**Java JDBC:**

**JDBC** stands for ***Java Database Connectivity***.

* JDBC is an API specification for connecting java applications to data in popular databases.
* The JDBC API lets us encode access request statements in Structured Query Language(SQL) that are then passed to the application that manages the database. It returns the results through a similar interface.
* Simply means, enable a standards-based method to access data using Java language.

JDBC:ODBC enables a single JDBC application to access number of data sources and can rub on any machine with a Java Virtual Machine(JVM).

JDBC data standard defines a set of Java interfaces enable application developer’s abstract data access functionality including:

* Establish Connection with a data source.
* Execute SQL queries.
* Process result sets.

**DriverManager Class:**

Connection con=DriverManager.getConnection(jdbc:mysql://localhost:3306/mysql)

# **Establishing a Connection:**

First, you need to establish a connection with the data source you want to use. A data source can be a DBMS, a legacy file system, or some other source of data with a corresponding JDBC driver. Typically, a JDBC application connects to a target data source using one of two classes:

* **DriverManager**: This fully implemented class connects an application to a data source, which is specified by a database URL. When this class first attempts to establish a connection, it automatically loads any JDBC 4.0 drivers found within the class path. Note that your application must manually load any JDBC drivers prior to version 4.0.
* **DataSource**: This interface is preferred over DriverManager because it allows details about the underlying data source to be transparent to your application. A DataSource object's properties are set so that it represents a particular data source.

Note: Previously, to obtain connection, first we had to initialize the JDBC driver by calling the method  Class.forName. This method required an object of type java.sql.Driver. Each JDBC driver contains one or more classes that implements the interface java.sql.Driver. The drivers for Java DB are org.apache.derby.jdbc.EmbeddedDriver and org.apache.derby.jdbc.ClientDriver, and the one for MySQL Connector/J is com.mysql.jdbc.Driver.

Any JDBC 4.0 drivers that are found in your class path are automatically loaded. (However, you must manually load any drivers prior to JDBC 4.0 with the method Class.forName.)

The method returns a Connection object, which represents a connection with the DBMS or a specific database. Query the database through this object.

## Using the DriverManager Class

Connecting to your DBMS with the DriverManager class involves calling the method DriverManager.getConnection.

public Connection getConnection() throws SQLException {

Connection conn = null;

Properties connectionProps = new Properties();

connectionProps.put("user", this.userName);

connectionProps.put("password", this.password);

if (this.dbms.equals("mysql")) {

conn = DriverManager.getConnection(

"jdbc:" + this.dbms + "://" +

this.serverName +

":" + this.portNumber + "/",

connectionProps);

} else if (this.dbms.equals("derby")) {

conn = DriverManager.getConnection(

"jdbc:" + this.dbms + ":" +

this.dbName +

";create=true",

connectionProps);

}

System.out.println("Connected to database");

return conn;

}

The method DriverManager.getConnection establishes a database connection. This method requires a database URL, which varies depending on your DBMS. The following are some examples of database URLs:

## Specifying Database Connection URLs

A database connection URL is a string that your DBMS JDBC driver uses to connect to a database. It can contain information such as where to search for the database, the name of the database to connect to, and configuration properties. The exact syntax of a database connection URL is specified by your DBMS.

### Java DB Database Connection URLs

The following is the database connection URL syntax for Java DB:

jdbc:derby:[subsubprotocol:][databaseName]

[;attribute=value]\*

* subsubprotocol specifies where Java DB should search for the database, either in a directory, in memory, in a class path, or in a JAR file. It is typically omitted.
* databaseName is the name of the database to connect to.
* attribute=value represents an optional, semicolon-separated list of attributes. These attributes enable you to instruct Java DB to perform various tasks, including the following:
  + Create the database specified in the connection URL.
  + Encrypt the database specified in the connection URL.
  + Specify directories to store logging and trace information.
  + Specify a user name and password to connect to the database.

### MySQL Connector/J Database URL

The following is the database connection URL syntax for MySQL Connector/J:

jdbc:mysql://[host][,failoverhost...]

[:port]/[database]

[?propertyName1][=propertyValue1]

[&propertyName2][=propertyValue2]...

* host:port is the host name and port number of the computer hosting your database. If not specified, the default values of host and port are 127.0.0.1 and 3306, respectively.
* database is the name of the database to connect to. If not specified, a connection is made with no default database.
* failover is the name of a standby database (MySQL Connector/J supports failover).
* propertyName=propertyValue represents an optional, ampersand-separated list of properties. These attributes enable you to instruct MySQL Connector/J to perform various tasks.

# **Connecting with DataSource Objects:**

DataSource objects can provide connection pooling and distributed transactions. This functionality is essential for enterprise database computing. In particular, it is integral to Enterprise JavaBeans (EJB) technology.

The work performed to deploy the classes that make these operations possible, which a system administrator usually does with a tool (such as Apache Tomcat or Oracle WebLogic Server), varies with the type of DataSource object that is being deployed. As a result, most of this section is devoted to showing how a system administrator sets up the environment so that programmers can use a DataSource object to get connections.

Objects instantiated by classes that implement the DataSource represent a particular DBMS or some other data source, such as a file. A DataSource object represents a particular DBMS or some other data source, such as a file. If a company uses more than one data source, it will deploy a separate DataSource object for each of them. The DataSource interface is implemented by a driver vendor. It can be implemented in three different ways:

* A basic DataSource implementation produces standard Connection objects that are not pooled or used in a distributed transaction.
* A DataSource implementation that supports connection pooling produces Connection objects that participate in connection pooling, that is, connections that can be recycled.
* A DataSource implementation that supports distributed transactions produces Connection objects that can be used in a distributed transaction, that is, a transaction that accesses two or more DBMS servers.

A JDBC driver should include at least a basic DataSource implementation. For example, the Java DB JDBC driver includes the implementation org.apache.derby.jdbc.ClientDataSource and for MySQL, com.mysql.jdbc.jdbc2.optional.MysqlDataSource. If your client runs on Java 8 compact profile 2, then the Java DB JDBC driver is org.apache.derby.jdbc.BasicClientDataSource40. The sample for this tutorial requires compact profile 3 or greater.

A DataSource class that supports distributed transactions typically also implements support for connection pooling. For example, a DataSource class provided by an EJB vendor almost always supports both connection pooling and distributed transactions.

Suppose that the owner of the thriving chain of The Coffee Break shops, from the previous examples, has decided to expand further by selling coffee over the Internet. With the large amount of online business expected, the owner will definitely need connection pooling. Opening and closing connections involves a great deal of overhead, and the owner anticipates that this online ordering system will necessitate a sizable number of queries and updates. With connection pooling, a pool of connections can be used over and over again, avoiding the expense of creating a new connection for every database access. In addition, the owner now has a second DBMS that contains data for the recently acquired coffee roasting company. This means that the owner will want to be able to write distributed transactions that use both the old DBMS server and the new one.

The chain owner has reconfigured the computer system to serve the new, larger customer base. The owner has purchased the most recent JDBC driver and an EJB application server that works with it to be able to use distributed transactions and get the increased performance that comes with connection pooling. Many JDBC drivers are available that are compatible with the recently purchased EJB server. The owner now has a three-tier architecture, with a new EJB application server and JDBC driver in the middle tier and the two DBMS servers as the third tier. Client computers making requests are the first tier.

The system administrator needs to deploy DataSource objects so that The Coffee Break's programming team can start using them. Deploying a DataSource object consists of three tasks:

1. Creating an instance of the DataSource class
2. Setting its properties
3. Registering it with a naming service that uses the Java Naming and Directory Interface (JNDI) API

Suppose a company that wants only a basic implementation of DataSource has bought a driver from the JDBC vendor DB Access, Inc. This driver includes the class com.dbaccess.BasicDataSource that implements the DataSource interface. The following code excerpt creates an instance of the class BasicDataSource and sets its properties. After the instance of BasicDataSource is deployed, a programmer can call the method DataSource.getConnection to get a connection to the company's database, CUSTOMER\_ACCOUNTS. First, the system administrator creates the BasicDataSource object ds using the default constructor. The system administrator then sets three properties. Note that the following code is typically be executed by a deployment tool:

com.dbaccess.BasicDataSource ds = new com.dbaccess.BasicDataSource();

ds.setServerName("grinder");

ds.setDatabaseName("CUSTOMER\_ACCOUNTS");

ds.setDescription("Customer accounts database for billing");

The variable ds now represents the database CUSTOMER\_ACCOUNTS installed on the server. Any connection produced by the BasicDataSource object ds will be a connection to the database CUSTOMER\_ACCOUNTS.

### Registering DataSource Object with Naming Service That Uses JNDI API

With the properties set, the system administrator can register the BasicDataSource object with a JNDI (Java Naming and Directory Interface) naming service. The particular naming service that is used is usually determined by a system property, which is not shown here. The following code excerpt registers the BasicDataSource object and binds it with the logical name jdbc/billingDB:

Context ctx = new InitialContext();

ctx.bind("jdbc/billingDB", ds);

This code uses the JNDI API. The first line creates an InitialContext object, which serves as the starting point for a name, similar to root directory in a file system. The second line associates, or binds, the BasicDataSource object ds to the logical name jdbc/billingDB. In the next code excerpt, you give the naming service this logical name, and it returns the BasicDataSource object. The logical name can be any string. In this case, the company decided to use the name billingDB as the logical name for the CUSTOMER\_ACCOUNTS database.

**Creating Statements:**

A **Statement** is an **interface** that represents a SQL statement. You execute Statement objects, and they generate ResultSet objects, which is a table of data representing a database result set. You need a Connection object to create a Statement object.

For example, CoffeesTables.viewTable creates a Statement object with the following code:

stmt = con.createStatement();

There are three different kinds of statements:

* **Statement**: Used to implement simple SQL statements with no parameters.
* **PreparedStatement**: (Extends Statement.) Used for precompiling SQL statements that might contain input parameters.
* **CallableStatement**: (Extends PreparedStatement.) Used to execute stored procedures that may contain both input and output parameters.

**Executing Queries**

To execute a query, call an execute method from Statement such as the following:

* **execute**: Returns true if the first object that the query returns is a ResultSet object. Use this method if the query could return one or more ResultSet objects. Retrieve the ResultSet objects returned from the query by repeatedly calling Statement.getResultSet.
* **executeQuery**: Returns one ResultSet object.
* **executeUpdate**: Returns an integer representing the number of rows affected by the SQL statement. Use this method if you are using INSERT, DELETE, or UPDATE SQL statements.

For example, CoffeesTables.viewTable executed a Statement object with the following code:

ResultSet rs = stmt.executeQuery(query);

## Overview of Prepared Statements

Sometimes it is more convenient to use a PreparedStatement object for sending SQL statements to the database. This special type of statement is derived from the more general class, Statement, that you already know.

If you want to execute a Statement object many times, it usually reduces execution time to use a PreparedStatement object instead.

The main feature of a PreparedStatement object is that, unlike a Statement object, it is given a SQL statement when it is created. The advantage to this is that in most cases, this SQL statement is sent to the DBMS right away, where it is compiled. As a result, the PreparedStatement object contains not just a SQL statement, but a SQL statement that has been precompiled. This means that when the PreparedStatement is executed, the DBMS can just run the PreparedStatement SQL statement without having to compile it first.

Although PreparedStatement objects can be used for SQL statements with no parameters, you probably use them most often for SQL statements that take parameters. The advantage of using SQL statements that take parameters is that you can use the same statement and supply it with different values each time you execute it. Examples of this are in the following sections.

# Using RowSet Objects

A JDBC RowSet object holds tabular data in a way that makes it more flexible and easier to use than a result set.

Oracle has defined five RowSet interfaces for some of the more popular uses of a RowSet, and standard reference are available for these RowSet interfaces. In this tutorial you will learn how to use these reference implementations.

These versions of the RowSet interface and their implementations have been provided as a convenience for programmers. Programmers are free to write their own versions of the javax.sql.RowSet interface, to extend the implementations of the five RowSet interfaces, or to write their own implementations. However, many programmers will probably find that the standard reference implementations already fit their needs and will use them as is.

This section introduces you to the RowSet interface and the following interfaces that extend this interface:

* JdbcRowSet
* CachedRowSet
* WebRowSet
* JoinRowSet
* FilteredRowSet

# Using JdbcRowSet Objects

A JdbcRowSet object is an enhanced ResultSet object. It maintains a connection to its data source, just as a ResultSet object does. The big difference is that it has a set of properties and a listener notification mechanism that make it a JavaBeans component.

One of the main uses of a JdbcRowSet object is to make a ResultSet object scrollable and updatable when it does not otherwise have those capabilities.

### Using the Default Constructor

The first statement in the following code excerpt creates an empty JdbcRowSet object.

public void createJdbcRowSet(String username, String password) {

jdbcRs = new JdbcRowSetImpl();

jdbcRs.setCommand("select \* from COFFEES");

jdbcRs.setUrl("jdbc:myDriver:myAttribute");

jdbcRs.setUsername(username);

jdbcRs.setPassword(password);

jdbcRs.execute();

// ...

}

The object jdbcRs contains no data until you specify a SQL statement with the method setCommand, specify how the JdbcResultSet object connects the database, and then run the method execute.

# Using CachedRowSetObjects

A CachedRowSet object is special in that it can operate without being connected to its data source, that is, it is a *disconnected* RowSet object. It gets its name from the fact that it stores (caches) its data in memory so that it can operate on its own data rather than on the data stored in a database.

The CachedRowSet interface is the superinterface for all disconnected RowSet objects, so everything demonstrated here also applies to WebRowSet, JoinRowSet, and FilteredRowSet objects.

Note that although the data source for a CachedRowSet object (and the RowSet objects derived from it) is almost always a relational database, a CachedRowSet object is capable of getting data from any data source that stores its data in a tabular format. For example, a flat file or spreadsheet could be the source of data. This is true when the RowSetReader object for a disconnected RowSet object is implemented to read data from such a data source. The reference implementation of the CachedRowSet interface has a RowSetReader object that reads data from a relational database,

**ResultSet Interface**: provides methods to retrieving and manipulating the results of executed quries,

The use of ResultSet objects and cursors

A ResultSet object is a table of data representing a database result set, which is usually generated by executing a statement that queries the database. For example, the [CoffeeTables.viewTable](https://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html) method creates a ResultSet, rs, when it executes the query through the Statement object, stmt.

Note that a ResultSet object can be created through any object that implements the Statement interface, including PreparedStatement, CallableStatement, and RowSet.

You access the data in a ResultSet object through a cursor. Note that this cursor is not a database cursor. This cursor is a pointer that points to one row of data in the ResultSet. Initially, the cursor is positioned before the first row. The method ResultSet.next moves the cursor to the next row. This method returns false if the cursor is positioned after the last row. This method repeatedly calls the ResultSet.next method with a while loop to iterate through all the data in the ResultSet.

To execute the SQL query, use a Statement object. To create a Statement object, call the method Connection.createStatement from an existing Connection object. To execute a SQL query, call the method Statement.executeUpdate.

All Statement objects are closed when the connection that created them is closed. However, it is good coding practice to explicitly close Statement objects as soon as you are finished with them. This allows any external resources that the statement is using to be released immediately. Close a statement by calling the method Statement.close. Place this statement in a finally to ensure that it closes even if the normal program flow is interrupted because an exception (such as SQLException) is thrown.

**JDBC Driver** is a software component that is used to interact with the java application with the database.

The purpose of JDBC driver is to convert java calls into database-specific calls and database specific calls into java calls.

**Types of JDBC Drivers:**

1. JDBC-ODBC bridge driver. ( also known as Type-1 driver).
2. Native API driver. ( also known as Type-2 driver).
3. Network protocol driver. ( also known as Type-3 driver).
4. Pure java driver. ( also known as Type-4 driver).

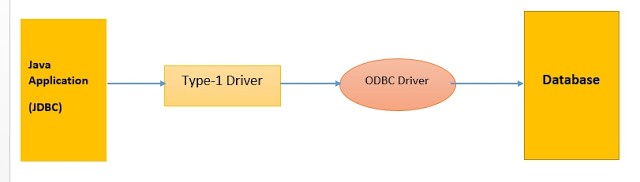
**1. JDBC-ODBC bridge Driver:**

> This driver **Internally** will **take support** of **ODBC Driver** to **communicate** with **database**. It converts **JDBC calls** into **ODBC calls** and **ODBC Driver** converts **ODBC calls** into **database-specific calls**.

> Using JDBC Driver for prototyping only not for production purposes.

> platform dependent.

> provided by Sun Microsystems as a part of JDK.



**2. Native API Driver:**

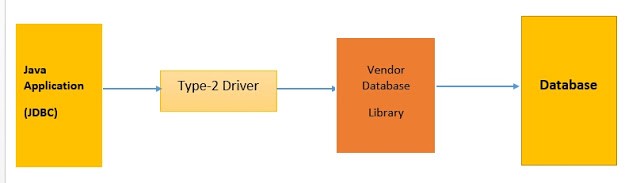
> Native API driver converts **JDBC calls** into **database-specific native libraries calls** and these are **directly understood** by the **database engine**.

> Large database vendors, such as oracle and IBM, user this driver for their enterprise databases.

> these drivers force developers to write platform specific code.

> these drivers are operating system specific and complied.

> is database dependent driver and platform-dependent driver.



**3. Network protocol driver:**

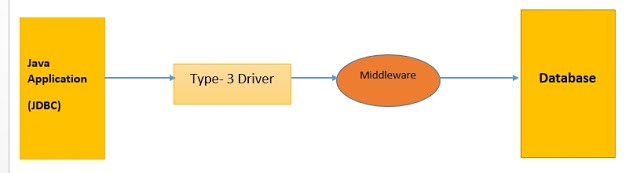
> Network protocol drivers are pure Java drivers, for database middleware.

> Java application **communicates** with **Network Protocol driver**, **Network protocol driver** converts **JDBC calls** into **middle-ware specific calls**, the **middle-ware server communicates** with **database**, the **middle-ware server** converts **middle-ware specific calls** into **database-specific calls**.

> This driver does not directly communicate with database, it is database independent driver, for any database this driver is same.

> it is written in Java, so it is platform independent.

> No client-side libraries are required.



**4.Pure Java Driver:**

> This driver **converts JDBC calls** into **database-specific calls directly**.

> This driver **directly communicates** with the database by using database specific native protocols provided by the database vendor

> It is platform independent Driver

