**Chapter 5:**

**Introduction to JDBC**

**What is ODBC Driver?**

An ODBC driver uses the Open Database Connectivity (ODBC) interface by Microsoft that allows applications to access data in database management systems (DBMS) using SQL as a standard for accessing the data.

An ODBC driver uses the Open Database Connectivity (ODBC) interface by Microsoft that allows applications to access data in database management systems (DBMS) using SQL as a standard for accessing the data. ODBC permits maximum interoperability, which means a single application can access different DBMS. Application end users can then add ODBC database drivers to link the application to their choice of DBMS.

The ODBC driver interface defines:

* A library of ODBC function calls of two types:
  + Core functions that are based on the X/Open and SQL Access Group
  + Call Level Interface specification
  + Extended functions that support additional functionality, including scrollable cursors
* SQL syntax based on the X/Open and SQL Access Group SQL CAE specification (1992)
* A standard set of error codes
* A standard way to connect and logon to a DBMS
* A standard representation for data types

The ODBC solution for accessing data led to ODBC database drivers, which are dynamic-link libraries on Windows and shared objects on Linux/UNIX. These drivers allow an application to gain access to one or more data sources. ODBC provides a standard interface to allow application developers and vendors of database drivers to exchange data between applications and data sources.

**What is JDBC Driver?**

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.

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
* 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.

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

# JDBC Architecture

The JDBC API supports both two-tier and three-tier processing models for database access.

***Figure 1: Two-tier Architecture for Data Access.***



In the two-tier model, a Java application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

***Figure 2: Three-tier Architecture for Data Access.***



Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeans™, the Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.

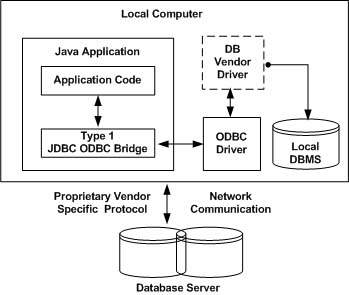
With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets. The JDBC API is also what allows access to a data source from a Java middle tier.

**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.

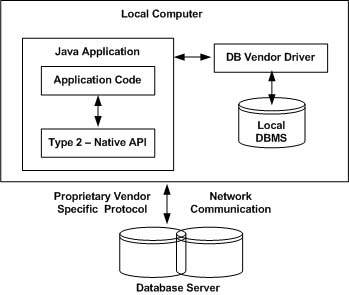
When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.

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.

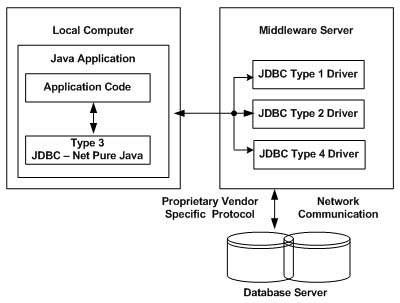


The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

**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.



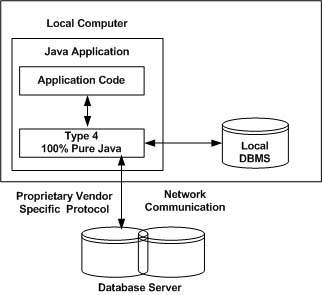
You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.

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.



MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

**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.

**JDBC API**

JDBC API is mainly divided into two package

1. java.sql
2. javax.sql

java.sql package

This package include classes and interface to perform almost all JDBC operation such as creating and executing SQL Queries.

Important classes and interface of java.sql package

|  |  |
| --- | --- |
| classes/interface | Description |
| java.sql.BLOB | Provide support for BLOB(Binary Large Object) SQL type. |
| java.sql.Connection | creates a connection with specific database |
| java.sql.CallableStatement | Execute stored procedures |
| java.sql.CLOB | Provide support for CLOB(Character Large Object) SQL type. |
| java.sql.Date | Provide support for Date SQL type. |
| java.sql.Driver | create an instance of a driver with the DriverManager. |
| java.sql.DriverManager | This class manages database drivers. |
| java.sql.PreparedStatement | Used to create and execute parameterized query. |
| java.sql.ResultSet | It is an interface that provide methods to access the result row-by-row. |
| java.sql.Savepoint | Specify savepoint in transaction. |
| java.sql.SQLException | Encapsulate all JDBC related exception. |
| java.sql.Statement | This interface is used to execute SQL statements. |

javax.sql package

This package is also known as JDBC extension API. It provides classes and interface to access server-side data.

Important classes and interface of javax.sql package

|  |  |
| --- | --- |
| classes/interface | Description |
| javax.sql.ConnectionEvent | Provide information about occurence of event. |
| javax.sql.ConnectionEventListener | Used to register event generated by PooledConnectionobject. |
| javax.sql.DataSource | Represent the DataSource interface used in an application. |
| javax.sql.PooledConnection | provide object to manage connection pools. |

**JDBC API Interfaces**

The JDBC library was designed as an interface for executing SQL statements.

A JDBC application is well insulated from the particular characteristics of the database system, which is being used & therefore for specific database it doesn’t have to be re-engineered. By having an implementation of the JDBC interface for each specific database, a driver, JDBC manages to operate with a variety of different relational database systems. This handles the mapping of java method calls in the JDBC classes to the database API.

JDBC defines following interfaces.

1. **Driver interface:**

This is the interface that every driver class must implement.

The Java SQL framework allows for multiple database drivers.

Each driver should supply a class that implements the Driver interface.

The DriverManager will try to load as many drivers as it can find and then for any given connection request, it will ask each driver in turn to try to connect to the target URL.

It is strongly recommended that each Driver class should be small and standalone so that the Driver class can be loaded and queried without bringing in vast quantities of supporting code.

When a Driver class is loaded, it should create an instance of itself and register it with the DriverManager. This means that a user can load and register a driver by calling

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")

**Method**

[**Connection**](https://docs.oracle.com/javase/7/docs/api/java/sql/Connection.html)**connect(**[**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html)**url,** [**Properties**](https://docs.oracle.com/javase/7/docs/api/java/util/Properties.html)**info) throws** [**SQLException**](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Attempts to make a database connection to the given URL. The driver should return "null" if it realizes it is the wrong kind of driver to connect to the given URL. This will be common, as when the JDBC driver manager is asked to connect to a given URL it passes the URL to each loaded driver in turn.

The driver should throw an SQLException if it is the right driver to connect to the given URL but has trouble connecting to the database.

The java.util.Properties argument can be used to pass arbitrary string tag/value pairs as connection arguments. Normally at least "user" and "password" properties should be included in the Properties object.

url - the URL of the database to which to connect

info - a list of arbitrary string tag/value pairs as connection arguments. Normally at least a "user" and "password" property should be included.

Returns:a Connection object that represents a connection to the URL

Throws:[SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html) - if a database access error occurs

**2) Connection Interface:**

A connection (session) with a specific database. SQL statements are executed and results are returned within the context of a connection.

A Connection object's database is able to provide information describing its tables, its supported SQL grammar, its stored procedures, the capabilities of this connection, and so on. This information is obtained with the **getMetaData** method.

**Note:** When configuring a Connection, JDBC applications should use the appropriate Connection method such as setAutoCommit or setTransactionIsolation. Applications should not invoke SQL commands directly to change the connection's configuration when there is a JDBC method available. By default a Connection object is in auto-commit mode, which means that it automatically commits changes after executing each statement. If auto-commit mode has been disabled, the method commit must be called explicitly in order to commit changes; otherwise, database changes will not be saved.

**Methods**

* + - [Statement](https://docs.oracle.com/javase/7/docs/api/java/sql/Statement.html) createStatement()
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)
    - Creates a Statement object for sending SQL statements to the database. SQL statements without parameters are normally executed using Statement objects. If the same SQL statement is executed many times, it may be more efficient to use a PreparedStatement object.

Result sets created using the returned Statement object will by default be type TYPE\_FORWARD\_ONLY and have a concurrency level of CONCUR\_READ\_ONLY

* + - [PreparedStatement](https://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html) prepareStatement([String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html" \o "class in java.lang) sql)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Creates a PreparedStatement object for sending parameterized SQL statements to the database.

A SQL statement with or without IN parameters can be pre-compiled and stored in a PreparedStatement object. This object can then be used to efficiently execute this statement multiple times.

**Note:** This method is optimized for handling parametric SQL statements that benefit from precompilation. If the driver supports precompilation, the method prepareStatement will send the statement to the database for precompilation. Some drivers may not support precompilation. In this case, the statement may not be sent to the database until the PreparedStatement object is executed. This has no direct effect on users; however, it does affect which methods throw certain SQLException objects.

Result sets created using the returned PreparedStatement object will by default be type TYPE\_FORWARD\_ONLY and have a concurrency level of CONCUR\_READ\_ONLY.

* + - [CallableStatement](https://docs.oracle.com/javase/7/docs/api/java/sql/CallableStatement.html) prepareCall([String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html" \o "class in java.lang) sql)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Creates a CallableStatement object for calling database stored procedures. The CallableStatement object provides methods for setting up its IN and OUT parameters, and methods for executing the call to a stored procedure.

**Note:** This method is optimized for handling stored procedure call statements. Some drivers may send the call statement to the database when the method prepareCall is done; others may wait until the CallableStatement object is executed. This has no direct effect on users; however, it does affect which method throws certain SQLExceptions.

Result sets created using the returned CallableStatement object will by default be type TYPE\_FORWARD\_ONLY and have a concurrency level of CONCUR\_READ\_ONLY.

* + - void setAutoCommit(boolean autoCommit)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Sets this connection's auto-commit mode to the given state. If a connection is in auto-commit mode, then all its SQL statements will be executed and committed as individual transactions. Otherwise, its SQL statements are grouped into transactions that are terminated by a call to either the method commit or the method rollback. By default, new connections are in auto-commit mode.

The commit occurs when the statement completes. The time when the statement completes depends on the type of SQL Statement:

* + - * For DML statements, such as Insert, Update or Delete, and DDL statements, the statement is complete as soon as it has finished executing.
      * For Select statements, the statement is complete when the associated result set is closed.
      * For CallableStatement objects or for statements that return multiple results, the statement is complete when all of the associated result sets have been closed, and all update counts and output parameters have been retrieved.

**NOTE:** If this method is called during a transaction and the auto-commit mode is changed, the transaction is committed. If setAutoCommit is called and the auto-commit mode is not changed, the call is a no-op.

* + - boolean getAutoCommit()
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Retrieves the current auto-commit mode for this Connection object.

* + - void commit()
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Makes all changes made since the previous commit/rollback permanent and releases any database locks currently held by this Connection object. This method should be used only when auto-commit mode has been disabled.

* + - void rollback()
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Undoes all changes made in the current transaction and releases any database locks currently held by this Connection object. This method should be used only when auto-commit mode has been disabled.

* + - void close()
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Releases this Connection object's database and JDBC resources immediately instead of waiting for them to be automatically released.

Calling the method close on a Connection object that is already closed is a no-op.

It is **strongly recommended** that an application explicitly commits or rolls back an active transaction prior to calling the close method. If the close method is called and there is an active transaction, the results are implementation-defined.

* + - [DatabaseMetaData](https://docs.oracle.com/javase/7/docs/api/java/sql/DatabaseMetaData.html) getMetaData()
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Retrieves a DatabaseMetaData object that contains metadata about the database to which this Connection object represents a connection. The metadata includes information about the database's tables, its supported SQL grammar, its stored procedures, the capabilities of this connection, and so on.

* + - [Statement](https://docs.oracle.com/javase/7/docs/api/java/sql/Statement.html) createStatement(int resultSetType,
    - int resultSetConcurrency)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Creates a Statement object that will generate ResultSet objects with the given type and concurrency. This method is the same as the createStatement method above, but it allows the default result set type and concurrency to be overridden.

* + - [PreparedStatement](https://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html) prepareStatement([String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html" \o "class in java.lang) sql,
    - int resultSetType,
    - int resultSetConcurrency)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Creates a PreparedStatement object that will generate ResultSet objects with the given type and concurrency. This method is the same as the prepareStatement method above, but it allows the default result set type and concurrency to be overridden.

* + - [CallableStatement](https://docs.oracle.com/javase/7/docs/api/java/sql/CallableStatement.html) prepareCall([String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html" \o "class in java.lang) sql,
    - int resultSetType,
    - int resultSetConcurrency)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Creates a CallableStatement object that will generate ResultSet objects with the given type and concurrency. This method is the same as the prepareCall method above, but it allows the default result set type and concurrency to be overridden.

**3) Statement Interface:**

* The object used for executing a static SQL statement and returning the results it produces.

By default, only one ResultSet object per Statement object can be open at the same time. Therefore, if the reading of one ResultSet object is interleaved with the reading of another, each must have been generated by different Statement objects. All execution methods in the Statement interface implicitly close a statment's current ResultSet object if an open one exists.

**Methods**

* + - [ResultSet](https://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html) executeQuery([String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html" \o "class in java.lang) sql)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Executes the given SQL statement, which returns a single ResultSet object.

* + - int executeUpdate([String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html" \o "class in java.lang) sql)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Executes the given SQL statement, which may be an INSERT, UPDATE, or DELETE statement or an SQL statement that returns nothing, such as an SQL DDL statement.

* + - void close()
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Releases this Statement object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed. It is generally good practice to release resources as soon as you are finished with them to avoid tying up database resources.

Calling the method close on a Statement object that is already closed has no effect.

* + - boolean execute([String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html" \o "class in java.lang) sql)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Executes the given SQL statement, which may return multiple results. In some (uncommon) situations, a single SQL statement may return multiple result sets and/or update counts. Normally you can ignore this unless you are (1) executing a stored procedure that you know may return multiple results or (2) you are dynamically executing an unknown SQL string.

The execute method executes an SQL statement and indicates the form of the first result. You must then use the methods getResultSet or getUpdateCount to retrieve the result, and getMoreResults to move to any subsequent result(s).

* + - [ResultSet](https://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html) getResultSet()
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Retrieves the current result as a ResultSet object. This method should be called only once per result.

* + - int getUpdateCount()
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Retrieves the current result as an update count; if the result is a ResultSet object or there are no more results, -1 is returned. This method should be called only once per result.

* + - int getResultSetType()
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Retrieves the result set type for ResultSet objects generated by this Statement object.

**Returns:**

one of ResultSet.TYPE\_FORWARD\_ONLY, ResultSet.TYPE\_SCROLL\_INSENSITIVE, or ResultSet.TYPE\_SCROLL\_SENSITIVE

**4) PreparedStatement interface:**

* An 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.

**Note:** The setter methods (setShort, setString, and so on) for setting IN parameter values must specify types that are compatible with the defined SQL type of the input parameter. For instance, if the IN parameter has SQL type INTEGER, then the method setInt should be used.

If arbitrary parameter type conversions are required, the method setObject should be used with a target SQL type.

In the following example of setting a parameter, con represents an active connection:

PreparedStatement pstmt = con.prepareStatement("UPDATE EMPLOYEES

SET SALARY = ? WHERE ID = ?");

pstmt.setBigDecimal(1, 153833.00)

pstmt.setInt(2, 110592)

**Methods**

* + - [ResultSet](https://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html) executeQuery()
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Executes the SQL query in this PreparedStatement object and returns the ResultSet object generated by the query.

* + - int executeUpdate()
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Executes the SQL statement in this PreparedStatement object, which must be an SQL Data Manipulation Language (DML) statement, such as INSERT, UPDATE or DELETE; or an SQL statement that returns nothing, such as a DDL statement.

* + - void setNull(int parameterIndex,
    - int sqlType)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Sets the designated parameter to SQL NULL.

* + - void setBoolean(int parameterIndex,
    - boolean x)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Sets the designated parameter to the given Java boolean value. The driver converts this to an SQL BIT or BOOLEAN value when it sends it to the database.

* + - void setByte(int parameterIndex,
    - byte x)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Sets the designated parameter to the given Java byte value. The driver converts this to an SQL TINYINT value when it sends it to the database.

* + - void setShort(int parameterIndex,
    - short x)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Sets the designated parameter to the given Java short value. The driver converts this to an SQL SMALLINT value when it sends it to the database.

* + - void setInt(int parameterIndex,
    - int x)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Sets the designated parameter to the given Java int value. The driver converts this to an SQL INTEGER value when it sends it to the database.

* + - void setLong(int parameterIndex,
    - long x)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Sets the designated parameter to the given Java long value. The driver converts this to an SQL BIGINT value when it sends it to the database.

* + - void setFloat(int parameterIndex,
    - float x)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Sets the designated parameter to the given Java float value. The driver converts this to an SQL REAL value when it sends it to the database.

* + - void setDouble(int parameterIndex,
    - double x)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Sets the designated parameter to the given Java double value. The driver converts this to an SQL DOUBLE value when it sends it to the database.

* + - void setString(int parameterIndex,
    - [String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) x)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Sets the designated parameter to the given Java String value. The driver converts this to an SQL VARCHAR or LONGVARCHAR value (depending on the argument's size relative to the driver's limits on VARCHAR values) when it sends it to the database.

* + - void setDate(int parameterIndex,
    - [Date](https://docs.oracle.com/javase/7/docs/api/java/sql/Date.html) x)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Sets the designated parameter to the given java.sql.Date value using the default time zone of the virtual machine that is running the application. The driver converts this to an SQL DATE value when it sends it to the database.

* + - void clearParameters()
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Clears the current parameter values immediately.

In general, parameter values remain in force for repeated use of a statement. Setting a parameter value automatically clears its previous value. However, in some cases it is useful to immediately release the resources used by the current parameter values; this can be done by calling the method clearParameters.

* + - boolean execute()
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Executes the SQL statement in this PreparedStatement object, which may be any kind of SQL statement. Some prepared statements return multiple results; the execute method handles these complex statements as well as the simpler form of statements handled by the methods executeQuery and executeUpdate.

**5) ResultSet Interface:**

* A table of data representing a database result set, which is usually generated by executing a statement that queries the database.

A ResultSet object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The next method moves the cursor to the next row, and because it returns false when there are no more rows in the ResultSet object, it can be used in a while loop to iterate through the result set.

A default ResultSet object is not updatable and has a cursor that moves forward only. Thus, you can iterate through it only once and only from the first row to the last row. 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. See ResultSet fields for other options.

Statement stmt = con.createStatement(

ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_UPDATABLE);

ResultSet rs = stmt.executeQuery("SELECT a, b FROM TABLE2");

// rs will be scrollable, will not show changes made by others,

// and will be updatable

The ResultSet interface provides *getter* methods (getBoolean, getLong, and so on) for retrieving column values from the current row. Values can be retrieved using either the index number of the column or the name of the column. In general, using the column index will be more efficient. Columns are numbered from 1. For maximum portability, result set columns within each row should be read in left-to-right order, and each column should be read only once.

For the getter methods, a JDBC driver attempts to convert the underlying data to the Java type specified in the getter method and returns a suitable Java value. The JDBC specification has a table showing the allowable mappings from SQL types to Java types that can be used by the ResultSet getter methods.

**Methods**

* + - boolean next()
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Moves the cursor froward one row from its current position. A ResultSet cursor is initially positioned before the first row; the first call to the method next makes the first row the current row; the second call makes the second row the current row, and so on.

When a call to the next method returns false, the cursor is positioned after the last row. Any invocation of a ResultSet method which requires a current row will result in a SQLException being thrown. If the result set type is TYPE\_FORWARD\_ONLY, it is vendor specified whether their JDBC driver implementation will return false or throw an SQLException on a subsequent call to next.

If an input stream is open for the current row, a call to the method next will implicitly close it. A ResultSet object's warning chain is cleared when a new row is read

* + - boolean next()
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Moves the cursor froward one row from its current position. A ResultSet cursor is initially positioned before the first row; the first call to the method next makes the first row the current row; the second call makes the second row the current row, and so on.

When a call to the next method returns false, the cursor is positioned after the last row. Any invocation of a ResultSet method which requires a current row will result in a SQLException being thrown. If the result set type is TYPE\_FORWARD\_ONLY, it is vendor specified whether their JDBC driver implementation will return false or throw an SQLException on a subsequent call to next.

If an input stream is open for the current row, a call to the method next will implicitly close it. A ResultSet object's warning chain is cleared when a new row is read.

**Returns:**

true if the new current row is valid; false if there are no more rows

**Throws:**

[SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html) - if a database access error occurs or this method is called on a closed result set

* + - **close**
    - void close()

throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Releases this ResultSet object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed.

The closing of a ResultSet object does **not** close the Blob, Clob or NClob objects created by the ResultSet. Blob, Clob or NClob objects remain valid for at least the duration of the transaction in which they are creataed, unless their free method is invoked.

When a ResultSet is closed, any ResultSetMetaData instances that were created by calling the getMetaData method remain accessible.

**Note:** A ResultSet object is automatically closed by the Statement object that generated it when that Statement object is closed, re-executed, or is used to retrieve the next result from a sequence of multiple results.

* + - [String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) getString(int columnIndex)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Retrieves the value of the designated column in the current row of this ResultSet object as a String in the Java programming language.

* + - boolean getBoolean(int columnIndex)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Retrieves the value of the designated column in the current row of this ResultSet object as a boolean in the Java programming language.

If the designated column has a datatype of CHAR or VARCHAR and contains a "0" or has a datatype of BIT, TINYINT, SMALLINT, INTEGER or BIGINT and contains a 0, a value of false is returned. If the designated column has a datatype of CHAR or VARCHAR and contains a "1" or has a datatype of BIT, TINYINT, SMALLINT, INTEGER or BIGINT and contains a 1, a value of true is returned.

* + - byte getByte(int columnIndex)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Retrieves the value of the designated column in the current row of this ResultSet object as a byte in the Java programming language.

* + - short getShort(int columnIndex)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Retrieves the value of the designated column in the current row of this ResultSet object as a short in the Java programming language.

* + - int getInt(int columnIndex)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Retrieves the value of the designated column in the current row of this ResultSet object as an int in the Java programming language.

* + - long getLong(int columnIndex)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Retrieves the value of the designated column in the current row of this ResultSet object as a long in the Java programming language.

* + - float getFloat(int columnIndex)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Retrieves the value of the designated column in the current row of this ResultSet object as a float in the Java programming language.

* + - double getDouble(int columnIndex)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Retrieves the value of the designated column in the current row of this ResultSet object as a double in the Java programming language.

* + - [Date](https://docs.oracle.com/javase/7/docs/api/java/sql/Date.html) getDate(int columnIndex)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Retrieves the value of the designated column in the current row of this ResultSet object as a java.sql.Date object in the Java programming language.

* + - [String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) getString([String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html" \o "class in java.lang) columnLabel)
    - throws [SQLException](https://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)

Retrieves the value of the designated column in the current row of this ResultSet object as a String in the Java programming language.

**All Above methods can have String Colname as parameter instead of index**

**Procedure For Creating Database with MS Access**

Open MS Access-> Blank database-> Provide database name-> Now right click on table1 and open in design view-> provide columns and their data types-> now provide table1 name of your choice-> now double click on table name -> enter data in table-> save database

**Procedure for creating database with Oracle**

Open oracle database -> login with user name & password-> using sql query syntaxes create table and insert data into that

**Steps for creating DSN when using JDBC-ODBC bridge driver**

Control panel-> Administrative tools -> ODBC data source-> click on add button

->select driver name for database(Microsoft Access driver(\*.mdb,\*.accdb) for access database and Oracle driver for oracle database)-> provide dsn name -> if it is access database then select the database name from your path-> click on ok.

**Steps for using Type4 driver for oracle database**

Download ojdbc14.jar file from internet and then paste that file in

C:/program files/java/jre/lib/ext folder

Then update that file path in environment variable.

C:/program files/java/jre/lib/ext/ojdbc14.jar

## Fundamental Steps in JDBC

The fundamental steps involved in the process of connecting to a database and executing a query consist of the following:

* Import JDBC packages.
* Load and register the JDBC driver.
* Open a connection to the database.
* Create a statement object to perform a query.
* Execute the statement object and return a query resultset.
* Process the resultset.
* Close the resultset and statement objects.
* Close the connection.

These steps are described in detail in the sections that follow.

### Import JDBC Packages

This is for making the JDBC API classes immediately available to the application program. The following import statement should be included in the program irrespective of the JDBC driver being used:

import java.sql.\*;

Additionally, depending on the features being used, Oracle-supplied JDBC packages might need to be imported. For example, the following packages might need to be imported while using the Oracle extensions to JDBC such as using advanced data types such as BLOB, and so on.

import oracle.jdbc.driver.\*;

import oracle.sql.\*;

### Load and Register the JDBC Driver

This is for establishing a communication between the JDBC program and the Oracle database. This is done by using the static registerDriver() method of the DriverManager class of the JDBC API. The following line of code does this job:

DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

**JDBC Driver Registration**

For the entire Java application, the JDBC driver is registered only once per each database that needs to be accessed. This is true even when there are multiple database connections to the same data server.

Alternatively, the forName() method of the java.lang.Class class can be used to load and register the JDBC driver:

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

However, the forName() method is valid for only JDK-compliant Java Virtual Machines and implicitly creates an instance of the Oracle driver, whereas the registerDriver() method does this explicitly.

### Connecting to a Database

Once the required packages have been imported and the Oracle JDBC driver has been loaded and registered, a database connection must be established. This is done by using the getConnection()method of the DriverManager class. A call to this method creates an object instance of thejava.sql.Connection class. The getConnection() requires three input parameters, namely, a connect string, a username, and a password. The connect string should specify the JDBC driver to be yes and the database instance to connect to.

The getConnection() method is an overloaded method that takes

* Three parameters, one each for the URL, username, and password.
* Only one parameter for the database URL. In this case, the URL contains the username and password.

The following lines of code illustrate using the getConnection() method:

Connection conn = DriverManager.getConnection(URL, username, passwd);

Connection conn = DriverManager.getConnection(URL);

where URL, username, and passwd are of String data types.

We will discuss the methods of opening a connection using the Oracle JDBC OCI and thin \_drivers.

When using the OCI driver, the database can be specified using the TNSNAMES entry in the tnsnames.ora file. For example, to connect to a database on a particular host as user oratest and password oratest that has a TNSNAMES entry of oracle.world, use the following code:

Connection conn = DriverManager.getConnection("jdbc:oracle:oci8:

@oracle.world", "oratest", "oratest");

Both the ":" and "@" are mandatory.

When using the JDBC thin driver, the TNSNAMES entry cannot be used to identify the database. There are two ways of specifying the connect string in this case, namely,

* Explicitly specifying the hostname, the TCP/IP port number, and the Oracle SID of the database to connect to. This is for thin driver only.
* Specify a Net8 keyword-value pair list.

For example, for the explicit method, use the following code to connect to a database on host trainingwhere the TCP/IP listener is on port 1521, the SID for the database instance is Oracle, the username and password are both oratest:

Connection conn = DriverManager.getConnection

("jdbc:oracle:thin:@training:1521:Oracle",

"oratest", "oratest");

For the Net8 keyword-value pair list, use the following:

Connection conn = DriverManager.getConnection

("jdbc:oracle:thin@(description=(address=

(host=training)(protocol=tcp)(port=1521))

(connect\_data=(sid=Oracle))) ", \_"oratest", "oratest");

This method can also be used for the JDBC OCI driver. Just specify oci8 instead of thin in the above keyword-value pair list.

### Querying the Database

Querying the database involves two steps: first, creating a statement object to perform a query, and second, executing the query and returning a resultset.

#### Creating a Statement Object

This is to instantiate objects that run the query against the database connected to. This is done by the createStatement() method of the conn Connection object created above. A call to this method creates an object instance of the Statement class. The following line of code illustrates this:

Statement sql\_stmt = conn.createStatement();

#### Executing the Query and Returning a ResultSet

Once a Statement object has been constructed, the next step is to execute the query. This is done by using the executeQuery() method of the Statement object. A call to this method takes as parameter a SQL SELECT statement and returns a JDBC ResultSet object. The following line of code illustrates this using the sql\_stmt object created above:

ResultSet rset = sql\_stmt.executeQuery

("SELECT empno, ename, sal, deptno FROM emp ORDER BY ename");

Alternatively, the SQL statement can be placed in a string and then this string passed to the executeQuery() function. This is shown below.

String sql = "SELECT empno, ename, sal, deptno FROM emp ORDER BY ename";

ResultSet rset = sql\_stmt.executeQuery(sql);

**Statement and ResultSet Objects**

Statement and ResultSet objects open a corresponding cursor in the database for SELECT and other DML statements.

The above statement executes the SELECT statement specified in between the double quotes and stores the resulting rows in an instance of the ResultSet object named rset.

### Processing the Results of a Database Query That Returns Multiple Rows

Once the query has been executed, there are two steps to be carried out:

* Processing the output resultset to fetch the rows
* Retrieving the column values of the current row

The first step is done using the next() method of the ResultSet object. A call to next() is executed in a loop to fetch the rows one row at a time, with each call to next() advancing the control to the next available row. The next() method returns the Boolean value true while rows are still available for fetching and returns false when all the rows have been fetched.

The second step is done by using the getXXX() methods of the JDBC rset object. Here getXXX()corresponds to the getInt(), getString() etc with XXX being replaced by a Java datatype.

The following code demonstrates the above steps:

String str;

while (rset.next())

{

str = rset.getInt(1)+ " "+ rset.getString(2)+ "

"+rset.getFloat(3)+ " "rset.getInt(4)+ "\n";

}

byte buf[] = str.getBytes();

OutputStream fp = new FileOutputStream("query1.lst");

fp.write(buf);

fp.close();

Here the 1, 2, 3, and 4 in rset.getInt(), rset.getString(), getFloat(), and getInt()respectively denote the position of the columns in the SELECT statement, that is, the first column empno, second column ename, third column sal, and fourth column deptno of the SELECT statement respectively.

**Specifying get() Parameters**

The parameters for the getXXX() methods can be specified by position of the corresponding columns as numbers 1, 2, and so on, or by directly specifying the column names enclosed in double quotes, asgetString("ename") and so on, or a combination of both.

### Closing the ResultSet and Statement

Once the ResultSet and Statement objects have been used, they must be closed explicitly. This is done by calls to the close() method of the ResultSet and Statement classes. The following code illustrates this:

rset.close();

sql\_stmt.close();

If not closed explicitly, there are two disadvantages:

* Memory leaks can occur
* Maximum Open cursors can be exceeded

Closing the ResultSet and Statement objects frees the corresponding cursor in the database.

### Closing the Connection

The last step is to close the database connection opened in the beginning after importing the packages and loading the JDBC drivers. This is done by a call to the close() method of the Connection class.

The following line of code does this:

conn.close();

**Explicitly Close your Connection**

Closing the ResultSet and Statement objects does not close the connection. The connection should be closed by explicitly invoking the close() method of the Connection class.

A complete example of the above procedures using a JDBC thin driver is given below. This program queries the emp table and writes the output rows to an operating system file.

//Import JDBC package

import java.sql.\*;

// Import Java package for File I/O

import java.io.\*;

public class QueryExample {

public static void main (String[] args) throws SQLException, IOException

{

//Load and register Oracle driver

DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

//Establish a connection

Connection conn = DriverManager.getConnection("jdbc:oracle:thin:

@training:1521:Oracle", "oratest", "oratest");

//Create a Statement object

Statement sql\_stmt = conn.createStatement();

//Create a ResultSet object, execute the query and return a

// resultset

ResultSet rset = sql\_stmt.executeQuery("SELECT empno, ename, sal,

deptno FROM emp ORDER BY ename");

//Process the resultset, retrieve data in each row, column by column

//and write to an operating system file

String str = "";

while (rset.next())

{

str += rset.getInt(1)+" "+ rset.getString(2)+" "+

rset.getFloat(3)+" "+rset.getInt(4)+"\n";

}

byte buf[] = str.getBytes();

OutputStream fp = new FileOutputStream("query1.lst");

fp.write(buf);

fp.close();

//Close the ResultSet and Statement

rset.close();

sql\_stmt.close();

//Close the database connection

conn.close();

}

}

### Processing the Results of a Database Query That Returns a Single Row

The above sections and the complete example explained the processing of a query that returned multiple rows. This section highlights the processing of a single-row query and explains how to write code that is the analogue of the PL/SQL exception NO\_DATA\_FOUND.

**NO DATA FOUND Exception**

NO\_DATA\_FOUND exception in PL/SQL is simulated in JDBC by using the return value of the next()method of the ResultSet object. A value of false returned by the next() method identifies aNO\_DATA\_FOUND exception.

Consider the following code (this uses the ResultSet object rset defined in the above sections):

if (rset.next())

// Process the row returned

else

System.out.println("The Employee with Empno "+ args[1] +

"does not exist");

Instead of the while loop used earlier, an if statement is used to determine whether the SELECT statement returned a row or not.

**Example Programs**

**// Using Jdbc-Odbc bridge driver**

import java.sql.\*;

class demodata

{

public static void main(String ar[])throws Exception

{

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Connection c=DriverManager.getConnection("jdbc:odbc:sample");//for access

/\*Connection c= DriverManager.getConnection("jdbc:odbc:sample","system","manager");//for oracle\*/

Statement st=c.createStatement();

ResultSet rs=st.executeQuery("select \* from student");

while(rs.next())

{

System.out.println(rs.getInt(1)+"\t"+rs.getString(2)+"\t"+rs.getInt(3));

}

st.close();

c.close();

}

}

**// Using Type 4 driver of oracle**

import java.sql.\*;

class demooradata

{

public static void main(String ar[])throws Exception

{

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

Connection c= DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","manager");

Statement st=c.createStatement();

ResultSet rs=st.executeQuery("select \* from student");

while(rs.next())

{

System.out.println(rs.getInt(1)+"\t"+rs.getString(2)+"\t"+rs.getInt(3));

}

st.close();

c.close();

}

}

**Example on PreparedStatement**

import java.sql.\*;

class InsertPrepared

{

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","manager");

  PreparedStatement stmt=con.prepareStatement("insert into stud values(?,?,?)");

stmt.setInt(1,10);//1 specifies the first parameter in the query

stmt.setString(2,"Ratan");

stmt.setFloat(3,80.10);

int i=stmt.executeUpdate();

System.out.println(i+" records inserted");

con.close();

}

}