PrepareCall:--

The PreparedStatement Objects

- The *PreparedStatement* interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object.
- This statement gives you the flexibility of supplying arguments dynamically.

Creating PreparedStatement Object

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   . . .
}
```

- All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.
- The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.
- Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.
- All of the **Statement object's** methods for interacting with the database:
  (a) execute()
  (b) executeQuery(), and
  (c) executeUpdate()
- It also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can input the parameters.

Closing PreparedStatement Object

- Just as you close a Statement object, for the same reason you should also close the PreparedStatement object.
- A simple call to the close() method will do the job. If you close the Connection object first, it will close the PreparedStatement object as well. However, you should always explicitly close the PreparedStatement object to ensure proper cleanup.

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   pstmt.close();
}
```

Resultsetmetadata:

- *ResultSetMetaData* is an interface in *java.sql* package of JDBC API which is used to get the metadata about a *ResultSet* object.
- Whenever you query the database using SELECT statement, the result will be stored in a *ResultSet* object.
- Every *ResultSet* object is associated with one *ResultSetMetaData* object. This object will have all the meta data about a *ResultSet* object like schema name, table name, number of columns, column name, datatype of a column etc. You can get this *ResultSetMetaData* object using *getMetaData()* method of *ResultSet*.
- *getMetaData()* method of *java.sql.ResultSet* interface returns ResultSetMetaData object associated with a ResultSet object.Below is the syntax to get the ResultSetMetaData object:

  ResultSetMetaData rsmd = rs.getMetaData();

  Where '*rs*' is a reference to *ResultSet* object.

JDBC Drives Types: JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

JDBC-ODBC bridge driver

- The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls.
- This is now discouraged because of thin driver.
- Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers of your database instead of the JDBC-ODBC Bridge.

Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into nativ database API. It is not written entirely in java.

Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly database protocol. It is fully written in java.

Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as th fully written in Java language.

Example:

```
package com.pack;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Scanner;
public class DerbyDemo {
public static void main(String[] args) {
       try {
              Scanner sc=new Scanner(System.in);
              System.out.println("enter empid:");
              String empid=sc.nextLine();

             Class.forName("org.apache.derby.jdbc.ClientDriver");
             Connection
conn=DriverManager.getConnection("jdbc:derby://localhost:1527/sample;create=true","user","user");
             Statement st = conn.createStatement();
             System.out.println("Query->"+"select * from app.employee where empid = '"+empid+"'");
             ResultSet rs = st.executeQuery("select * from app.employee where empid='"+empid+"'");

             while (rs.next()) {
              System.out.println(rs.getString(1)+"---"+rs.getString(2));
             }
             conn.close();;
         } catch (Exception e) {
             System.out.println("Exception in database"+e.getMessage());
         }

       }
}
```

Sqlexception:

- In JDBC, we may get exceptions when we execute or create the query. Exceptions that occur due to the Databa come under SQL Exception.

- Using Exception handling, we can handle the SQL Exception like we handle the normal exception.

SQLException is available in the java.sql package.

- It extends the Exception class which means that we can use the methods available in the Exception class in the SQLException class as well.

Example for SQL Exception

Syntax error in the SQL statement may result in SQL Exception. When such an exception occurs, an object of the SQL class will be passed to the catch block. By using the information in the SQLException object, we can catch that except continue the program.

The SQLException object has the following methods:

Method Name	Description
getErrorCode()	It returns the error number
getMessage()	It returns the error message
getSQLState()	It returns the SQLState of the SQLException object. It can return null as well For Database error, it will return XOPEN SQL State
getNextException()	It returns the next exception in the exception chain.
printStackTrace()	It prints the current exception and its backtrace to a standard error stream
setNextException(SQLEXception ex)	It is used to add another SQL exception in the chain

How To Handle Exceptions

JDBC-related exception mostly throws SQLException, and it is a checked exception so we must either catch it or throw business logic and commit data should be done in a Try block, if any exception happened in the block we should catch it in the Catch block. Based on the exception type, we should do the rollbacks or commit in the Catch block.

Execute and executequery:

execute() method: This method is used to execute SQL DDL statements, it returns a boolean value sp
weather the ResultSet object can be retrieved.

executeQuery(): This method is used to execute statements that returns tabular data (example select)
an object of the class ResultSet.

How to perform transactions(Commit and rollback) in java?

COMMIT and ROLLBACK are the two terms used in the transactional statement to perform or undo
transaction.

A COMMIT is the SQL command used **in the transaction tables or database** to make the current tra
database **statement** as permanent. It shows the successful completion of a transaction. If we have s
executed the transaction statement or a simple database query, we want to make the changes permaner
to perform the **commit command** to save the changes, and these changes become permanent fo
Furthermore, once the commit command is executed in the database, we cannot regain its previous stat
it was earlier before the execution of the first statement.

Syntax: COMMIT;

The **SQL ROLLBACK** command is used to roll back the current transaction state if any error occu
the execution of a transaction. In a transaction, the error can be a system failure, power outage, incorrec
of the transaction, system crash, etc. Generally, a rollback command performs the current transactio
action to return the transaction on its **previous state** or the first statement. A rollback command c
executed if the user has not performed the **COMMIT** command on the current transaction or stateme

Syntax: ROLLBACK;

The commit command ensures the transaction changes are permanently saved in the database or tables
a transaction. In contrast, the rollback command is used to undo all changes during the transaction fo
any types of issues such as power failure, wrong data, or return the current transaction to its initial pha

ACID PROPERTIES:

Transaction represents **a single unit of work**.

The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.

Atomicity means either all successful or none.

Consistency ensures bringing the database from one consistent state to another consistent state.

Isolation ensures that transaction is isolated from other transaction.

Durability means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

example of transaction management in jdbc using Statement

Let's see the simple example of transaction management using Statement.

```
import java.sql.*;

1.  class FetchRecords{
2.  public static void main(String args[])throws Exception{
3.  Class.forName("oracle.jdbc.driver.OracleDriver");
4.  Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1
    521:xe","system","oracle");
5.  con.setAutoCommit(false);
6.
7.  Statement stmt=con.createStatement();
8.  stmt.executeUpdate("insert into user420 values(190,'abhi',40000)");
9.  stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");
10.
11. con.commit();
12. con.close();
13. }
14. }
```

Pagination in java:

Pagination is **the process of dividing content into several pages**. The user has a navigation interface for accessing these pages with specific page links. The navigation often includes previous/next and first/last links.

Google search results page is a typical example of such a search.

== and equals,equalsIgnoreCase:

Double equals operator is used to compare two or more than two objects, If they are referring to the same object then return true, otherwise return false. String is immutable in java. When two or more objects are created without new keyword, then both object refer same value. Double equals operator actually compares objects references.

```
public class Sample {
  public static void main(String []args) {
    String s1 = "java";
    String s2 = “java";
    String s3 = new String ("java python");
    System.out.println(s1 == s2);
    System.out.println(s2 == s3);
  }
}
```

Output

```
true
false
```

You can also compare two strings using == operator. But, it compares references of the given variables not values.

The **equals()** method compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

Example

```
public class Sample {
  public static void main(String []args) {
    String s1 = "java";
    String s2 = "java";
    String s3 = new String ("java python");
    System.out.println(s1.equals(s2));
    System.out.println(s2.equals(s3));
  }
}
```

Output

```
true
false
```

Java String **equalsIgnoreCase()** method is much similar to equals() method, except that case is ignored like in above example String object s4 compare to s3 then equals() method return false, but here in case of equalsIgnoreCase() it will return true. Hence equalsIgnoreCase() method is Case Insensitive.

```
public class Demo {
   public static void main(String[] args) {
      String one = "qwerty";
      String two = "Qwerty";
      if(one.equalsIgnoreCase(two)) {
         System.out.println("String one is equal to two (ignoring the case) i.e.
one==two");
      }else{
         System.out.println("String one is not equal to String two (ignoring the
case) i.e. one!=two");
      }
      if(one.equals(two)) {
         System.out.println("String one is equal to two i.e. one==two");
      }else{
```

```
        System.out.println("String one is not equal to String two i.e. one!=two");
      }
    }
}
```

What is hashcode and equals and what is the use of these?

The equals() and hashcode() are the two important methods provided by the **Object** class for comparing objects. Since the Object class is the parent class for all Java objects, hence all objects inherit the default implementation of these two methods. In this topic, we will see the detailed description of equals() and hashcode() methods, how they are related to each other, and how we can implement these two methods in Java

Java equals()

- The java equals() is a method of *lang.Object* class, and it is used to compare two objects.
- To compare two objects that whether they are the same, it compares the values of both the object's attributes.
- By default, two objects will be the same only if stored in the same memory location.

Syntax: **public boolean** equals(Object obj)

Java hashcode()

- A **hashcode** is an integer value associated with every object in Java, facilitating the hashing in hash tables.
- To get this hashcode value for an object, we can use the hashcode() method in Java. It is the means ***hashcode() method that returns the integer hashcode value of the given object***.
- Since this method is defined in the Object class, hence it is inherited by user-defined classes also.

- The hashcode() method returns the same hash value when called on two objects, which are equal according to the equals() method. And if the objects are unequal, it usually returns different hash values.

Syntax: **public int** hashCode()