18

Building Database Applications with JDBC

ORACLE

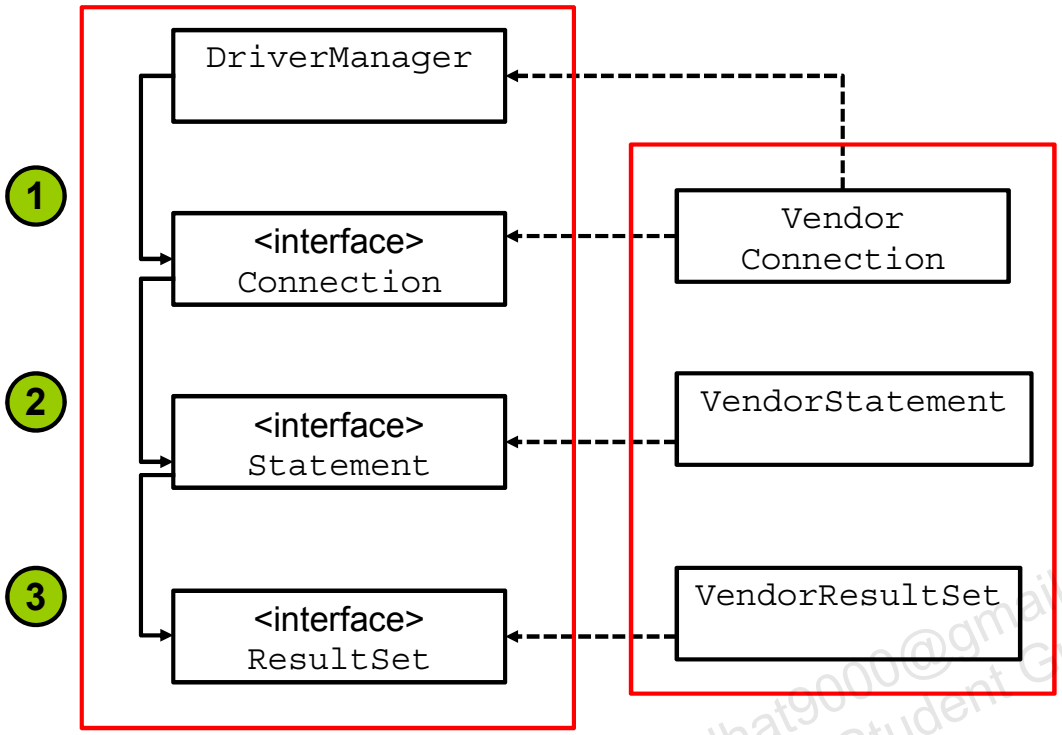
Objectives

After completing this lesson, you should be able to:

- Define the layout of the JDBC API
- Connect to a database by using a JDBC driver
- Submit queries and get results from the database
- Specify JDBC driver information externally
- Perform CRUD operations by using the JDBC API

ORACLE

Using the JDBC API



The JDBC API is made up of some concrete classes, such as `Date`, `Time`, and `SQLException`, and a set of interfaces that are implemented in a driver class that is provided by the database vendor.

Because the implementation is a valid instance of the interface method signature, after the database vendor's Driver classes are loaded, you can access them by following the sequence shown in the slide:

1. Use the `DriverManager` class to obtain a reference to a `Connection` object by using the `getConnection` method. The typical signature of this method is `getConnection (url, name, password)`, where `url` is the JDBC URL, and `name` and `password` are strings that the database accepts for a connection.
2. Use the `Connection` object (implemented by some class that the vendor provided) to obtain a reference to a `Statement` object through the `createStatement` method. The typical signature for this method is `createStatement ()` with no arguments.
3. Use the `Statement` object to obtain an instance of a `ResultSet` through an `executeQuery (query)` method. This method typically accepts a string (`query`), where `query` is a static string.

Using a Vendor's Driver Class

The `DriverManager` class is used to get an instance of a `Connection` object by using the JDBC driver named in the JDBC URL:

```
String url = "jdbc:derby://localhost:1527/EmployeeDB";
Connection con = DriverManager.getConnection (url);
```

- The URL syntax for a JDBC driver is:

```
jdbc:<driver>:[subsubprotocol:][databaseName][;attribute=value]
```

- Each vendor can implement its own subprotocol.
- The URL syntax for an Oracle Thin driver is:

```
jdbc:oracle:thin:@//[HOST][:PORT]/SERVICE
```

Example:

```
jdbc:oracle:thin:@//myhost:1521/orcl
```


DriverManager

Any JDBC 4.0 drivers that are found in the class path are automatically loaded. The `DriverManager.getConnection` method will attempt to load the driver class by looking at the `META_INF/services/java.sql.Driver` file. This file contains the name of the JDBC driver's implementation of `java.sql.Driver`. For example, the `META-INF/services/java.sql.driver` file in `derbyclient.jar` contains `org.apache.derby.jdbc.ClientDriver`.

Drivers prior to JDBC 4.0 must be loaded manually by using:

```
try {
    java.lang.Class.forName("<fully qualified path of the driver>");
} catch (ClassNotfoundException c) {
}
```

Driver classes can also be passed to the interpreter on the command line:

```
java -djdbc.drivers=<fully qualified path to the driver> <class to run>
```

Key JDBC API Components

Each vendor's JDBC driver class also implements the key API classes that you will use to connect to the database, execute queries, and manipulate data:

- `java.sql.Connection`: A connection that represents the session between your Java application and the database

```
Connection con = DriverManager.getConnection(url,
   username, password);
```

- `java.sql.Statement`: An object used to execute a static SQL statement and return the result

```
Statement stmt = con.createStatement();
```

- `java.sql.ResultSet`: An object representing a database result set

```
String query = "SELECT * FROM Employee";
ResultSet rs = stmt.executeQuery(query);
```

Connections, Statements, and ResultSets

The main advantage of the JDBC API is that it provides a flexible and portable way to communicate with a database.

The JDBC driver that is provided by a database vendor implements each of the following Java interfaces. Your Java code can use the interface knowing that the database vendor provided the implementation of each of the methods in the interface:

- **`Connection`:** Is an interface that provides a session with the database. While the connection object is open, you can access the database, create statements, get results, and manipulate the database. When you close a connection, the access to the database is terminated and the open connection closed.
- **`Statement`:** Is an interface that provides a class for executing SQL statements and returning the results. The `Statement` interface is for static SQL queries. There are two other subinterfaces: `PreparedStatement`, which extends `Statement` and `CallableStatement`, which extends `PreparedStatement`.
- **`ResultSet`:** Is an interface that manages the resulting data returned from a `Statement`

Note: SQL commands and keywords are not case-sensitive—that is, you can use `SELECT` or `Select`. SQL table and column names (identifiers) can be case-sensitive or not case-sensitive, depending upon the database. SQL identifiers are not case-sensitive in the Derby database (unless delimited).

Writing Queries and Getting Results

To execute SQL queries with JDBC, you must create a SQL query wrapper object, an instance of the `Statement` object.

```
Statement stmt = con.createStatement();
```

- Use the Statement instance to execute a SQL query:

```
ResultSet rs = stmt.executeQuery (query);
```

- Note that there are three Statement execute methods:

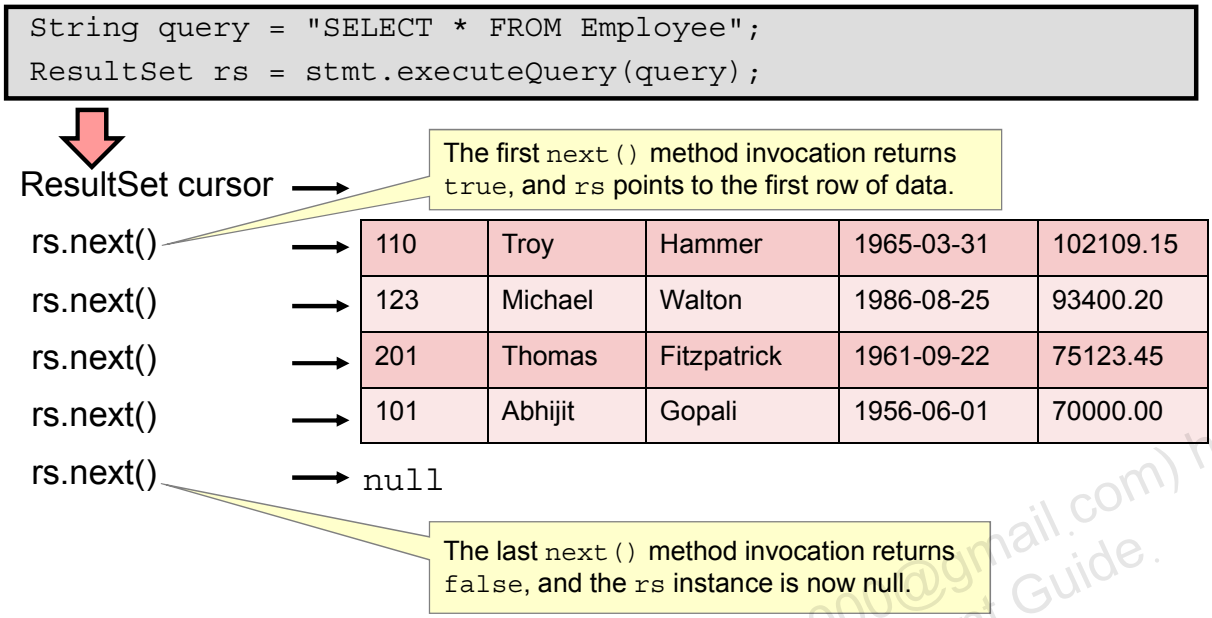
Method	Returns	Used for
`executeQuery(sqlString)`	`ResultSet`	`SELECT` statement
`executeUpdate(sqlString)`	`int` (rows affected)	`INSERT`, `UPDATE`, `DELETE`, or a DDL
`execute(sqlString)`	`boolean` (true if there was a `ResultSet`)	Any SQL command or commands

A SQL statement is executed against a database using an instance of a `Statement` object. The `Statement` object is a wrapper object for a query. A `Statement` object is obtained through a `Connection` object—the database connection. So it makes sense that from a `Connection`, you get an object that you can use to write statements to the database.

The Statement interface provides three methods for creating SQL queries and returning a result. Which one you use depends upon the type of SQL statement you want to use:

- `executeQuery(sqlString)`: For a `SELECT` statement, returns a `ResultSet` object
- `executeUpdate(sqlString)`: For `INSERT`, `UPDATE`, and `DELETE` statements, returns an `int` (number of rows affected), or `0` when the statement is a Data Definition Language (DDL) statement, such as `CREATE TABLE`.
- `execute(sqlString)`: For any SQL statement, returns a `boolean` indicating if a `ResultSet` was returned. Multiple SQL statements can be executed with `execute`.

Using a `ResultSet` Object



`ResultSet` Objects

- `ResultSet` maintains a cursor to the returned rows. The cursor is initially pointing before the first row.
- The `ResultSet.next()` method is called to position the cursor in the next row.
- The default `ResultSet` is not updatable and has a cursor that points only forward.
- It is possible to produce `ResultSet` objects that are scrollable and/or updatable. The following code fragment, in which `con` is a valid `Connection` object, illustrates how to make a result set that is scrollable and insensitive to updates by others, and that is updatable:

```
Statement stmt
       = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                             ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT a, b FROM TABLE2");
```

Note: Not all databases support scrollable result sets.

`ResultSet` has accessor methods to read the contents of each column returned in a row. `ResultSet` has a `getter` method for each type.

CRUD Operations Using JDBC API: Retrieve

```
1 package com.example.text;
2
3 import java.sql.DriverManager;
4 import java.sql.ResultSet;
5 import java.sql.SQLException;
6 import java.util.Date;
7
8 public class SimpleJDBCTest {
9
10    public static void main(String[] args) {
11        String url = "jdbc:derby://localhost:1527/EmployeeDB";
12        String username = "public";
13        String password = "tiger";
14        String query = "SELECT * FROM Employee";
15        try (Connection con =
16             DriverManager.getConnection (url, username, password);
17             Statement stmt = con.createStatement ();
18             ResultSet rs = stmt.executeQuery (query)) {
```

The hard-coded JDBC URL, username, and password are just for this simple example.

ORACLE

CRUD (Create, Retrieve, Update, and Delete) operations are equivalent to the `INSERT`, `SELECT`, `UPDATE`, and `DELETE` statements in SQL.

In the following slide, you see a complete example of a JDBC application, a simple one that reads all the rows from an Employee database and returns the results as strings to the console.

- **Lines 15–16:** Use a `try`-with-resources statement to get an instance of an object that implements the `Connection` interface.
- **Line 17:** Use the connection object to get an instance of an object that implements the `Statement` interface from the `Connection` object.
- **Line 18:** Create a `ResultSet` by executing the string query using the `Statement` object.

Note: Hard coding the JDBC URL, username, and password makes an application less portable. Instead, consider using `java.io.Console` to read the username and password and/or some type of authentication service.

CRUD Operations Using JDBC: Retrieve

Loop through all of the rows in the `ResultSet`.

```
19          while (rs.next()) {
20              int empID = rs.getInt("ID");
21              String first = rs.getString("FirstName");
22              String last = rs.getString("LastName");
23              Date birthDate = rs.getDate("BirthDate");
24              float salary = rs.getFloat("Salary");
25              System.out.println("Employee ID:   " + empID + "\n"
26              + "Employee Name: " + first + " " + last + "\n"
27              + "Birth Date:    " + birthDate + "\n"
28              + "Salary:        " + salary);
29          } // end of while
30      } catch (SQLException e) {
31          System.out.println("SQL Exception: " + e);
32      } // end of try-with-resources
33    }
34 }
```


- **Lines 20–24:** Get the results of each of the data fields in each row read from the `Employee` table.
- **Lines 25–28:** Print the resulting data fields to the system console.
- **Line 30:** `SQLException`: This class extends `Exception` thrown by the `DriverManager`, `Statement`, and `ResultSet` methods.
- **Line 32:** This is the closing brace for the `try`-with-resources statement on line 15.

This example is from the `SimpleJDBCExample` project.

Output:

```
run:
Employee ID:   110
Employee Name: Troy Hammer
Birth Date:    1965-03-31
Salary:        102109.15
```

CRUD Operations Using JDBC API: Create

```
1.  public class InsertJDBCExample {
2.      public static void main(String[] args) {
3.          // Create the "url"
4.          // assume database server is running on the localhost
5.          String url = "jdbc:derby://localhost:1527/EmployeeDB";
6.          String username = "scott";
7.          String password = "tiger";
8.  try (Connection con = DriverManager.getConnection(url, username,
    password))
9.  {
10.   Statement stmt = con.createStatement();
11.   String query = "INSERT INTO Employee VALUES (500, 'Jill',
    'Murray','1950-09-21', 150000)";
12. if (stmt.executeUpdate(query) > 0) {
13.    System.out.println("A new Employee record is added");
14.    }
15. String query1="select *  from Employee";
16. ResultSet rs = stmt.executeUpdate(query1);
17. //code to display the rows
18. }
```

Query to insert a row in the Employee.

This slide demonstrates the insert operation. An employee record is added to the Employee table and the content of the Employee table after the insert operation is displayed in the output console.

Lines 10–13: Create a query to insert an employee record and execute the query.

Lines 15–17: Print the resulting data fields to the system console.

CRUD Operations Using JDBC API: Update

```
1. public class UpdateJDBCExample {
2.    public static void main(String[] args) {
3.        // Create the "url"
4.        // assume database server is running on the localhost
5.        String url = "jdbc:derby://localhost:1527/EmployeeDB";
6.        String username = "scott";
7.        String password = "tiger";
8.    try (Connection con = DriverManager.getConnection(url, username,
password)) {
9.            Statement stmt = con.createStatement();
10.          query = "Update Employee SET salary= 200000 where id=500";
11.           if (stmt.executeUpdate(query) > 0) {
12.  System.out.println("An existing employee record was updated
successfully!");
13.         }
14.           String query1="select * from Employee";
15.           ResultSet rs = stmt.executeQuery(query1);
16.     //code to display the records//
17.}
```


This slide demonstrates the update operation. An existing employee record is updated and the content of the Employee table after the update operation is displayed in the output console.

Lines 9–12: Create a query to update an employee record with ID 500 and execute the query.

Lines 14–16: Print the resulting data fields to the system console.

CRUD Operations Using JDBC API: Delete

```
1.public class DeleteJDBCExample {

2.    public static void main(String[] args) {
3.       String url = "jdbc:derby://localhost:1527/EmployeeDB";
4.        String username = "scott";
5.        String password = "tiger”;
6.   try (Connection con = DriverManager.getConnection(url, username,
password)) {
7.         Statement stmt = con.createStatement();
8.         String query = "DELETE FROM Employee where id=500";
9.   if (stmt.executeUpdate(query) > 0) {
10.   System.out.println("An employee record was deleted successfully");
11.   }
12.       String query1="select * from Employee";
13.       ResultSet rs = stmt.executeQuery(query1);
```


This slide demonstrates the delete operation. An existing employee record is deleted and the content of the Employee table after the delete operation is displayed in the output console.

Lines 7–10: Create a query to delete an employee record with ID 500 and execute the query.

Lines 12–13: Print the resulting data fields to the system console.

SQLException Class

SQLException can be used to report details about resulting database errors. To report all the exceptions thrown, you can iterate through the SQLExceptions thrown:

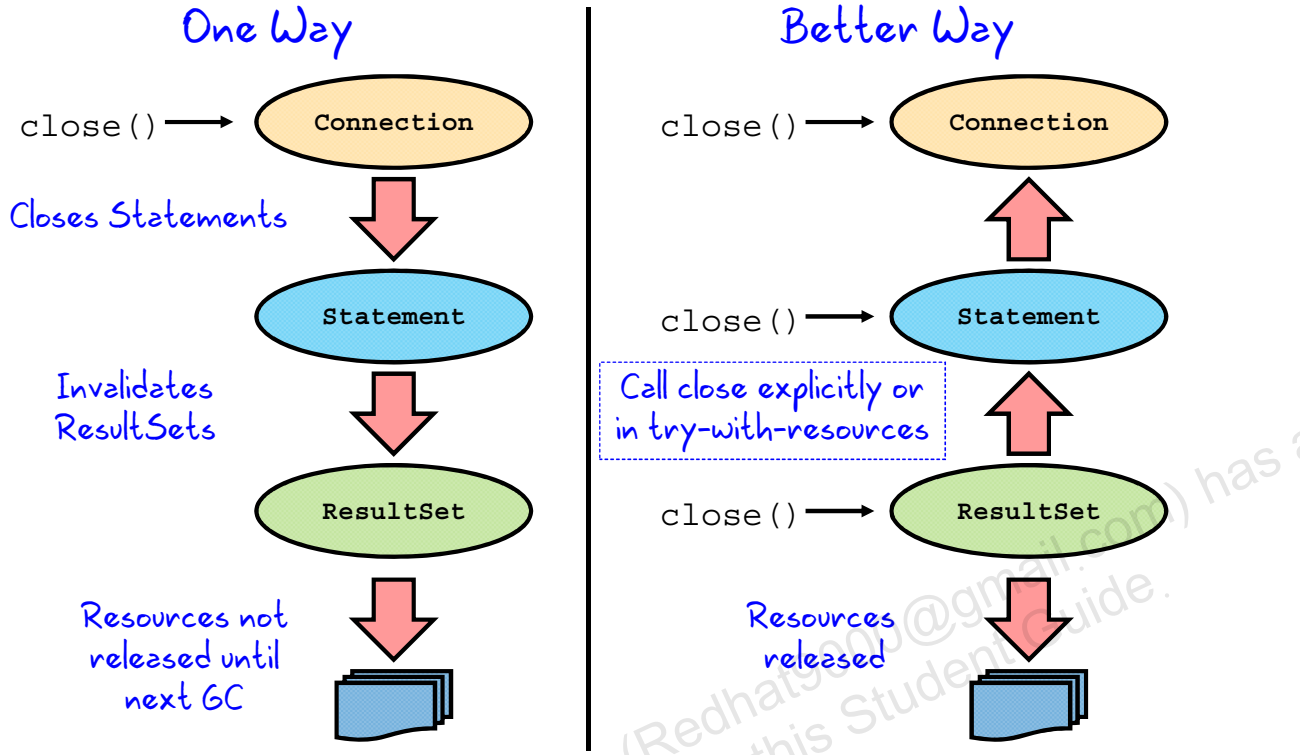
```
1 catch(SQLException ex) {
2     while(ex != null) {
3         System.out.println("SQLState:   " + ex.getSQLState());
4         System.out.println("Error Code:" + ex.getErrorCode());
5         System.out.println("Message:    " + ex.getMessage());
6         Throwable t = ex.getCause();
7         while(t != null) {
8             System.out.println("Cause:" + t);
9             t = t.getCause();
10         }
11         ex = ex.getNextException();
12     }
13 }
```

Vendor-dependent state codes, error codes, and messages

- A SQLException is thrown from errors that occur in one of the following types of actions: driver methods, methods that access the database, or attempts to get a connection to the database.
- The SQLException class also implements Iterable. Exceptions can be chained together and returned as a single object.
- A SQLException is thrown if the database connection cannot be made due to incorrect username or password information or if the database is offline.
- SQLException can also result by attempting to access a column name that is not part of the SQL query.
- SQLException is also subclassed, providing granularity of the actual exception thrown.

Note: SQLState and SQLErrorCode values are database dependent. For Derby, the SQLState values are defined at:
http://download.oracle.com/javadb/10.8.1.2/ref/rrefexcept71493.html

Closing JDBC Objects



- Closing a `Connection` object will automatically close any `Statement` objects created with this `Connection`.
- Closing a `Statement` object will close and invalidate any instances of `ResultSet` created by the `Statement` object.
- Resources held by the `ResultSet` may not be released until garbage is collected. Therefore, it is a good practice to explicitly close `ResultSet` objects when they are no longer needed.
- When the `close()` method on `ResultSet` is executed, external resources are released.
- `ResultSet` objects are also implicitly closed when an associated `Statement` object is re-executed.

In summary, it is a good practice to explicitly close JDBC `Connection`, `Statement`, and `ResultSet` objects when you no longer need them.

Note: A connection with the database can be an expensive operation. It is a good practice to either maintain `Connection` objects for as long as possible, or use a connection pool.

`try`-with-resources Construct

Given the following `try`-with-resources statement:

```
try (Connection con =
     DriverManager.getConnection(url, username, password);
     Statement stmt = con.createStatement();
     ResultSet rs = stmt.executeQuery (query)){
```

- The compiler checks to see that the object inside the parentheses implements `java.lang.AutoCloseable`.
  - This interface includes one method: `void close()`.
- The `close()` method is automatically called at the end of the `try` block in the proper order (last declaration to first).
- Multiple closeable resources can be included in the `try` block, separated by semicolons.

One of the features is the `try`-with-resources statement. This is an enhancement that will automatically close open resources.

With JDBC 4.1, the JDBC API classes including `ResultSet`, `Connection`, and `Statement`, implement `java.lang.AutoCloseable`. The `close()` method of the `ResultSet`, `Statement`, and `Connection` objects will be called in order in this example.

Using `PreparedStatement`

`PreparedStatement` is a subclass of `Statement` that allows you to pass arguments to a precompiled SQL statement.

```
double value = 100_000.00;
String query = "SELECT * FROM Employee WHERE Salary > ?";
PreparedStatement pStmt = con.prepareStatement(query);
pStmt.setDouble(1, value);
ResultSet rs = pStmt.executeQuery();
```

Parameter for substitution.

Substitutes `value` for the first parameter in the prepared statement.

- In this code fragment, a prepared statement returns all columns of all rows whose salary is greater than $100,000.
- `PreparedStatement` is useful when you want to execute a SQL statement multiple times.

The `PreparedStatement` provides two additional benefits:

- Faster execution
- Parameterized SQL Statements

The SQL statement in the example in the slide is precompiled and stored in the `PreparedStatement` object. This statement can be used efficiently to execute this statement multiple times. This example could be in a loop, looking at different values.

Prepared statements can also be used to prevent SQL injection attacks. For example, where a user is allowed to enter a string and that string is executed as a part of a SQL statement, it enables the user to alter the database in unintended ways (such as granting the user permissions).

Note: `PreparedStatement` setXXXX methods index parameters from 1, and not 0. The first parameter in a prepared statement is 1, the second parameter is 2, and so on.

Using `PreparedStatement`: Setting Parameters

In general, there is a **`setXXX`** method for each type in the Java programming language.

`setXXX` arguments:

- The first argument indicates which question mark placeholder is to be set.
- The second argument indicates the replacement value.

For example:

```
pStmt.setInt(1, 175);
pStmt.setString(2,"Charles");
```

Executing `PreparedStatement`

In general, there is a **`setXXX`** method for each type in the Java programming language.

`setXXX` arguments:

- The first argument indicates which question mark placeholder is to be set.
- The second argument indicates the replacement value.

For example:

```
pStmt.setInt(1, 175);
pStmt.setString(2,"Charles");
```

PreparedStatement: Using a Loop to Set Values

```
PreparedStatement updateEmp;
    String updateString = "update Employee"
     + "set SALARY= ? where EMP_NAME like ?";
    updateEmp  = con.prepareStatement(updateString);
    int[] salary = {1750, 1500, 6000, 1550, 9050};
    String[] names = {"David", "Tom", "Nick",
"Harry", "Mark"};
    for(int i:names)
     {
       updateEmp.setInt(1, salary[i]);
       updateEmp.setString(2, names[i]);
       updateEmp.executeUpdate();
    }
```


When using a `PreparedStatement` you can make coding easier by using a `for` loop or a `while` loop to set values for input parameters.

The code snippet in the slide demonstrates using a `for` loop to set the values for input parameters.

A `PreparedStatement` object is created and a `for` loop executes five times. Each time through the loop it sets a new value and executes the SQL statement and updates salaries for five different employees.

Using `CallableStatement`

A `CallableStatement` allows non-SQL statements (such as stored procedures) to be executed against the database.

```
CallableStatement cStmt
       = con.prepareCall("{CALL EmplAgeCount (?, ?)}");
int age = 50;
cStmt.setInt (1, age);
ResultSet rs = cStmt.executeQuery();
cStmt.registerOutParameter(2, Types.INTEGER);
boolean result = cStmt.execute();
int count = cStmt.getInt(2);
System.out.println("There are " + count +
          " Employees over the age of " + age);
```

The `IN` parameter is passed in to the stored procedure.

The `OUT` parameter is returned from the stored procedure.

- Stored procedures are executed on the database.

Stored Procedure

A stored procedure is a group of SQL statements that form a logical unit and perform a particular task. They are used to encapsulate a set of operations or queries to execute on a database server. Stored procedures are supported by most DBMSs, but there is a fair amount of variation in their syntax and capabilities.

Calling a Stored Procedure from JDBC

The first step is to create a `CallableStatement` object. As with Statement and `PreparedStatement` objects, this is done with an open Connection object. A `CallableStatement` object contains a call to a stored procedure; it does not contain the stored procedure itself.

Summary

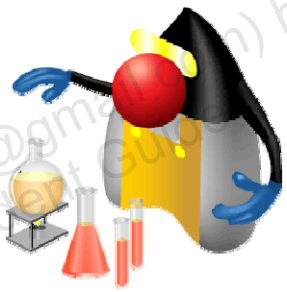
In this lesson, you should have learned how to:

- Define the layout of the JDBC API
- Connect to a database by using a JDBC driver
- Submit queries and get results from the database
- Specify JDBC driver information externally
- Perform CRUD operations by using the JDBC API

Practice 18-1 Overview: Working with the Derby Database and JDBC

This practice covers the following topics:

- Starting the JavaDB (Derby) database from within NetBeans IDE
- Populating the database with data (the Employee table)
- Running SQL queries to look at the data
- Compiling and running the sample JDBC application



In this practice, you will start the database from within NetBeans, populate the database with data, run some SQL queries, and compile and run a simple application that returns the rows of the Employee database table.

Quiz

Which `Statement` method executes a SQL statement and returns the number of rows affected?

a. `stmt.execute(query);`
b. `stmt.executeUpdate(query);`
c. `stmt.executeQuery(query);`
d. `stmt.query(query);`

Quiz

When using a `Statement` to execute a query that returns only one record, it is not necessary to use the `ResultSet`'s `next()` method.

a. True
b. False