2. JAVA DATABASE CONNECTIVITY (JDBC)

Introduction to JDBC

JDBC is a Java API that enables Java programs to execute SQL statements. This allows Java programs to interact with any SQL compliant database. JDBC API provides universal data access from the Java language. It provides methods for querying and updating data in the database.

As Java itself runs on most platforms, JDBC makes it possible to write a single database application that can run on different platforms and interact with different DBMSs.

JDBC was developed by ***JavaSoft***, a subsidiary of Sun Microsystems. JDBC 4.0 API is divided into 2 packages: **`java.sql`** and **`javax.sql`**; both packages are included in the Java SE and Java EE platforms.

The JDBC library includes APIs for each of the tasks commonly associated with database usage:

- Making a connection to a database
- Creating SQL statements
- Executing that SQL queries in the database
- Viewing & Modifying the resulting records

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 Server Pages (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.

List the Advantages of JDBC

- **Provide Existing Enterprise Data:** Businesses can continue to use their installed databases and access information even if it is stored on different database management systems.

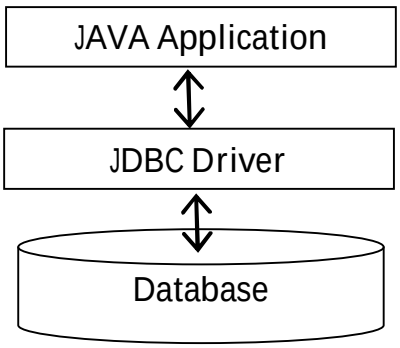
- **Simplified Enterprise Development:** The combination of the Java API and the JDBC API makes application development easy and cost effective.
- **Zero Configuration for Network Computers:** No configuration is required on the client side centralizes software maintenance. Driver is written in the Java, so all the information needed to make a connection is completely defined by the JDBC URL or by a Data Source object. Data Source object is registered with a Java Naming and Directory Interface (JNDI) naming service.
- **Full Access to Metadata:** The underlying facilities and capabilities of a specific database connection need to be understood. The JDBC API provides metadata access that enables the development of sophisticated applications.
- **No Installation:** A pure JDBC technology-based driver does not require special installation.
- **Database Connection Identified by URL:** The JDBC API includes a way to identify and connect to a data source, using a Data Source object. This makes code even more portable and easier to maintain.

ODBC (Open Database Connectivity)

ODBC is the Microsoft's solution for access databases. ODBC drivers are available for most of the major databases. The goal of ODBC is to make it possible to access any data from any application, regardless of which database management system (DBMS) is handling the data. ODBC manages this by inserting a middle layer, called a database driver, between an application and the DBMS. The purpose of this layer is to translate the application's data queries into commands that the DBMS understands.

JDBC Driver

JDBC is a general access method. It cannot directly process internal format of databases of different vendors of different vendors. To use the JDBC API with a particular DBMS, you need a JDBC technology-based driver to access database. Drivers for different databases are either provided by the vendors or by third parties provided by the vendors or by third parties.



JDBC drivers are client side adapters that convert requests from Java programs to a protocol that the DBMS can understand.

Type of JDBC Drivers

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 are explained below. The following are the different types of JDBC drivers that can be used with JDBC:

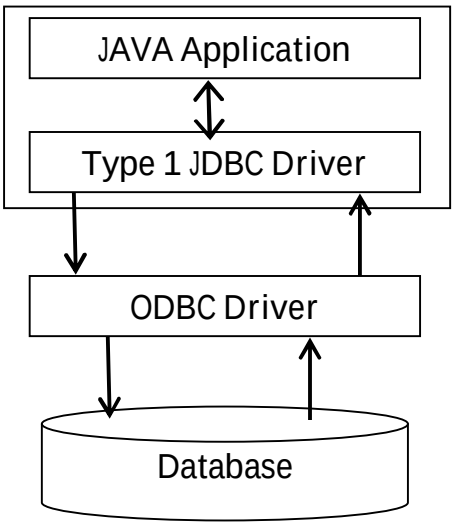
1. JDBC – ODBC Bridge Driver (Type 1)
2. Java Native – API Partly Java Driver (Type 2)
3. JDBC Net – Pure Java Driver (Type 3)
4. Native Protocol Pure Java Driver (Type 4)

Type 1: JDBC – ODBC Bridge Driver

- It allows Java programs to use JDBC with any existing ODBC drivers. The Bridge is itself a JDBC driver defined in the class

```
sun.jdbc.odbc.JdbcOdbcDriver
```

- It provides a way to access less popular DBMS only when no JDBC driver is available.
- Development of pure-Java JDBC drivers will make the JDBC-ODBC Bridge unnecessary.
- Allows you to access any ODBC data source using ODBC driver. JDBC driver accesses ODBC driver.
- Requires the ODBC driver to be present on the client.



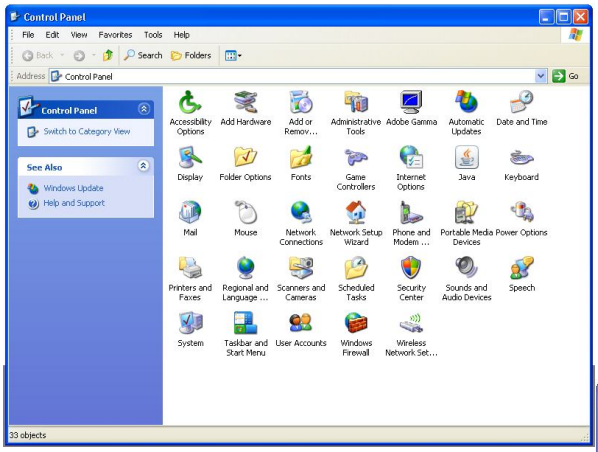
Using Type 1 Driver

In order to use ODBC driver, first we have to create DSN (Data Source Name) using ODBC Administrator and then use DSN in Connection URL.

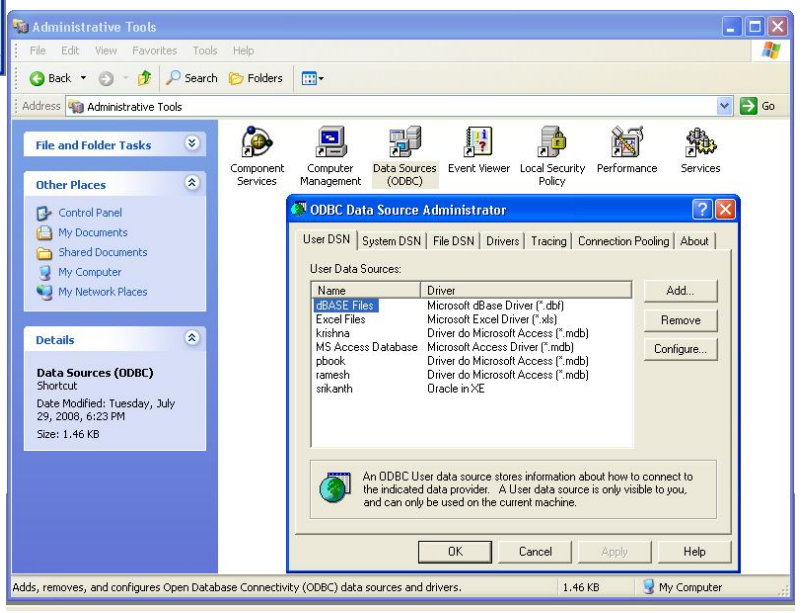
Steps to create DSN

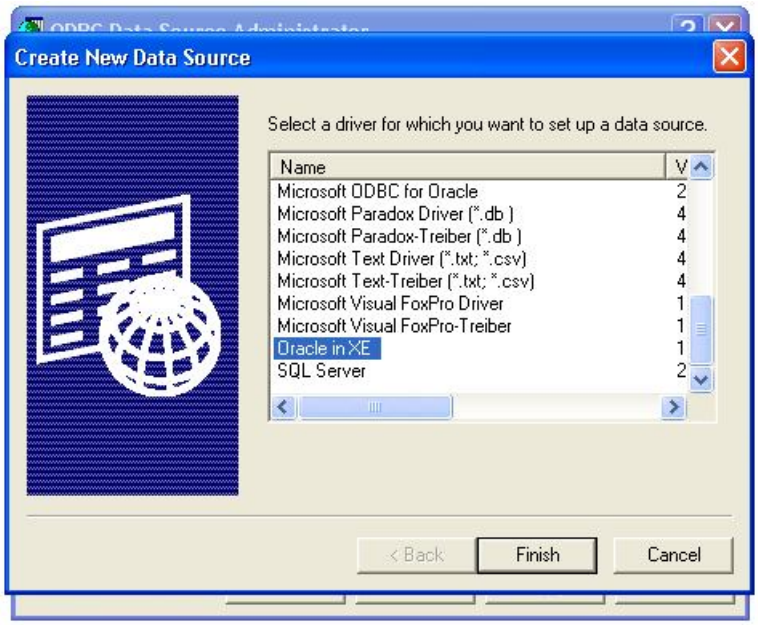
1. Invoke Control Panel
2. Select Performance and Maintenance
3. Select Administrative task
4. Select Data Sources (ODBC). It displays ODBC Data Source Administrator window.
5. Go to User DSN tab and click on **Add** button.
6. Select Oracle Driver from List of Drivers and click on **Finish** button.
7. In the next dialog box enter Data Source name – **somename (e.g. srikanth)** and other information such as username and password.
8. In case you access Oracle from a remote client then enter Oracle service name in Server field.
9. Click on OK button. Now the name given (**e.g. srikanth**) must appear in the list of DSNs.

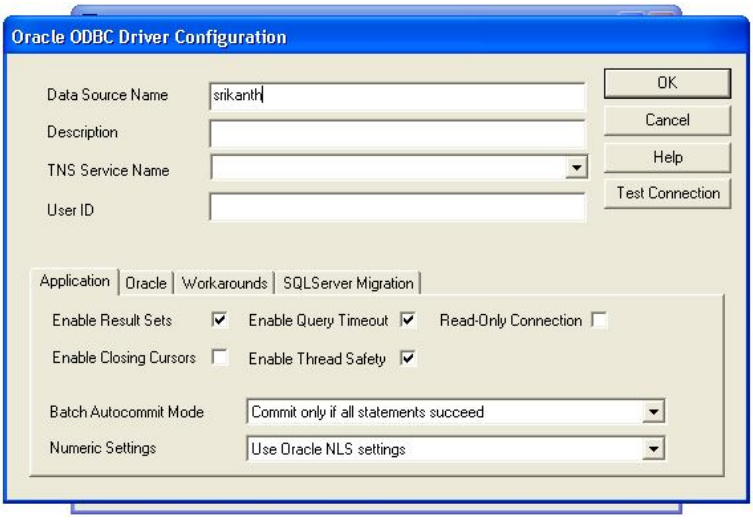
Steps to create DSN (Pictorial Representation)

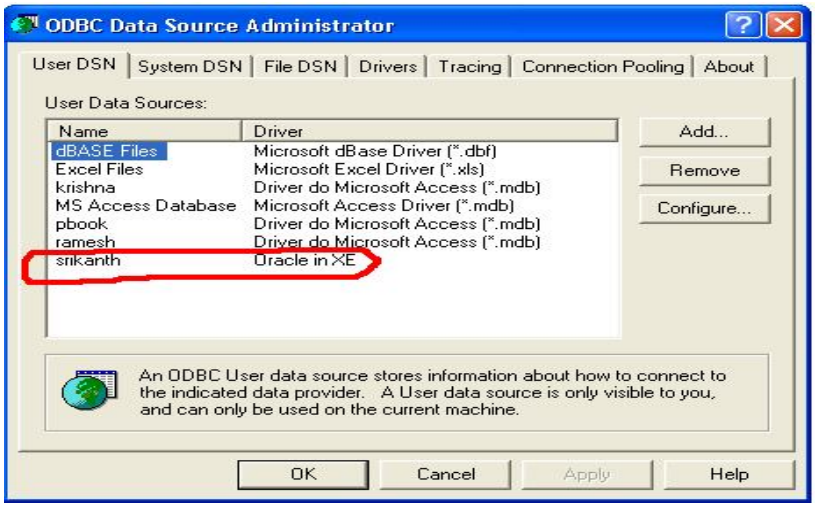


Click on Data Sources (ODBC)









Note: The above process lists and pictorial representation are steps in Windows XP. Change the steps regarding ODBC Administrator in case you use another operating system.

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.

Advantages:

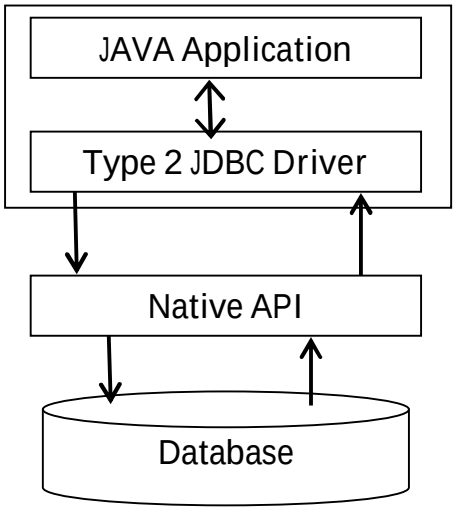
- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls. The ODBC driver needs to be installed on the client machine.

Type 2: Java Native – API Partly Java Driver

- Converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2 or other DBMS.
- JDBC API calls are converted into native C/C++ API calls which are unique to the database. These drivers 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.
- Requires that some binary code be loaded on each client machine.
- Oracle's OCI (Oracle Call Interface) driver is an example for this.
- 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.



Advantage:

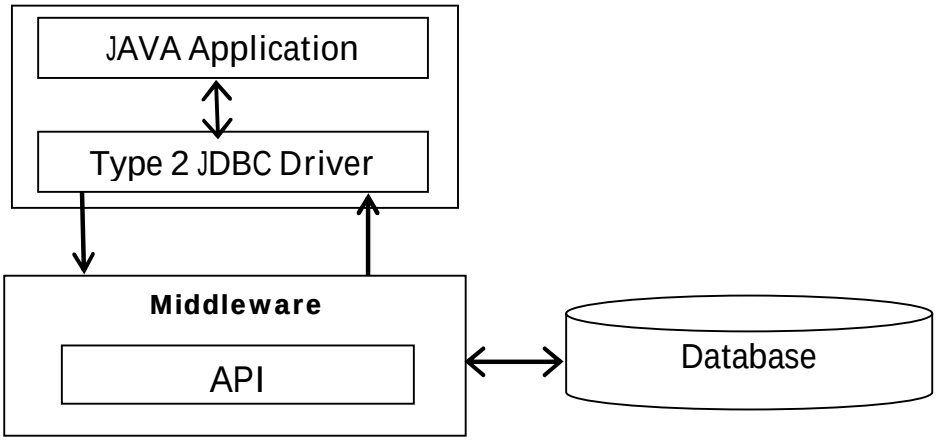
- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

Type 3: JDBC – Net Pure Java Driver

- Translates JDBC calls into a DBMS-independent net protocol, which is then translated to a DBMS protocol by a server.
- This driver is provided by middleware (application server) vendor.
- Clients need not have any binary code.
- In a Type 3 driver, a three-tier approach is used to accessing 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.



Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

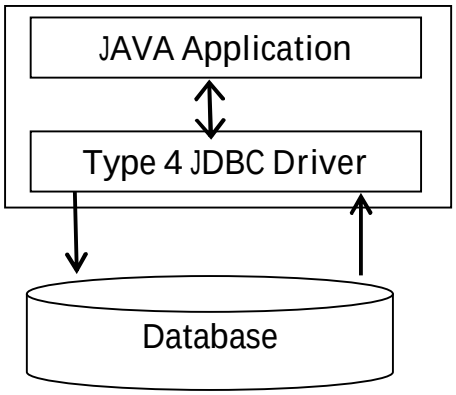
Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

Type 4: Native-Protocol pure Java driver

- This kind of driver converts JDBC calls into the network protocol used by DBMS directly.
- Oracle Thin driver is an example for this.
- 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.



- Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depend on the Database.

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.
- Eventually, driver categories 3 and 4 will be preferred way to access databases from JDBC.

JavaSoft Framework (or) JDBC Architecture

JavaSoft provides the following components as part of the JDK. The JDBC API supports both two-tier and three-tier processing models for database access but in general JDBC Architecture consists of two layers:

– **JDBC API: This provides the <u>application-to-JDBC Manager</u> connection.** The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

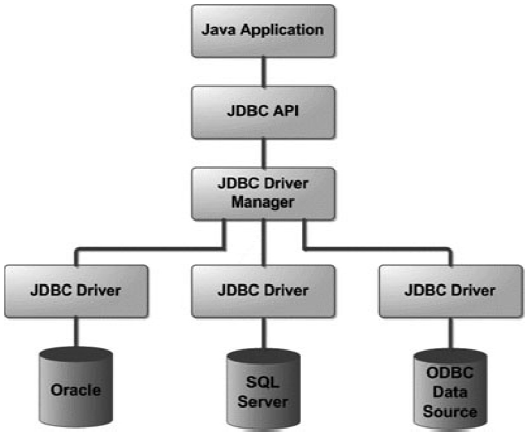
– **JDBC Driver API: This supports the <u>JDBC Manager-to-Driver</u> Connection.**
  - The JDBC driver manager ensures that the correct driver is used to access each data source.
  - The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

- **JDBC Driver Manager**

  The JDBC driver manager is the backbone of the JDBC architecture. It actually is quite small and simple. Its primary function is to connect Java applications to the correct JDBC driver and then get out of the way.

- **JDBC-ODBC Bridge**

  The JDBC-ODBC bridge allows ODBC drivers to be used as JDBC drivers.



Important Interfaces & Classes

The following are the interfaces provided with JDBC API. These interfaces are implemented by driver. A driver contains a collection of classes to implement these interfaces.

Interface	Meaning
CallableStatement	The interface used to execute SQL Stored Procedures
Connection	A connection (session) with a specific database
DatabaseMetaData	Comprehensive information about a database as a whole
Driver	The interface that every Driver class must implement
PreparedStatement	An object that represents a pre-compiled SQL statement
ResultSet	Provides access to a table of data
ResultSetMetaData	An object that can be used to find about the types and properties of the columns in a ResultSet
Statement	The object used for executing a static SQL statement and obtaining the results produced by it.

The following are the classes provided by JDBC API. These classes are provided in addition to interfaces mentioned above:

Class	Description
Date	A thin wrapper around a millisecond value that allows JDBC to identify this as a SQL DATE
DriverManager	The basic service for managing a set of JDBC Drivers
DriverPropertyInfo	Driver properties for making a connection
Time	A thin wrapper around java.util.Date that allows JDBC to identify this as a SQL TIME value
TimeStamp	A thinner class around java.util.Date that allows JDBC to identify this as a SQL TIMESTAMP value
Types	This class defines constants that are used to identify generic SQL types, called JDBC Types.

Example: Program to show how the Date and Time classes format standard Java Date and Time values to match the SQL Date Type Requirements.

```
import java.sql.Date;
import java.sql.Time;
import java.sql.Timestamp;
import java.util.*;
public class SqlDateTime {
   public static void main(String[] args) {
      //Get standard date and time
      java.util.Date javaDate = new java.util.Date();
      long javaTime = javaDate.getTime();
      System.out.println("Java Date is: " + javaDate.toString());
      //Get and display SQL DATE
      java.sql.Date sqlDate = new java.sql.Date(javaTime);
      System.out.println("The SQL DATE is: " + sqlDate.toString());
      //Get and display SQL TIME
      java.sql.Time sqlTime = new java.sql.Time(javaTime);
      System.out.println("The SQL TIME is: " + sqlTime.toString());
      //Get and display SQL TIMESTAMP
      java.sql.Timestamp sqlTs = new java.sql.Timestamp(javaTime);
      System.out.println("The SQL TIMESTAMP is: " + sqlTs.toString());
     }//end main
}//end SqlDateTime
```

Output:

```
Java Date is: Tue Aug 18 13:46:02 GMT+04:00 2009
The SQL DATE is: 2009-08-18
The SQL TIME is: 13:46:02
The SQL TIMESTAMP is: 2009-08-18 13:46:02.828
```

DriverManager Class

- Part of **`java.sql`** package, used to manage JDBC drivers.
- Sits between the application programs and the drivers.
- Keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver.

Methods	Meaning
Connection getConnection(String url, String un, String pwd)	Establishes a connection with the specified database.
Driver getDriver(String url)	Returns a driver that can understand the URL.
Enumeration getDrivers()	Returns the drivers that are currently loaded.
void registerDriver (Driver driver)	Registers the given driver

Driver Interface

- This is the interface that every driver has to implement.
- Each driver should supply a class that implements the Driver interface.

Methods	Meaning
boolean acceptsURL(String url)	Returns true if the driver thinks that it can open a connection to the given URL.
Connection connect(String url, Properties info)	Connects to a database and returns connection object
int getMajorVersion()	Returns major version number
Int getMinorVersion()	Returns minor version number

Connection Interface

- Represents a connection to specific database
- SQL statements are executed and results are returned within the context of a connection.

Methods	Meaning
Statement createStatement()	Creates and returns an object of Statement
Statement createStatement(int resultSetType, int resultSetConcurrency)	Creates a Statement object that will generate ResultSet objects with the given type and concurrency
DatabaseMetaData getMetaData()	Returns an object of DatabaseMetaData, which can be used to get information about the database.
boolean isClosed()	Returns true of connection is closed
CallableStatement prepareCall(String sql)	Creates a Callable statement
PreparedStatement prepareStatement(String sql)	Creates a prepared statement

void close()	Closes the connection

Getting Started with Oracle 10g Express Edition

Oracle 10g express edition provides HR account by default. But this account is locked by default. So you must log in as SYSTEM (DBA) to unlock this account.

Steps to unlock HR Account

1. Open Run SQL Command Line application
2. Use connect command at SQL> prompt to connect to Oracle using **system** account.
   ```
   SQL> connect system
   ```
3. When prompted to enter password, enter password that you gave at the time installing Oracle.
4. Unlock HR account and reset password by giving following two commands at the SQL> prompt.
   ```
   SQL>alter user hr account unlock;
   SQL>alter user hr identified by hr;
   ```
5. Then connect Oracle as HR by using CONNECT command as follows:
   ```
   SQL>connect hr/hr
   SQL>select * from tab;
   ...
   ```

Getting Started with NetBeans

- NetBeans is an IDE (Integrated Development Environment) to build simple to enterprise applications using Java. It runs on Windows, Linux, Mac OS X and Solaris.
- You can download NetBeans IDE from **www.netbeans.org**
- NetBeans IDE is open-source and free.

Steps for Creating a new project in NetBeans:

1. Start NetBeans IDE
2. Select **File -> New Project**
3. Select **Java** in **Categories**
4. Select **Java Application** in **Projects**
5. Click on Next
6. Enter **Project Name**. For example: **jdbcdemo**
7. Select **Project Location**, which is the folder where project is to be created.
8. Enter class name for the class to be created in the project with main() function in textbox after Create Main Class. By default it provides projectname.Main, where projectname is the package and Main is the class name. Erase and enter new name.
9. Click on Finish button.
10. A new project is created with **NewName.java**, which contains **main()** function.

The NetBeans Project **jdbcdemo** contains

- ***Source Packages***: Contains classes in different packages. Node, **default package**, contains classes that are placed in default package.
- ***Test Packages***
- ***Libraries***: node contains libraries (.jar files) related to the project. You can add new libraries like **ojdbc6.jar** using right click on libraries node and select **Add Library** or **Add Jar/Folder** option.
- ***Test Libraries***

Note: You can add .jar file containing JDBC driver for Oracle by adding **ojdbc6.jar** from the following location

ORACLE_HOME\app\oracle\product\10.2.0\server\jdbc\lib\ojdbc6.jar

ORACLE_HOME is the folder where oracle is installed.

Steps to connect to Oracle using OCI Driver (i.e. Type 2)

- Load jdbc driver for Oracle using Class.forName() method
- Include ojdbc6.jar file in classpath of the application. In case of NetBeans IDE, add .jar file to the libraries of the project.
- Establish a connection to database by using DriverManager.getConnection() with required connection string, username and password.

Example: Program to connect to oracle using OCI driver

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class OracleOCIConnection {
    public static void main(String args[]) {
        try {
            //load oracle driver
            Class.forName("oracle.jdbc.OracleDriver");
            //connect using Native-API (OCI) driver
            Connection con = DriverManager.getConnection
                      ("jdbc:oracle:oci8:@", "hr", "hr");
            System.out.println("Connected to Oracle");
            con.close();
        } catch (ClassNotFoundException | SQLException ex) {
            System.out.println(ex);
        }
    }
}
```

Using Oracle's Thin driver

- Thin driver provided by Oracle belongs to Type 4 category of JDBC drivers.
- This accesses oracle's TNS listener on the server.

- Connection string contains host. Port number of listener. Oracle instance name, username and password.

Example: Program to connect to oracle using Oracle Thin driver

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class OracleThinConnection {
    public static void main(String args[]) {
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con=DriverManager.getConnection
                ("jdbc:oracle:thin:@Localhost:1521:XE", "hr", "hr");
            System.out.println("Connected to Oracle");
            con.close();
        } catch (ClassNotFoundException | SQLException ex) {
            System.out.println(ex);
        }
    }
}
```

Statement Interface

- It is used to execute an SQL command.
- At the time of creating **Statement** we have to specify what type of **ResultSet** is to be created from this **Statement**.
- Only one ResultSet per a statement can be opened at a time.

Methods	Meaning
ResultSet executeQuery(String)	Executes the query in the statement and returns ResultSet.
int executeUpdate(String)	Execute DML command and returns no. of rows updated.
Boolean execute(String)	Execute SQL command and returns true if query is executed.
Connection getConnection()	Returns the connection object that produced this statement.
ResultSet getResultSet()	Returns current ResultSet.
int getUpdateCount()	Return update count of most recently executed command.
void close()	Enables the resources to be released.

ResultSet Interface

- Provides access to a set of rows.
- Contains a cursor that points to a particular record in the ResultSet. This cursor is positioned initially before the first row.
- You can retrieve values of columns using get<type>() methods. Columns can be referred either by column number or name. Columns are numbered from 1.

- A ResultSet may be either *scrollable* or *forward* only. It may be either *updatable* or *readonly*.

The following are the constant declared in ResultSet interface that are used at the time of creating Statement to specify the type of ResultSet required:

Type of ResultSet:

If you don't specify any ResultSet type, you will get TYPE_FORWARD_ONLY.

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forwards and backwards, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE.	The cursor can scroll forwards and backwards, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

Concurrency of ResultSet:

If you don't specify any Concurrency type, you will get CONCUR_READ_ONLY.

Concurrency	Description
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set. This is the default
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.

Methods	Meaning
type get<type>(int index)	Returns the value of the column identified by index
type get<type>(String name)	Returns the value of the column identified by name
ResultSetMetaData getMetaData()	Returns metadata of the ResultSet
boolean next()	Moves cursor to next row and returns true if next row is valid
boolean wasNull()	Returns true when the last columns read was null
void close()	Closes ResultSet
boolean absolute(int row)	Moves the cursor to the given row
void cancelRowUpdates()	Cancels the updates made to a row
int findColumn(columnName)	Maps the given ResultSet column name to its ResultSet column index
int getRow()	Returns current row number. Row numbers starts with 1
boolean first()	Moves to first record. Returns true on success.
boolean last()	Moves to the last record. Return true on success.
boolean previous()	Moves to the previous record.
void beforeFirst()	Moves to the immediately before first record

void afterLast()	Moves to the immediately after the last record
boolean relative(int)	Moves forward or backward by specified number of rows
boolean isFirst()	Returns true if cursor is on the first record.
boolean isLast()	Returns true if cursor is on the last record.
boolean isBeforeFirst()	Returns true if the cursor is before the first record.
boolean isAfterLast()	Returns true if the cursor is after the last record.
void refreshRow()	Refreshes current row with its most recent values in the database
void update<type>(int value)	Changes the existing data
void cancelRowUpdates()	Cancels the updates made to a row
void moveToInsertRow()	Create a new blank row
void insertRow()	Appends the new row to the table
void deleteRow()	Deletes the current row of the resultset from the table
void updateRow()	Updates the database with new contents of the current row

Steps to execute an SQL command

- Establish a connection to database.
- Create a statement using createStatement() method of Connection interface.
- Execute an SQL statement using one of the methods of Statement interface.

Example: Program to display the data in Jobs table and update salary for employee number 100.

```
import java.sql.*;
public class ExecuteUpdate {
    public static void main(String args[]) {
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection
                ("jdbc:oracle:thin:@Localhost:1521:XE", "hr", "hr");
            Statement st = con.createStatement();
            ResultSet rs = st.executeQuery("select * from jobs");
            while(rs.next()) {
                System.out.println(rs.getString(1) + " | "
                        + rs.getString(2) + " | " + rs.getString(3)
                        + " | " + rs.getString(4));           }
            st.close();
            Statement st1 = con.createStatement();
            int count = st1.executeUpdate("update employees
                set salary=salary*1.1 where employee_id=100");
            if(count==1)
                System.out.println("Updation is successful");
            else
                System.out.println("Updation is unsuccessful");
```

```
            st1.close();
            con.close();
        }
        catch(Exception e) {
            System.err.println(e);
        }
    }
}
```

Modifying Rows in ResultSet

In order to update a ResultSet, first the Statement object is to be created as follows:

```
Statement st = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                   ResultSet.CONCUR_UPDATABLE);
```

First parameter specifies scrollable nature of the ResultSet. The second parameter of createStatement() specifies whether ResultSet created from this Statement is updatable or read-only.

Example: Program to update a row; insert a row using ResultSet Type.

```
import java.sql.*;
public class TestScroll {
    public static void main(String args[]) {
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection
                ("jdbc:oracle:thin:@Localhost:1521:XE", "hr", "hr");
            Statement st = con.createStatement
                      (ResultSet.TYPE_SCROLL_INSENSITIVE,
                       ResultSet.CONCUR_UPDATABLE);
            ResultSet rs = st.executeQuery("select job_id,
                                        job_title from jobs");
            //goto last row
            rs.last();
            System.out.println(rs.getString(1));
            //update title
            rs.absolute(1); //goto first row
            rs.updateString(2, "New Job Title");
            rs.updateRow();
            //data change updated in database table row
            System.out.println(rs.getString(2));
            //insert new row
            rs.moveToInsertRow(); //blank row in resultset is created
            rs.updateString(1,"JP");
            rs.updateString(2, "Java Programmer");
```

```
            rs.insertRow();
            System.out.println(rs.getString(1)+" : "+rs.getString(2));
            //goto row
            rs.moveToCurrentRow();
            System.out.println(rs.getString(1));
            rs.close();
            st.close();
            con.close();
        }
        catch(Exception e) { System.out.println(e); }
    }
}
```

Output

```
PR_REP
New Job Title
JP : Java Programmer
AD_PRES
```

PreparedStatement Interface

- Contains a precompiled SQL statement. Command can be executed efficiently for multiple times.
- Extends Statement interface. Obtained using prepareStatement() method of Connection interface.

Methods	Meaning
void clearParameter()	Clears values of all parameters
void set<type>(int,value)	Sets the given value to the parameter at the specified index

Example: Program to update name of an employee using PreparedStatement

```
import java.sql.*;
public class ChangeName {
    public static void main(String args[]) {
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection
                ("jdbc:oracle:thin:@Localhost:1521:XE", "hr", "hr");
            PreparedStatement ps = con.prepareStatement(
             "update employees set first_name=? where employee_id=?");
            ps.setString(1, "SMITH");
            ps.setInt(2, 100);
            int count = ps.executeUpdate();
            if(count == 1)
                System.out.println("Updation is successful");
            else
                System.out.println("Updation is unsuccessful");
```

```
            ps.close(); con.close();
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

ResultSetMetaData Interface

- It contains the properties of columns in the ResultSet.

Method	Meaning
int getColumnCount()	Returns the number of columns in the ResultSet
String getColumnName(int)	Returns the name of the column
int getColumnType(int)	Returns column's SQL type.
String getTableName(int)	Returns the table name for the given column
boolean isNullable(int)	Returns true if column can contain null
int getColumnDisplaySize(int)	Indicates the column's normal max width in chars
String getColumnTypeName(int)	Retrieves a column's database-specific type name
int getPrecision(int)	Gets a column's number of decimal digits
int getScale(int column)	Gets a column's number of digits to right of the decimal point.

Example: Program to use the properties of columns in the ResultSet

```
import java.sql.*;
import java.util.Scanner;
public class SQLPlus {
    public static void main(String args[]) {
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection
                ("jdbc:oracle:thin:@Localhost:1521:XE", "hr", "hr");
            Statement st = con.createStatement();
            Scanner s = new Scanner(System.in);
            String qry;
            while(true) {
                System.out.println("Enter query: ");
                qry = s.nextLine();
                if(qry.equals("end")) break;
                ResultSet rs = st.executeQuery(qry);
                ResultSetMetaData rsmd = rs.getMetaData();
                int nc = rsmd.getColumnCount();
                for(int i=1;i<=nc;i++) {
                    System.out.print(rsmd.getColumnName(i)+"  ");
                } System.out.println();
```

```
                String data;
                while(rs.next()) {
                    data="";
                    for(int i=1;i<=nc;i++){
                        if(i!=1) data = data + ", ";
                        data = data + rs.getString(i);
                    }
                    System.out.println(data);
                } rs.close();
            } st.close(); con.close();
        }
        catch(Exception e) { System.err.println(e.getMessage()); }
    }
}
```

Output:

```
Enter query: select * from jobs where min_salary > 5000
JOB_ID  JOB_TITLE  MIN_SALARY  MAX_SALARY
AD_PRES, New Job Title, 20000, 40000
AD_VP, Administration Vice President, 15000, 30000
FI_MGR, Finance Manager, 8200, 16000
AC_MGR, Accounting Manager, 8200, 16000
SA_MAN, Sales Manager, 10000, 20000
SA_REP, Sales Representative, 6000, 12000
PU_MAN, Purchasing Manager, 8000, 15000
ST_MAN, Stock Manager, 5500, 8500
MK_MAN, Marketing Manager, 9000, 15000
Enter query:
End
```

DatabaseMetaData Interface

It provides information about the database. We can find out the features supported by database and object available in database.

Method	Meaning
String getDatabaseProductName()	Returns the name of this database product.
String getDatabaseProductVersion()	Returns the version of this database product.
ResultSet getSchemas()	Returns the list of schemas of the database.
ResultSet getTables(String catalog, String schema, String table, String tabletype[])	Returns the list of tables that match the criteria.
ResultSet getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)	Gets a description of table columns available in the specified catalog.

Example: Program to display information about the database

```
import java.sql.*;
public class DBMetaData {
    public static void main(String args[]) {
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection
                ("jdbc:oracle:thin:@Localhost:1521:XE", "hr", "hr");
            DatabaseMetaData dbmd = con.getMetaData();
            System.out.println("Product Name: " +
                 dbmd.getDatabaseProductName());
            System.out.println("Product Name: " +
                 dbmd.getDatabaseProductVersion());
            System.out.println("List of users of the database: ");
            ResultSet s = dbmd.getSchemas();
            while(s.next())
                System.out.println(" " + s.getString(1));
            System.out.println("List of Table of user HR: ");
            ResultSet tl = dbmd.getTables(null,"HR","%",null);
            while(tl.next()) {
                System.out.println(" " + tl.getString("TABLE_NAME"));
            }
        }
        catch(Exception e) { System.out.println(e); }
    }
}
```

Output:

```
Product Name: Oracle
Product Name: Oracle Database 10g Express Edition Release 10.2.0.1.0 -
Production
List of users of the database:
 ANONYMOUS
 CTXSYS
 DBSNMP
 DIP
 FLOWS_020100
 FLOWS_FILES
 HR
 MDSYS
 OUTLN
 SYS
 SYSTEM
 TSMSYS
```

```
 XDB
List of Table of user HR:
 COUNTRIES
 DEPARTMENTS
 EMPLOYEES
 JOBS
 JOB_HISTORY
 LOCATIONS
 REGIONS
 EMP_DETAILS_VIEW
```

Transaction Management

- A transaction consists of one or more statements that have been executed, completed and then either committed or rolled back. When the method `commit` or `rollback` is called, the current transaction ends and another one begins.
- A new connection is in auto-commit mode by default, meaning that when a statement is completed, the method `commit` will be called on that statement automatically.

The following methods of Connection interface are used to manage transaction:

Method	Meaning
void setAutoCommit(Boolean)	Turns on/off auto commit mode.
Boolean getAutoCommit()	Returns the current auto commit mode.
void commit()	Makes all charges since previous commit point permanent
void rollback()	Drops all changes made since the previous commit/rollback
void rollback(Savepoint)	Rolls back changes made from the given savepoint
Savepoint setSavepoint(name)	Sets a savepoint in the current transaction.
void releaseSavepoint(sp)	Releases savepoint from transaction.

```
con.setAutoCommit(false);
try {
    st.executeUpdate(cmd1);
    st.executeUpdate(cmd2);
    con.commit();
}
catch(Exception e) {
    con.rollback();
}
```

Savepoint

- Savepoint is a point within a transaction to which you can roll back. Connection interface provides setSavepoint() method to set savepoint.

- Method releaseSavepoint(savepoint) releases savepoint from the current transaction.

```
Statement stmt = conc.createStatement();
int rows = stmt.executeUpate(sqlcommand);
Savepoint savepoint1 = conn.setSavepoint("SAVEPOINT");
rows = stmt.executeUpdate(sqlcommand);
...
if(rows>5)
   con.rollback(savepoint1);
...
con.commit();
```

Batch Updates

- It allows multiple DML statements to be executed in a single request.
- Reduces overhead on database server as it has to execute only one command.
- Improves performance when large number of statements is executed.

Example: Program to update employee's salaries with multiple statements

```
import java.sql.*;
public class BatchUpdateDemo {
    public static void main(String args[]) {
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection
                ("jdbc:oracle:thin:@Localhost:1521:XE", "hr", "hr");
            con.setAutoCommit(false);
            Statement stmt = con.createStatement();
            try {
                stmt.addBatch("update employees set
                      salary = salary + 2000 where salary > 10000");
                stmt.addBatch("update employees set
                      salary = salary + 2000 where salary < 10000");
                int[] uc = stmt.executeBatch(); //execute batch
                //no of rows affected by each command
                con.commit();
                System.out.println("Update Row Count");
                for(int i=0;i<uc.length;i++) {
                    System.out.println(i + ":" + uc[i]);
                }
            }
            catch(BatchUpdateException e) {
                System.out.println("Batch Update Exception : "
```

```
                                    + e.getMessage());
                    con.rollback();
                }
            }
            catch(Exception e) {
                System.out.println(e);
            }
        }
}
```

Output:

```
Update Row Count
0:15
1:88
```

CallableStatement Interface

- Used to execute a stored procedure or function.
- It extends **PreparedStatement** interface.

To call a function: {?=call <function-name>[<arg1>,<arg2>,...]} To call a procedure: {call <procedure-name>[<arg1>,<arg2>,...]}

Methods	Meaning
type get<type>(int index)	Returns the value of the given parameter
void registerOutParameter(int index, int sqltype)	Registers an OUT parameter at the specified parameter index to the SQLType.
boolean wasNull()	Returns true if last OUT parameter read has SQL NULL value.

UPDATESALARY procedure

```
create or replace procedure UpdateSalary(p_empid number, p_sal number) is
begin
      update employees set salary = p_sal where employee_id = p_empid;
      commit;
end;
/
```

Example: Program to update employee salary using CallableStatement

```
import java.sql.*;
public class CSExample {
    public static void main(String args[]) {
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection
                ("jdbc:oracle:thin:@Localhost:1521:XE", "hr", "hr");
            CallableStatement cs = con.prepareCall
                ("{call UpdateSalary(?,?)}");
```

```
            cs.setInt(1, 100);
            cs.setInt(2, 6000);
            boolean b = cs.execute();
            if(b==false)
                System.out.println("Update Successful");
            else
                System.out.println("Update unsuccessful");
            cs.close();
            con.close();
        }
        catch(Exception e) {
            e.printStackTrace(); }
    }
}
```

Explain the process of establishing database connection using JDBC

After we've installed the appropriate driver, it's time to establish a database connection using JDBC. The programming involved to establish a JDBC connection is fairly simple. Here are these simple four steps:

- **Import JDBC Packages:** Add **import** statements to your Java program to import required classes in your Java code.
- **Register JDBC Driver:** This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.
- **Database URL Formulation:** This is to create a properly formatted address that points to the database to which you wish to connect.
- **Create Connection Object:** Finally, code a call to the *DriverManager* object's *getConnection( )* method to establish actual database connection.

1) Import JDBC Packages:

The **Import** statements tell the Java compiler where to find the classes you reference in your code and are placed at the very beginning of your source code. To use the standard JDBC package, which allows you to select, insert, update, and delete data in SQL tables, add the following *imports* to your source code:

```
import java.sql.* ;  // for standard JDBC programs
import java.math.* ; // for BigDecimal and BigInteger support
```

2) Register JDBC Driver:

You must register the driver in your program before you use it. Registering the driver is the process by which the Oracle driver's class file is loaded into memory so it can be utilized as an implementation of the JDBC interfaces. You need to do this registration only once in your program. You can register a driver in one of two ways.

Approach 1 - Class.forName():

The most common approach to register a driver is to use Java's **Class.forName()** method to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

The following example uses Class.forName( ) to register the Oracle driver:

```
try {
   Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
}
```

Approach 2 - DriverManager.registerDriver():

The second approach you can use to register a driver is to use the static **DriverManager.registerDriver()** method.

You should use the *registerDriver()* method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

The following example uses registerDriver() to register the Oracle driver:

```
try {
   Driver myDriver = new oracle.jdbc.driver.OracleDriver();
   DriverManager.registerDriver( myDriver );
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
}
```

3) Database URL Formulation:

After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method. For easy reference, let me list the three overloaded DriverManager.getConnection() methods:

- getConnection(String url)
- getConnection(String url, Properties prop)
- getConnection(String url, String user, String password)

Here each form requires a database **URL**. A database URL is an address that points to your database. Formulating a database URL is where most of the problems associated with establishing a connection occur.

Following table lists down popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	**jdbc:mysql:**//hostname/databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:@**hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	**jdbc:db2:**hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	**jdbc:sybase:Tds:**hostname:port Number/databaseName

All the highlighted part in URL format is static and you need to change only remaining part as per your database setup.

4) Create Connection Object:

Using a database URL with a username and password:

I listed down three forms of **DriverManager.getConnection()** method to create a connection object. The most commonly used form of getConnection() requires you to pass a database URL, a *username*, and a*password*:

1) Assuming you are using Oracle's **thin** driver, you'll specify a host:port:databaseName value for the database portion of the URL.

Ex : If you have a host at TCP/IP address 192.0.0.1 with a host name of amrood, and your Oracle listener is configured to listen on port 1521, and your database name is EMP, then complete database URL would then be:

jdbc:oracle:thin:@amrood:1521:EMP

Now you have to call getConnection() method with appropriate username and password to get a **Connection** object as follows:

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
String USER = "username";
String PASS = "password"
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

2) Using only a database URL:

A second form of the DriverManager.getConnection( ) method requires only a database URL: DriverManager.getConnection(String url);

However, in this case, the database URL includes the username and password and has the following general form:

jdbc:oracle:driver:username/password@database

So the above connection can be created as follows:

```
String URL = "jdbc:oracle:thin:username/password@amrood:1521:EMP";
Connection conn = DriverManager.getConnection(URL);
```

3) Using a database URL and a Properties object:

A third form of the DriverManager.getConnection( ) method requires a database URL and a Properties object:

DriverManager.getConnection(String url, Properties info);

A Properties object holds a set of keyword-value pairs. It's used to pass driver properties to the driver during a call to the getConnection() method.

To make the same connection made by the previous examples, use the following code:

```
import java.util.*;
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
Properties info = new Properties( );
info.put( "user", "username" );
info.put( "password", "password" );
Connection conn = DriverManager.getConnection(URL, info);
```

Closing JDBC connections:

At the end of your JDBC program, it is required explicitly close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

Relying on garbage collection, especially in database programming, is very poor programming practice. You should make a habit of always closing the connection with the close() method associated with connection object.

To ensure that a connection is closed, you could provide a finally block in your code. A *finally* block always executes, regardless if an exception occurs or not.

To close above opened connection you should call close() method as follows:

```
conn.close();
```

Explicitly closing a connection conserves DBMS resources, which will make your database administrator happy.

Example: Java Program to insert data into database using Statement

```
import java.sql.*;
public class ExecuteUpdate {
    public static void main(String args[]) {
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection
                ("jdbc:oracle:thin:@Localhost:1521:XE", "hr", "hr");
            Statement st = con.createStatement();
            int count = st1.executeUpdate("insert into jobs
                values('JP','JAVA PROGRAMMER','5000','10000')");
```

```
            if(count==1)
                System.out.println("Insertion is successful");
            else
                System.out.println("Insertion is unsuccessful");
            st1.close();
            con.close();
        }
        catch(Exception e) {
            System.err.println(e);
        }
    }
}
```

Example: Java Program to insert data into database using PreparedStatement

```
import java.sql.*;
public class ChangeName {
    public static void main(String args[]) {
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection
                ("jdbc:oracle:thin:@Localhost:1521:XE", "hr", "hr");
            PreparedStatement ps = con.prepareStatement(
             "insert into jobs values(?,?,?,?)");
            ps.setString(1, "WD");
            ps.setString(2, "WEB DEVELOPER");
            ps.setInt(3, 10000);
            ps.setInt(4, 15000);
            int count = ps.executeUpdate();
            if(count == 1)
                System.out.println("Insertion is successful");
            else
                System.out.println("Insertion is unsuccessful");
            ps.close(); con.close();
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

SQL to Java Data Types

TABLE 6.1 Converting SQL Types to Java

SQL type	*Java type*	*Notes*
CHAR	String	Fixed length string of characters
VARCHAR	String	Variable length string of characters
LONGVARCHAR	String java.io.InputStream	Any length (multibyte) strings
NUMERIC	Java.math.BigDecimal	Absolute precision decimal value
DECIMAL	Java.math.BigDecimal	Absolute precision decimal value
BIT	Boolean	Single bit/binary value (on or off)
TINYINT	Byte	8-bit integer
SMALLINT	Short	16-bit integer
INTEGER	Integer	Signed 32-bit integer
BIGINT	Long	Signed 64-bit integer
REAL	Float	Floating-point value
FLOAT	Float	Floating-point value
DOUBLE	Double	Large floating-point value
BINARY	Byte[]	Array of binary values
VARBINARY	Byte[]	Variable length array of binary values
LONGVARBINARY	Byte[]	Any length (multi-megabyte) array of binary values
DATE	Java.sql.Date	Date value
TIME	Java.sql.Time	Time value
TIMESTAMP	Java.sql.Timestamp	Time value with additional nanosecond field

TABLE 6.2 Converting Java Object Types to SQL Types

Java type	*SQL type*
String	VARCHAR LONGVARCHAR
Java.math.BigDecimal	NUMERIC
Boolean	BIT
Integer	INTEGER
Long	BIGINT
Float	FLOAT
Double	DOUBLE
Byte[ ]	VARBINARY LONGVARBINARY
Byte	TINYINT
Short	SMALLINT
Java.sql.Date	DATE
Java.sql.Time	TIME
Java.sql.Timestamp	TIMESTAMP

JDBC Exception Types

As a matter of course, you should try to handle everything that may go wrong at execution time so that the user experiences a soft and user-friendly failure rather than a hard failure. JDBC provides many exception objects to facilitate a soft failure.

JDBC provides these types of exceptions:

- `SQLException`
- `SQLWarning`
- `DataTruncation`

SQLException

Most of the methods in the `java.sql` package throw `SQLException` to indicate failure, which requires a try/catch block.

The `SQLException` has methods to provide developers with more detailed information about the errors. To track many problems that occur, you may use `getNextException()`. This method will either retrieve the next `SQLException` or return null if there are no more exceptions.

The following example illustrates the use of this method:

```
try {
    ....
}
catch(SQLException sqle) {
    while(sqle != null )
    {
        System.err.print("SQLException: " ) ;
        System.println(sqle.toString());
        sqle = sqle.getNextException();
    }
}
```

Additionally, to get the vendor-specific exception error code, you may use the `getErrorCode()` method on `SQLException`.

SQLWarning

`SQLWarning` is a subclass of `SQLException`. You can get a list of all the warnings using the `getWarning()` method on the connection:

JDBC 2.1 New Data Types

The next version of the ANSI/ISO SQL standard defines some new data types, commonly referred to as the SQL3 types. The new SQL3 data types give a relational database more flexibility in determining what can be used as a type for each table column.

The primary new SQL3 types are

- `BLOB`—Binary Large Objects. Mapped to the `java.sql.Blob` type.
- `CLOB`—Character Large Objects. Mapped to the `java.sql.Clob` type.
- `ARRAY`—Can store values of a specified type. Mapped to the `java.sql.Array` type.
- `STRUCT`—This is the default mapping for any SQL structured type, and is mapped to the `java.sql.Struct` type.
- `REF`—Serves as a reference to SQL data within the database. Can be passed as a parameter to an SQL statement. Mapped to the `java.sql.Ref` type.

BLOB and CLOB

The `java.sql.Blob` and `java.sql.Clob` interfaces give you the ability to load only a portion of the column's value into memory. Methods in the interfaces `ResultSet`, `PreparedStatement`, and `CallableStatement`, such as `getBlob()` and `setBlob()`, allow a programmer to access the SQL `BLOB`. The `BLOB` contains a logical pointer to the SQL `BLOB` data rather than the data itself. Methods such as `getBlob()` and `setBlob()` return a pointer to the value. Using such a pointer, your application can read some or all of the data as needed.

For instance, assume that we have a table called `APPLICANT` that contains a column of type `VARCHAR` called `RESUME`. If we execute the following statement, it's straightforward to get the data stored in `RESUME` by calling the `getString()` method:

```
try {
...
String sResume = null;
String sSQL = "SELECT RESUME FROM APPLICANT "+
" WHERE APPLICANT_ID  = ? ";
PreparedStatement pstmt = con.prepareStatement(sSQL);
pstmt.setInt(1, 100); // Assume the APPLICANT_ID = 100
ResultSet rs = pstmt.executeQuery();
if( rs.next() ) {
        // Process the resultset record
        sResume = rs.getString("RESUME");
}
rs.close();
pstmt.close();
```

```
}
catch(SQLException sqle) {
        System.err.println("SQLException: " + sqle.getMessage());
}
```

After executing this code, the entire column value is loaded into memory.

Let's modify the code to load a portion of the column's value into memory by using the `java.sql.Clob` class. This will be useful for searching the `RESUME` column for `CLOBs`. It can also be used to search any other column for a binary sequence of `BLOBs`.

```
try {
...
String sResume = null;
String sSQL = "SELECT RESUME FROM APPLICANT "+
" WHERE APPLICANT_ID  = ? ";
PreparedStatement pstmt = con.prepareStatement(sSQL);
pstmt.setInt(1, 100); // Assume the APPLICANT_ID = 100
ResultSet rs = pstmt.executeQuery();
if( rs.next() ) {
        // Process the resultset record
        Clob clbResume = rs.getClob();
        long lResumeLen = clbResume.length();
        // Loop through the clbResume and process 50 characters:
        for ( long lLoop = 0 ; lLoop < lResumeLenl; lLoop +=50)
        {
                sResume = clbResume.getSubString(lLoop,50);
        // Do your search: if found, break the loop:
        ...
        }
}
rs.close();
pstmt.close();
}
catch(SQLException sqle) {
        System.err.println("SQLException: " + sqle.getMessage());
}
```

ARRAY

The new SQL data type `ARRAY` allows an array to be used as a column value in a database table.

Using an `Array` object is as easy as using a basic data type. An SQL `ARRAY` value is retrieved with the method `getArray()`.

For instance, assume we have a table called `APPLICANT` that contains a column called `LANGUAGE` and that it stores SQL `ARRAY` values.

```
String sSQL = "SELECT LANGUAGE FROM APPLICANT WHERE " +
" APPLICANT_ID=100";
ResultSet rs = stmt.executeQuery(sSQL);
rs.next();
Array arrLanguages = rs.getArray("LANGUAGE");
```

The variable `arrLanguages` now is a logical pointer to the `LANGUAGE` field for `APPLICANT_ID 100` that resides on the server. This means that you can operate on `arrLanguages` just as if you were operating on the SQL `ARRAY` object itself but without incurring the overhead of bringing all the data in the array over to the client.

STRUCT

The `STRUCT` interface is the standard mapping for an SQL structured type.

SQL structured types are similar to Java classes that contain public member variables only. They can be retrieved from `ResultSet` objects using the `getObject()` method.

REF

`REF` serves as a reference to SQL data within the database. Because the SQL `REF` value is a pointer and does not contain actual data, the `REF` object also is a pointer. An SQL `REF` can be retrieved by calling the `getRef()` method of the `ResultSet`.

For example:

```
...
String sSQL = "SELECT EMP_HIST FROM EMPLOYEE WHERE EMPL_ID = 300";
ResultSet rs = stmt.executeQuery(sSQL);
rs.next();
Ref ref = rs.getRef("EMP_HIST");
Experience exp = (Experience) ref.getObject();
...
```

JDBC 2.0 Optional Package API: `javax.sql`

The JDBC 2.0 Optional Package API adds new functionality:

- The `DataSource` interface—Used for working with JNDI, allowing a connection to be made using a `DataSource` object registered with a JNDI naming service.
- Connection Pooling—New layer for implementing connection pooling to reuse connections instead of creating new ones.

- Distributed transactions—Transactions that may be distributed across multiple database instances. They are usually managed using a technique called a two-phase commit. The first phase involves preparing each database involved in the transaction to make the appropriate changes and allows the database to indicate whether the changes would result in an error. If committing the changes would cause an error, the proposed changes are rolled back. Otherwise, in the second phase, the changes are committed in all of the participating databases.
- `RowSets`—JavaBeans-compliant objects that encapsulate database resultsets and the accessible information.

Database Access with JNDI

JDBC 2.0 introduced the `DataSource` interface and the JNDI technology for naming and location services.

JNDI can be used in addition to the JDBC driver manager to manage data sources and connections.

As mentioned before, you may obtain a connection using the `DriverManager` object; however, using a `DataSource` object is the preferred way because it has many advantages such as easier code maintenance, increased performance due to the use of connection pooling, and support for distributed transactions through `XADataSource`.

To create a database connection with JNDI, you specify a resource name, which corresponds to an entry in a database or naming service, and then receive back the information necessary to establish a connection with your database. JNDI provides a uniform way to find and access services on a network.

`DataSource` objects can be created and managed separately from the JDBC applications.

JDBC data sources are registered in the `jdbc` subcontext. For example:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("jdbc/demo");
Connection con = ds.getConnection("username", "password");
```

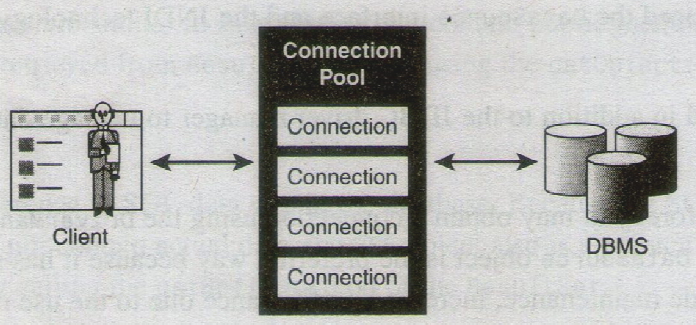
Connection Pooling

Connection pooling is defined as part of the JDBC 2.1 Standard Extension API. A connection pool is a set of database connections that are loaded into memory so that they can be reused. This allows sharing of database connections, instead of requiring a separate connection for each client.

Connecting to a database can be an expensive operation. Each time your application attempts to access a database, it must connect to that database. However, connecting to a database is a time-consuming activity since the database connection requires resources (communication, memory, authentication, and so on) to create each connection, maintain it, and then release it when it is no longer required. The overhead is particularly high for Web-based applications because Web users connect and disconnect more frequently. Usually, usage volumes can be large and difficult to predict. Establishing a connection once and then reusing it for subsequent requests can dramatically improve performance and response times of Web-based applications.

Most good Application Servers establish a pool of database connections that is shared by all the applications, as in Figure 6.7.

FIGURE 6.7
Connection pool.



The following is a JDBC Connection Pooling example:

```
//Get a handle to the JNDI context
Context ctx = new InitialContext();
//Get a reference to the connection pool
String dsName = "java:comp/env/jdbc/poolname";
DataSource ds = (DataSource) ctx.lookup(dsName);
//Get a connection
Connection con = ds.getConnection();
// Handle single transaction (optional)
con.setAutoCommit(false);
// Your Code goes here: Standard JDBC code SELECT/UPDATE/...)
...
// Commit or rollback the transaction:
con.commit();
// Close the connection (return the connection object
// back to the pool)
con.close();
```

Distributed Transactions

The JDBC 2.0 Specification provides the capability for handling distributed transactions. A distributed transaction is a single transaction that applies to multiple, heterogeneous databases that may reside on separate server machines.

For supporting distributed transactions, JDBC 2.0 provides two new interfaces: `javax.sql.XADataSource` and `javax.sql.XAConnection`.

JDBC RowSets

`RowSets` exist to provide the flexibility of a disconnected operation. JavaBean is a portable platform-independent component model written in Java and follows the JavaBean specification. It enables you to write reusable components. JDBC `RowSet` objects are JavaBeans capable of operating without an active database connection. This is useful for sending data across a network to thin clients, such as Web browsers, laptops, and PDAs.

Because a `RowSet` is a JavaBean, you can implement events for the `RowSet`, and you can set properties on the `RowSet`.

Here are some possible `RowSet` implementations:

- `CachedRowSet`: a `RowSet` that can be used to pass a set of rows across a network or to add scrollability and updateability to resultset data.
- `JDBCRowSet`: a `RowSet` that can be used to encapsulate a JDBC driver as a JavaBeans component.
- `WebRowSet`: a set of rows that are being cached outside of a data source. `WebRowSet` is an extension of `CachedRowSet`.