

**Code Snippet**:

```
Connection cn = DriverManager.getConnection ("jdbc:odbc:demo",
"sa","playware");
Statement st = cn.createStatement();
```

# Using executeQuery() and ResultSet Objects

## executeQuery() method

- The statement is executed over the database through `executeQuery()` method, where the SQL query is passed as a String parameter to the method.

## ResultSet object

- `ResultSet` objects are used to receive and store the data in the same form as it is returned from the SQL queries.

- `ResultSet` object is created to store the result of SQL statement.
- The following Code Snippet shows how the result of SQL query can be stored in a `ResultSet` object:

**Code Snippet:**

```
ResultSet rs = st.executeQuery("SELECT * FROM Department");
```

where,
`sql` is a `String` object containing SQL statement to be sent to the database.

# Using executeUpdate() and execute () Methods

**executeUpdate()**

Used to execute `INSERT`, `DELETE`, `UPDATE`, and other SQL DDL (Data Definition Language) such as `CREATE TABLE`, `DROP TABLE`.

**Syntax:**

```
public int executeUpdate(String sql) throws SQLException
```

where,

`sql` is a String object created to store the result of SQL statement.

`int` is an integer value storing the number of affected rows.

**execute()**

The `execute()` method is used to execute SQL statements that returns more than one result set.

**Syntax:**

```
public boolean execute (String sql) throws SQLException
```

where,

`sql` is String object created to store the result of SQL statement.

# Handling Exceptions in JDBC 1-2

## ClassNotFoundException

- While loading a driver using `Class.forName()`, if the class for the specified driver is not present in the package, then the method invocation throws a `ClassNotFoundException`.
- Following Code Snippet shows use of the `ClassNotFoundException`:

**Code Snippet**:

```
try {

  Class.forName("SpecifiedDriverClassName");
  } catch(java.lang.ClassNotFoundException e) {

  System.err.print("ClassNotFoundException: ");
  System.err.println(e.getMessage());
}
```

# Handling Exceptions in JDBC 2-2

## SQLException

- Every method defined by the JDBC objects such as `Connection`, `Statement`, and `ResultSet` can throw `java.sql.SQLException`.
- Following Code Snippet shows use of `SQLException`:

**Code Snippet**:

```
try {
// Code that could generate an exception goes here.
// If an exception is generated, the catch block below
// will print out information about it.
}
catch(SQLException ex){
System.err.println("SQLException: " + ex.getMessage());
System.out.println("ErrorCode: " + ex.getErrorCode ());
}
```

# Connecting a JDBC Application with SQL Server Database

- Appropriate drivers should be downloaded and installed to connect the database.

**Syntax:**

```
jdbc:sqlserver://serverName;instanceName:portNumber;property=
value[;property=value]
```

- Following Code Snippet shows how to connect to SQL Server database:

**Code Snippet:**

```
jdbc:sqlserver://localhost;user=Author;password=******;
```

# Creating Parameterized Queries

- Parameterized query accepts input from the user at runtime.
- The parameter expected at the runtime is represented by a '?' symbol in the SQL query.
- Can be created with the help of `PreparedStatement` object.
- Reduces execution time.
- Created as `PreparedStatement` object.
- '?' used as parameter placeholder.
- Following Code Snippet shows how to create parameterized query:

**Code Snippet**:

```
String sqlStmt = "UPDATE Employees SET Dept_ID = ? WHERE
Dept_Name LIKE ?";
PreparedStatement pStmt = cn.preparedStatement(sqlStmt);
```

# Passing Parameters to Parameterized Queries

- To substitute the '?' placeholder with a supplied value, one of the `setXXX()` method of the required primitive type is used.
- Following Code Snippet shows how to pass parameters:

**Code Snippet**:

```
pStmt.setInt(1, 25);
pStmt.setString(2, "Production");
```

- The first line of code sets the first parameter of the query with an int value **25**.
- In the second line of code the second parameter of the query is substituted by a string value '**Production**'.
- Following Code Snippet 10 demonstrates the execution of a parameterized query using the `executeUpdate()` method:

**Code Snippet**:

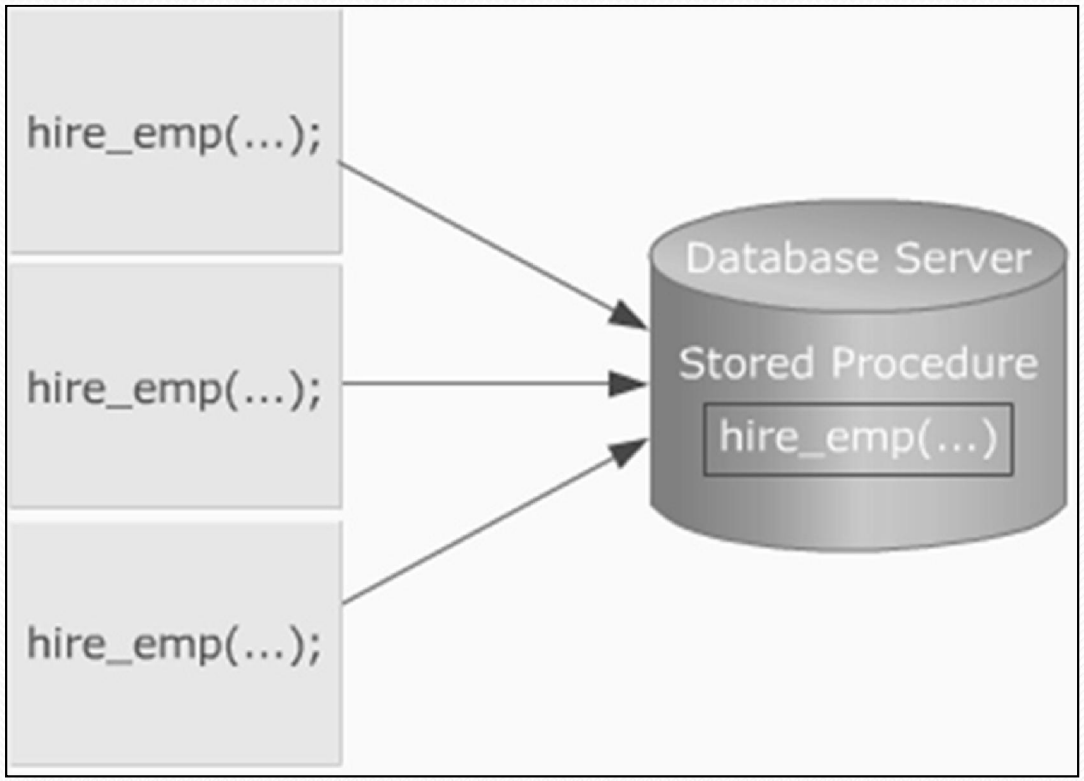
```
pStmt.executeUpdate();
```

# Stored Procedures 1-2

- Stored procedures are a set of SQL statements executed together to accomplish a task.
- Stored procedures are generally created through '`Create Procedure`' statement.
- Stored procedures can be created from JDBC application through `PreparedStatement` objects.
- Stored procedures are written in PL/SQL which is SQL along with other programming structures like loops and conditional statements.
- Stored procedures reduce database network traffic, improve performance, and help ensure the integrity of the database.
- As stored procedures are pre-compiled, they are faster and more efficient than using individual SQL query statements.

# Stored Procedures 2-2

Following figure illustrates how a stored procedure acts on the database:



# Creating a Stored Procedure Using a Statement Object 1-2

- **Creating a stored procedure and storing it in a String variable**

```
String createProcedure = "Create Procedure
DISPLAY_PRODUCTS"+ "as " + "select PRODUCTS.PRD_NAME,
COFFEES.COF_NAME " + "from PRODUCTS, COFFEES " + "where
PRODUCTS.PRD_ID = COFFEES.PRD_ID " + "order by PRD_NAME";
```

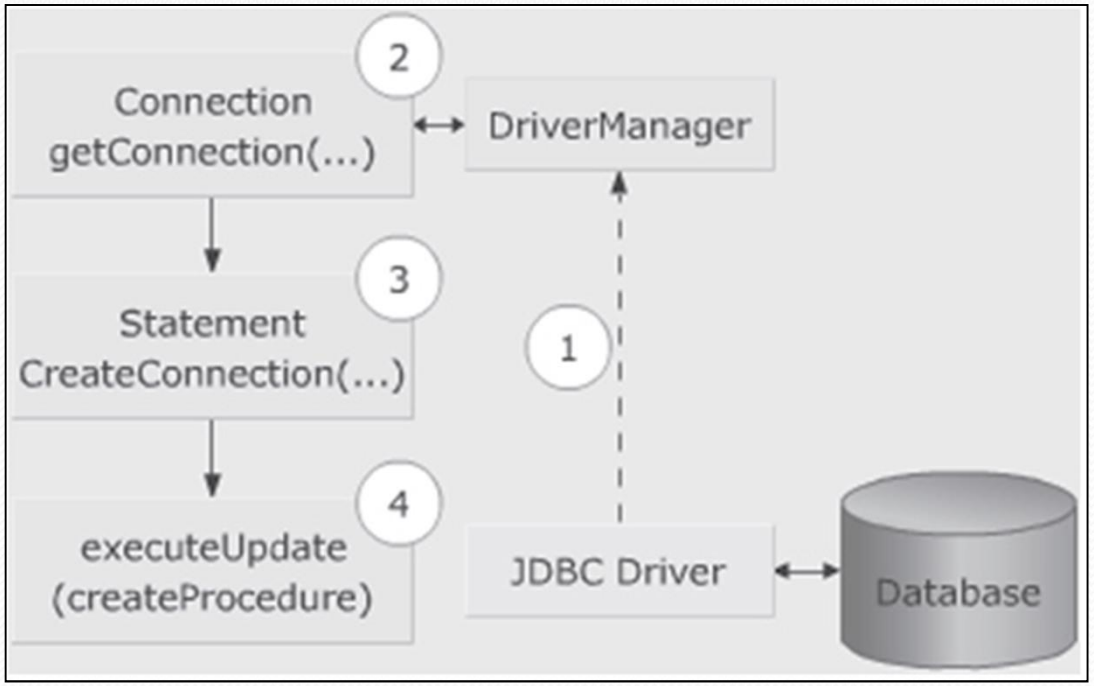
- The following Code Snippet shows creating a procedure using a `Statement` object:

**Code Snippet**:

```
// An active connection cn is used to create a Statement object
        Statement st = cn.createStatement();
// Execute the stored procedure
        st.executeUpdate(createProcedure);
```

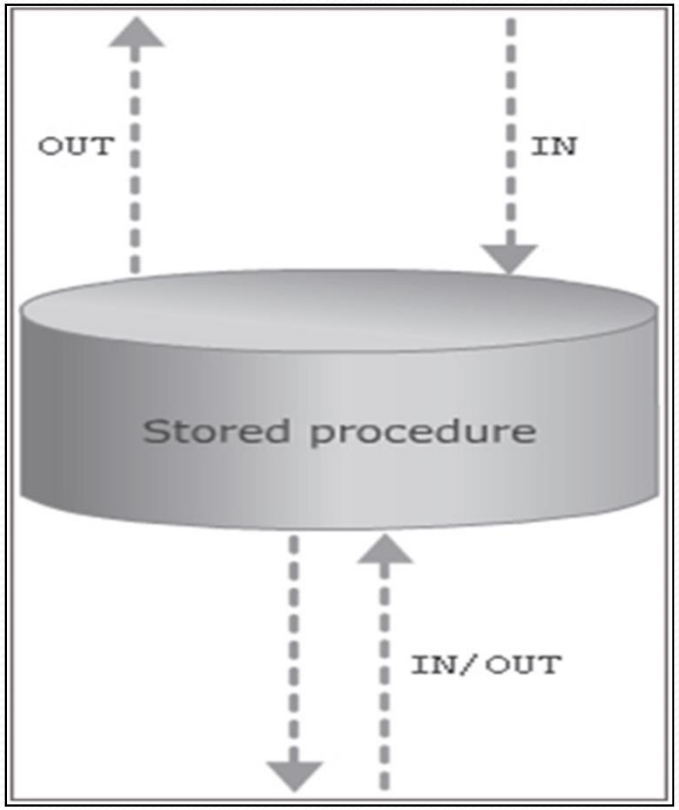
# Creating a Stored Procedure Using a Statement Object 2-2

The following figure graphically shows how various objects interact while creating a statement object:



# Parameters of a Stored Procedure

- `IN`: Used to pass input to the procedure.
- `OUT`: Used to read the output from the procedure.
- `IN/OUT`: Used for both input and output of the procedure.
- Following figure displays the parameters in stored procedure:



# Creating a Callable Statement

- Once the `executeUpdate()` creates the procedure in the database the procedure can be called by creating a '`CallableStatement`' object.

**Syntax**:

```
CallableStatement cst= cn.prepareCall("{call functionname(?, ?)}");
```

**Procedure call without parameters**

- `{call procedure_name}`

**Procedure call with input parameters**

- `{call procedure_name[(?, ?, ...)]}`

**Procedure call with both input and output parameters**

- `{? = call procedure_name[(?, ?, ...)]}`

**'?' acts as place holder for IN or OUT parameters**

# Using the registerOutParameter() Method 1-7

- The `registerOutParameter()` method is used to register the JDBC type. After the registration is done, the statement has to be executed.
- The `get<Type>()` methods of `CallableStatement` are used to retrieve the OUT parameter value.
- The `registerOutParameter()` method uses a JDBC type (so that it matches the JDBC type that the database will return), and `get<Type>()` casts this to a Java type.

# Using the registerOutParameter() Method 2-7

- The following Code Snippet demonstrates the use of the `registerOutParameter()` method:

**Code Snippet**:

```
...
CallableStatement cs = cn.prepareCall("{call getData(?, ?)}");
cs.registerOutParameter(1, java.sql.Types.INTEGER);
cs.registerOutParameter(2, java.sql.Types.DECIMAL, 3);
cs.executeQuery();
int x = cs.getInt(1);
java.math.BigDecimal n = cs.getBigDecimal(2, 3);
...
```

- The following Code Snippet demonstrates how to retrieve an OUT parameter returned by a stored procedure:

**Code Snippet**:

```
create procedure sal @a int OUT
as Select (Salary) from Employee;
```

# Using the registerOutParameter() Method 3-7

- The procedure copies the maximum salary from the Employee table into this parameter.
- To actually create the procedure, the following command is to be typed:

```
@ <path of the sal.sql file>
```

- This procedure will be invoked through the code as shown in the following Code Snippet:

**Code Snippet**:

```
import java.sql.*;
import java.util.*;
class CallOutProc {
Connection con;
String url;
String serverName;
String instanceName;
String databaseName;
String userName;
String password;
String sql;
```

# Using the registerOutParameter() Method 4-7

```
CallOutProc() {
url = "jdbc:sqlserver://";
serverName = "10.2.1.51";
instanceName = "martin";
databaseName = "DeveloperApps";
userName = "sa";
password = "playware";
}
private String getConnectionUrl() {
//constructing the connection string
return url + serverName + ";instanceName = " +instanceName
+"
;DatabaseName = " +databaseName;
}
private java.sql.Connection.getConnection(){
try {
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver
");
//establishing the connection
con =
DriverManager.getConnection(getConnectionUrl(),userName,
password);
```

# Using the registerOutParameter() Method 5-7

```
if(con != null)
System.out.println("Connection Successful!");
}
catch(Exception e) {
e.printStackTrace();
System.out.println("Error Trace in getConnection():
"+e.getMessage());
}
return con;
}
public void display(){
try {
con = getConnection();
CallableStatement cstmt = con.prepareCall("{call sal(?)}");
cstmt.setInt(1,2500);
cstmt.registerOutParameter(1, java.sql.Types.INTEGER);
ResultSet rs = cstmt.executeQuery();
rs.next();
int x = rs.getInt(1);
System.out.println(x);
}
```

# Using the registerOutParameter() Method 6-7

```
catch(SQLException ce) {
System.out.println(ce);
}
}
public static void main(String args[]) {
CallOutProc proObj = new CallOutProc();
proObj.display();
}
}
```

- A procedure for recalculating the salary of the highest salary earner is shown in the following Code Snippet. The procedure will accept a value and will store the value in an OUT parameter.

**Code Snippet**:

```
create procedure recalculatetotal
@a int, @inc_a int OUT
as
select @inc_a = max(salary) from Employee
set @inc_a = @a * @inc_a;
```

# Using the registerOutParameter() Method 7-7

- Following Code Snippet shows the use of IN and OUT parameter with SQL:

**Code Snippet**:

```
. . .
Connection cn = getConnection();
CallableStatement cstmt = cn.prepareCall("{call
recalculatetotal(?,?)}");
cstmt.setInt(1, 10);
cstmt.registerOutParameter(2, java.sql.Types.INTEGER);
cstmt.execute();
int x = cstmt.getInt(2);
System.out.println("New Salary after increment = " + x);
. . .
```

# Executing a Procedure Using Callable Statement 1-2

- **Executing using a `CallableStatement` object**

  `{call DISPLAY_PRODUCTS}`

- The driver translates the escape syntax into the native SQL used by the database to call the stored procedure named `DISPLAY_PRODUCTS`.
- **`cst.executeQuery()`**

  The `executeQuery()` method is used to execute the `CallableStatement` object **`cst`**.
- The following Code Snippet creates a `CallableStatement` object using a `Connection` object.

**Code Snippet**:

```
String createProcedure = "Create Procedure DISPLAY_PRODUCTS
" + "as " + "select PRODUCTS.PRD_NAME, COFFEES.COF_NAME " +
"from PRODUCTS, COFFEES " + "where PRODUCTS.PRD_ID =
COFFEES.PRD_ID " + "order by PRD_NAME";
CallableStatement cst = cn.prepareCall("{call
DISPLAY_PRODUCTS}");
ResultSet rs = cst.executeQuery();
```

# Executing a Procedure Using Callable Statement 2-2

- On successful completion of `executeQuery()` method, a procedure named **`DISPLAY_PRODUCTS`** is created that consists of series of statements which are stored in the `String` variable named **`createProcedure`**.
- Following figure displays the output:

| PRD_NAME | COF_NAME |
|---|---|
| Lakme, Inc. | Russian |
| Lakme, Inc. | Russian_Decaf |
| Superior Coffee | French_Roast |
| Superior Coffee | French_Roast_Decaf |
| The Mountain | Espresso |

# Stored Procedure Using PreparedStatement Interface 1-4

- The execution of `PreparedStatement` objects is faster than that of Statement objects as they are precompiled.
- `PreparedStatement` objects are also useful for specifying many arguments for a particular SQL command.
- The code in the given statement demonstrates how to create a `PreparedStatement` object.

```
PreparedStatement pst = cn.prepareStatement("UPDATE
course SET hours = ? WHERE CourseTitle = ?");
```

- The value of each `?` parameter must be set using the `set<Type>()` method as shown in the following Code Snippet:

**Code Snippet**:

```
pst.setInt(1,800);
pst.setString(2,"ACCP");
```

- Now, the statement can be executed as follows:

```
pst.executeUpdate();
```

# Stored Procedure Using PreparedStatement Interface 2-4

- Following table shows the structure of the **Course** table:

| Column Name | Data type |
|---|---|
| Id | integer |
| Course_title | varchar(20) |
| Hours | integer |

- Following Code Snippet demonstrates the usage of a `PreparedStatement` by connecting to a table named **Course**:

**Code Snippet**:

```
...
public void display() {
try {
PreparedStatement pst = cn.prepareStatement("UPDATE Course SET
Hours=? WHERE Course_Title like ?");
pst.setInt(1, 800);
pst.setString(2, "Windows 2003");
pst.executeUpdate();
System.out.println("Record updated!");
Statement st = cn.createStatement();
String sql = "SELECT * FROM Course";
ResultSet rs = st.executeQuery(sql);
```

# Stored Procedure Using PreparedStatement Interface 3-4

```
while(rs.next()) {
System.out.println(" ");
System.out.print(rs.getInt(1) + " ");
System.out.print(rs.getString(2) + " ");
System.out.println(rs.getInt(3));
}
}catch(SQLException ce) {
System.out.println(ce);
}
}
...
```

- Following Code Snippet shows how to use `PreparedStatement` in an SQL query based upon a condition and the condition is given as an IN parameter:

**Code Snippet**:

```
...
public void display() {
try {
sql = "select * from Friends where Salary >?";
PreparedStatement pst = cn.prepareStatement(sql,
ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

# Stored Procedure Using PreparedStatement Interface 4-4

```
pst.setInt(1, 5000);
ResultSet rs = pst.executeQuery();

while(rs.next()) {
System.out.print(rs.getString(1) + ”\t”);
System.out.print(rs.getString(2) + ”\t”);
System.out.print(rs.getLong(3) + “\t”);
System.out.print(rs.getDate(4) + “\t”);
System.out.print(rs.getDouble(5) + “\t”);
System.out.println(“ “);
}
}catch(SQLException ce) {
System.out.println(ce);
}
}
...
```

# Transactions

- A transaction refers to a set of coherent operations performed on the database.
- In the context of bank database, deposit made into the account is a transaction which will involve the following operations:
  1. Retrieve the current balance in the account
  2. Add the newly deposited amount to the current balance
  3. Update the database with the new balance
- All the operations must be performed to conclude that the transaction is complete.

# Properties of a Transaction

## Atomicity

- Ensure that either all of the tasks of a transaction are performed or none of them are performed

## Consistency

- Ensure that the database is in a legal state even after the transaction

## Isolation

- Ensure that there are no conflicts between concurrent transactions

## Durability

- Ability of DBMS to recover committed transactions even if the system or storage media fails

# Concurrent Transactions

Concurrent access to the database may result in:

**Dirty read**

- A dirty read happens when an uncommitted transaction A has locked a row and transaction B is also reading the same row.

**Phantom read**

- Phantom read refers to a situation when transaction reads additional rows for the same conditional query.

**Non Repeatable read**

- Non repeatable read refers to a situation when transaction A retrieves the same row twice but sees different data.

Locking mechanism is introduced in the database to avoid inconsistency of the databases.

# Isolation Levels

Transaction isolation levels determine the method of acquiring the locks on the database.
Following table shows various isolation levels:

| Isolation Level | Transactions | Dirty Read | Non-Repeatable Reads | Phantom Read |
|---|---|---|---|---|
| TRANSACTION_NONE | Not Supported | NA | NA | NA |
| TRANSACTION_READ_COMMITTED | Supported | Prevented | Allowed | Allowed |
| TRANSACTION_READ_UNCOMMITTED | Supported | Allowed | Allowed | Allowed |
| TRANSACTION_REPEATABLE_READ | Supported | Prevented | Prevented | Allowed |
| TRANSACTION_SERIALIZABLE | Supported | Prevented | Prevented | Prevented |

# Savepoints

- `Savepoint` refers to a logical point in the transaction where a subset of the locks held by the transaction can be released.
- In JDBC, save points are implemented through Savepoint objects as follows:

```
Savepoint save1 = con.setSavepoint();
```

- In order to rollback a transaction to a certain savepoint, the savepoint is passed as a parameter to the rollback method as follows:

```
con.rollback(save1);
```

- Savepoints can be released through releaseSavepoint() method as follows:

```
con.releasaSavepoint(save1);
```

# Implementing Transactions Using JDBC 1-3

- **Step 1 – Start the Transaction**
  - Disable `autocommit` mode through `setAutoCommit()`.
  - Perform transaction- prepared statements are committed only after call to `commit()` method.
  - Use `Savepoint`.
  - Close the transaction
- **Step 2 – Perform the Transaction**
  - The second step is to perform the transaction as shown in the following Code Snippet:

**Code Snippet**:

```
...
PreparedStatement updateSales = cn.prepareStatement("UPDATE
COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Russian");
updateSales.executeUpdate();
```

# Implementing Transactions Using JDBC 2-3

```
PreparedStatement updateTotal = cn.prepareStatement("UPDATE
COFFEES SET
TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Russian");
updateTotal.executeUpdate();
...
```

- The two prepared statements **updateSales** and **updateTotal** will be committed together after the call to `commit()` method.
- **Step 3 – Use Savepoint**
  - The third step is to use Savepoint in the transaction. Consider the following example:

```
// Create an instance of Statement object
Statement st = cn.createStatement();
int rows = st.executeUpdate("INSERT INTO EMPLOYEE
(NAME) VALUES (?COMPANY?)");
```

# Implementing Transactions Using JDBC 3-3

```
// Set the Savepoint
Savepoint svpt = cn.setSavepoint("SAVEPOINT_1");
rows = st.executeUpdate("INSERT INTO EMPLOYEE (NAME)
VALUES (?FACTORY?)");
```

- The code inserts a row into a table, sets the Savepoint **`svpt`**, and then inserts a second row.

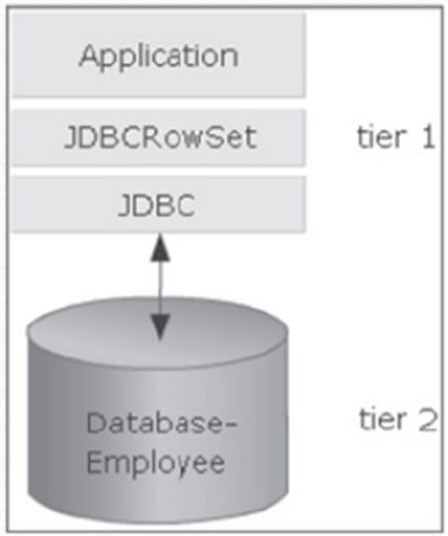
- **Step 4 – Close the Transaction**
  - The last step is to close or end the transaction.
  - A transaction can end either with a commit or with a rollback.
  - Following Code Snippet demonstrates how to close the transaction:

**Code Snippet**:

```
...
// End the transaction
cn.rollback(svpt);
OR
...
cn.commit();
```

# RowSets

- All SQL queries return a set of rows as a result.
- RowSet objects hold the set of rows.
- Following figure displays the `RowSet`:



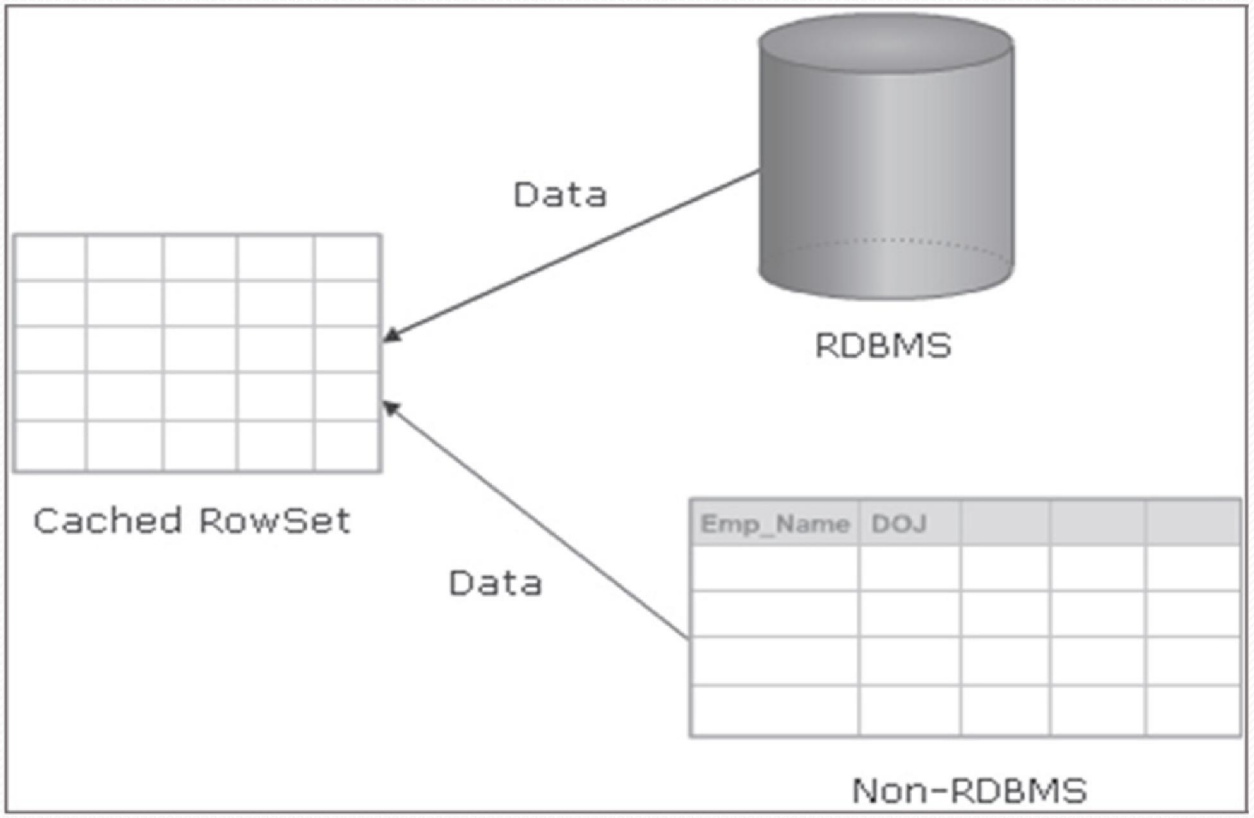

- Following are two types of RowSets:
  - `JdbcRowSet`
  - `CachedRowSet`

# Introduction to JdbcRowSet 1-2

- It is a connected `RowSet` which implies that it requires a dedicated connection to the database to retrieve the set of rows.
- `JdbcRowSet` can perform the following operations on the database:
  - **Insert**: To insert a record into the `JDBCRowSet` object the `moveToInsertRow()` method is invoked.
  - **Delete**: Deleting a row from the `JDBCRowSet` object is simple. The Code Snippet deletes the seventh row from the `RowSet` as well as from the underlying database.
  - **Update**: Updating a record from a `RowSet` object involves navigating to that row, updating data from that row, and finally updating the database.
  - **Retrieve**: A `JDBCRowSet` object is scrollable. Once the cursor is on the desired row, the getter methods can be invoked on the `RowSet` to retrieve the desired values from the columns.

# Introduction to JdbcRowSet 2-2

Following figure illustrates the behavior of `CachedRowSet` objects:



# Introduction to CachedRowSet

- `CachedRowSet` is a disconnected `RowSet` and therefore does not maintain a dedicated connection with the database.
- `CachedRowSet` objects can also perform the following operations:
  - Insert
  - Delete
  - Update
  - Retrieve
- Implementation of these operations is on discontinuous connection with the database server.

# Summary

- JDBC is a software API to access database connectivity for Java applications that provides a collection of application libraries and database drivers.
- The java.sql package offers classes that set up a connection with databases, send the SQL queries to the databases, and retrieve the computed results.
- The classes and interfaces of JDBC API are used to represent objects of database connections, SQL statements, Result sets, Database metadata, Prepared statements, Callable statements, and so on.
- PreparedStatement object is used to execute parameterized runtime queries.
- A stored procedure is a group of SQL statements. Stored procedures are used to group or batch a set of operations or queries to execute on a database server.
- A transaction is a set of one or more statements that are executed together as a unit, so either all of the statements are executed, or none of them are executed.
- A JDBCRowSet is the only implementation of a connected RowSet. Any change made to a JDBCRowSet object is reflected on the database directly.