**What is prepareCall?**

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.

### How to get the instance of CallableStatement?

The prepareCall() method of Connection interface returns the instance of CallableStatement. Syntax is given below:

**public** CallableStatement prepareCall("{ call procedurename(?,?...?)}");

The example to get the instance of CallableStatement is given below:

CallableStatement stmt=con.prepareCall("{call myprocedure(?,?)}");

# Java ResultSetMetaData ?

The metadata means data about data i.e. we can get further information from the data.

If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

## Commonly used methods of ResultSetMetaData interface?

|  |  |
| --- | --- |
| **Method** | **Description** |
| public int getColumnCount()throws SQLException | it returns the total number of columns in the ResultSet object. |
| public String getColumnName(int index)throws SQLException | it returns the column name of the specified column index. |
| Public String getColumnTypeName(int index)throws SQLException | it returns the column type name for the specified index. |
| public String getTableName(int index)throws SQLException | it returns the table name for the specified column index. |

**import** java.sql.\*;

**class** Rsmd{

**public** **static** **void** main(String args[]){

**try**{

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

Connection con=DriverManager.getConnection(

"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

PreparedStatement ps=con.prepareStatement("select \* from emp");

ResultSet rs=ps.executeQuery();

ResultSetMetaData rsmd=rs.getMetaData();

System.out.println("Total columns: "+rsmd.getColumnCount());

System.out.println("Column Name of 1st column: "+rsmd.getColumnName(1));

System.out.println("Column Type Name of 1st column: "+rsmd.getColumnTypeName(1));

con.close();

}**catch**(Exception e){ System.out.println(e);}

}

}

# What Are the Types of JDBC Drivers?

Today, there are five types of JDBC drivers in use:

* ****Type 1:**** JDBC-ODBC bridge
* ****Type 2:**** partial Java driver
* ****Type 3:**** pure Java driver for database middleware
* ****Type 4:**** pure Java driver for direct-to-database
* ****Type 5:**** highly-functional drivers with superior performance

****Type 1 JDBC drivers**** are used for testing JDBC applications against an ODBC data source.

****Type 2 JDBC drivers**** require a native database API to be used. Both Type 1 and Type 2 JDBC driver types mix a Java-based API with another API.

****Type 3 is a single JDBC driver**** used to access a middleware server, which, in turn, makes the relevant calls to the database. A good example of Type 3 JDBC driver is the [DataDirect SequeLink JDBC](https://www.progress.com/sequelink) driver

****Type 4 drivers**** are the most common and are designed for a particular vendor's database.

****Type 5 JDBC drivers**** (such as [DataDirect JDBC drivers](https://www.progress.com/jdbc)) offer advanced functionality and superior performance over other driver types.

### **SQLException?**

In JDBC, we may get exceptions when we execute or create the query. Exceptions that occur due to the Database or Driver 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 SQLException class will be passed to the catch block. By using the information in the SQLException object, we can catch that exception and 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 |

**EXCUTE AND EXCUTE QUERY AND UPDATE?**

execute : Returns true if the first object that the query returns is a ResultSet object. ...

executeQuery : Returns one ResultSet object.

executeUpdate : Returns an integer representing the number of rows affected by the SQL statement.

**How do I rollback a JDBC transaction?**

**Execute the transactions for rollback**

1. Import the database.
2. Load and register drivers if necessary.
3. Create a new connection.
4. Create a statement for commit/rollback.
5. Execute the query for commit/rollback.
6. Process the results.
7. Close the connection else previous processing may lose if any.

## Commit & Rollback

Once you are done with your changes and you want to commit the changes then call **commit()** method on connection object as follows −

conn.commit( );

Otherwise, to roll back updates to the database made using the Connection named conn, use the following code −

conn.rollback( );

The following example illustrates the use of a commit and rollback object −

try{

//Assume a valid connection object conn

conn.setAutoCommit(false);

Statement stmt = conn.createStatement();

String SQL = "INSERT INTO Employees " +

"VALUES (106, 20, 'Rita', 'Tez')";

stmt.executeUpdate(SQL);

//Submit a malformed SQL statement that breaks

String SQL = "INSERTED IN Employees " +

"VALUES (107, 22, 'Sita', 'Singh')";

stmt.executeUpdate(SQL);

// If there is no error.

conn.commit();}catch(SQLException se){

// If there is any error.

conn.rollback();

}

**ACID PROPERTIES:**

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

**Atomicity** means either all successful or none.

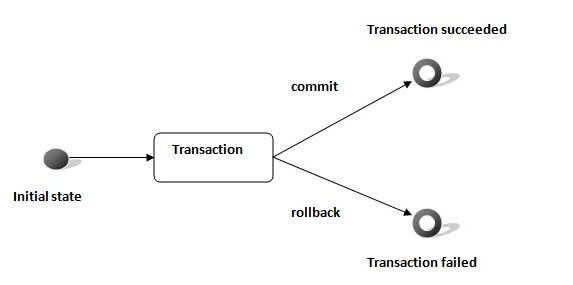
**Consistency** ensures bringing the database from one consistent state to another consistent state.

**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.

#### Advantage of Transaction Mangaement

**fast performance** It makes the performance fast because database is hit at the time of commit.



**JDBC pagination:**

I have a table with millions of records in it. In order to make the system faster, I need to implement the pagination concept in my Java code. I need to fetch just 1000 records at a time and process them, then pick another 1000 records and do my processing and so on. I have already tried a few things and none of them is working. Some of the things I tried are listed below -

1) String query = "select \* from TABLENAME" + " WHERE ROWNUM BETWEEN %d AND %d";

sql = String.format(query, firstrow, firstrow + rowcount);

In the above example, when the query is SELECT \* from TABLENAME Where ROWNUM BETWEEN 0 and 10 it gives me a result but when the query is SELECT \* from TABLENAME Where ROWNUM BETWEEN 10 and 20, it returns an empty result set. I even tried to run it in the DB, it return Empty result set (not sure why!!)

2) preparedStatement.setFetchSize(100); I have that in my Java code, but it still fetches all the records from the table. Adding this statement didnt affect my code in anyway.

# Equals() and Hashcode() in Java

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](https://www.javatpoint.com/java-tutorial)

.

## 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:

1. **public** **boolean** equals(Object obj)

### Parameter:

****obj****: It takes the reference object as the parameter, with which we need to make the comparison.

### Returns:

It returns the true if both the objects are the same, else returns false.

### General Contract of equals() method

There are some general principles defined by [Java SE](https://www.javatpoint.com/java-se)

that must be followed while implementing the equals() method in Java. The equals() method must be:

* **reflexive**: An object x must be equal to itself, which means, for object x, ****equals(x)**** should return true.
* **symmetric**: for two given objects x and y, x.equals(y) must return true if and only if equals(x) returns true.
* **transitive**: for any objects x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
* **consistent**: for any objects x and y, the value of x.equals(y) should change, only if the property in equals() changes.
* For any object x, the equals(null) must return false.

## 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:

1. **public** **int** hashCode()

### Returns:

It returns the hash code value for the given objects.

### Contract for hashcode() method in Java

* If two objects are the same as per the equals(Object) method, then if we call the hashCode() method on each of the two objects, it must provide the same integer result.

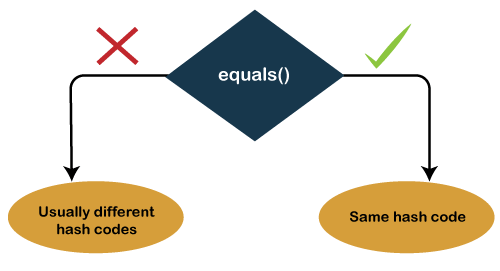
1. **public** **int** hashCode()

### Returns:

It returns the hash code value for the given objects.

### Contract for hashcode() method in Java

* If two objects are the same as per the equals(Object) method, then if we call the hashCode() method on each of the two objects, it must provide the same integer result.



**== and equals() equalignorecase?**

equals() in Java to check for equality between two strings.

Use equalsIgnoreCase() in Java to check for equality between two strings ignoring the case.

Let’s say the following are our two strings −

String one = "qwerty";

String two = "Qwerty";

Both are equal, but the case is different. Since the method ignores case, both of these strings would be considered equal using equalsIgnoreCase() method.

Here, we are checking the same −

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");}

However, under equals() case, they won’t be considered equal −

We can use == operators for reference comparison (address comparison) and . equals() method for **content comparison**. In simple words, == checks if both objects point to the same memory location whereas . equals() evaluates to the comparison of values in the objects.