**JDBC**

What is JDBC? How do you connect to a database?

JDBC stands for **Java Database Connectivity**. It is an API which provides easy connection to a wide range of

databases. To connect to a database we need to load the appropriate driver and then request for a connection

object. The Class.forName(….) will load the driver and register it with the DriverManager (Refer **Q4** in Java section

for dynamic class loading).

Class.forName(“oracle.jdbc.driver.OracleDriver”);

String url = jdbc:oracle:thin:@hostname:1526:myDB;

Connection myConnection = **DriverManager**.getConnection(url, “username”, “password”);

The ***DataSource*** interface provides an alternative to the *DriverManager* for making a connection. *DataSource*

makes the code more portable than DriverManager because it work with JNDI and it is created, deployed and

managed separately from the application that uses it. If the DataSource location changes, then there is no need to

change the code but change the configuration properties in the server. This makes your application code easier to

maintain. *DataSource* allows the use of connection pooling and support for distributed transactions. A *DataSource*

is not only a database but also can be a file or a spreadsheet. A *DataSource* object can be bound to JNDI and an

application can retrieve and use it to make a connection to the database. J2EE application servers provide tools to

define your DataSource with a JNDI name. When the server starts it loads all the DataSources into the Application

Server’s JNDI service.

DataSource configuration properties are shown below:

􀂃 **JNDI Name** 􀃆 jdbc/myDataSource

􀂃 **URL** 􀃆 jdbc:oracle:thin:@hostname:1526:myDB

􀂃 **UserName**, **Password**

􀂃 **Implementation classname** 􀃆 oracle.jdbc.pool.OracleConnectionPoolDataSource

􀂃 **Classpath** 􀃆 ora\_jdbc.jar

􀂃 **Connection pooling** settings like 􀃆 minimum pool size, maximum pool size, connection timeout, statement cache size etc.

Once the *DataSource* has been set up, then you can get the connection object as follows:

Context ctx = new InitialContext();

DataSource ds = (DataSource)ctx.lookup("jdbc/myDataSource");

Connection myConnection = ds.getConnection(“username”,”password”);

In a basic implementation a *Connection* obtained from a *DataSource* and a *DriverManager* are identical. But,

**DataSource is recommended because of its better portability**.

**Design Pattern:** JDBC architecture decouples an abstraction from its implementation so that the implementation

can vary independent of the abstraction. This is an example of the **bridge design pattern**. The JDBC API

provides the abstraction and the JDBC drivers provide the implementation. New drivers can be plugged-in to the

JDBC API without changing the client code.

What are JDBC Statements? What are different types of statements? How can you create them?

A **statement** object is responsible for sending the SQL statements to the Database. Statement objects are created

from the connection object and then executed. **CO**

Statement stmt = myConnection.createStatement();

ResultSet rs = stmt.executeQuery(“SELECT id, name FROM myTable where id =1245”);// to read

**Or**

stmt.executeUpdate(“INSERT INTO (field1,field2) values (1,3)”);// to insert/update/delete/create table

The types of statements are:

􀂃 **Statement** (regular statement as shown above)

􀂃 **PreparedStatement** (more efficient than statement due to pre-compilation of SQL)

􀂃 **CallableStatement** (to call stored procedures on the database)

To use prepared statement:

PreparedStatement prepStmt =

myConnection.prepareStatement("SELECT id, name FROM myTable where id = **?** ");

prepStmt.setInt(1, 1245);

Callable statements are used for calling stored procedures.

CallableStatement calStmt = myConnection.prepareCall("{call PROC\_SHOWMYBOOKS}");

ResultSet rs = cs.executeQuery();

What is a **Transaction**? What does **setAutoCommit** do?

A transaction is a set of operations that should be completed as a unit. If one operation fails then all the other

operations fail as well. For example if you transfer funds between two accounts there will be two operations in the

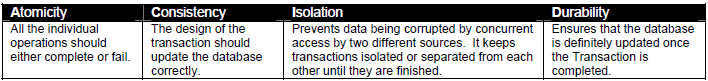
set

1. Withdraw money from one account.

2. Deposit money into other account.

These two operations should be completed as a single unit. Otherwise your money will get lost if the withdrawal is

successful and the deposit fails. There are four characteristics (**ACID** properties) for a Transaction.



Transactions maintain data integrity. A transaction has a beginning and an end like everything else in life. The

**setAutocommit(….)**, **commit()** and **rollback()** are used for marking the transactions (known as transaction

demarcation). When a connection is created, it is in **auto-commit** mode. This means that each individual SQL

statement is treated as a transaction and will be automatically committed immediately after it is executed. The way

to allow two or more statements to be grouped into a transaction is to **disable** auto-commit mode:

try{

Connection myConnection = dataSource.getConnection();

// set autoCommit to false

**myConnection .setAutoCommit(false);**

withdrawMoneyFromFirstAccount(.............); //*operation 1*

depositMoneyIntoSecondAccount(.............); //*operation 2*

**myConnection .commit();**

}

catch(Exception sqle){

try{

**myConnection .rollback();**

}catch( Exception e){}

}

**finally**{

try{if( conn != null) {conn.close();}} catch( Exception e) {}

}

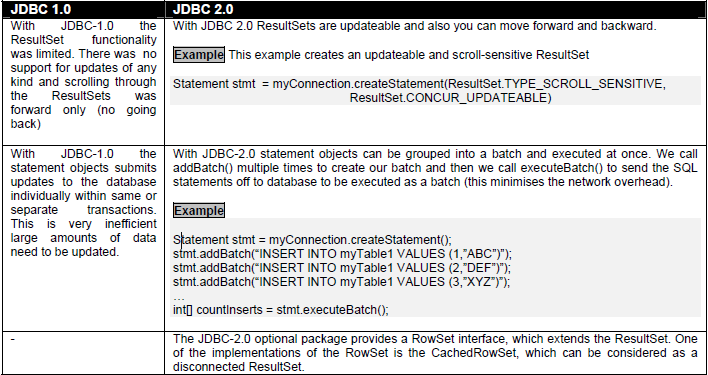
The above code ensures that both operation 1 and operation 2 succeed or fail as an atomic unit and consequently

leaves the database in a consistent state. Also turning auto-commit off will provide better performance.

What is the difference between JDBC-1.0 and JDBC-2.0? What are Scrollable ResultSets, Updateable ResultSets,

RowSets, and Batch updates?

JDBC2.0 has the following additional features or functionality:



What is the difference between statements and prepared statements?

􀂃 Prepared statements offer better performance, as they are **pre-compiled**. Prepared statements reuse the

same **execution plan** for different arguments rather than creating a new execution plan every time. Prepared

statements use bind arguments, which are sent to the database engine. This allows mapping different

requests with same prepared statement but different arguments to execute the same execution plan.

􀂃 Prepared statements are more secure because they use bind variables, which can prevent SQL injection

attack.

The most common type of SQL injection attack is SQL manipulation. The attacker attempts to modify the

SQL statement by adding elements to the WHERE clause or extending the SQL with the set operators like

UNION, INTERSECT etc.

***Example*** Let us look at the following SQL:

SELECT \* FROM users where username=’bob’ AND password=’xyfdsw’ ;

The attacker can manipulate the SQL as follows

SELECT \* FROM users where username=’bob’ AND password=’xyfdsw’ **OR ‘a’ = ‘a’ ;**

The above “WHERE” clause is always true because of the operator precedence. The PreparedStatement

can prevent this by using bind variables:

String strSQL = SELECT \* FROM users where username=? AND password=?);

PreparedStatement pstmt = myConnection.prepareStatement(strSQL);

pstmt.setString(1,”bob”);

pstmt.setString(2, “xyfdsw”);

pstmt.execute();

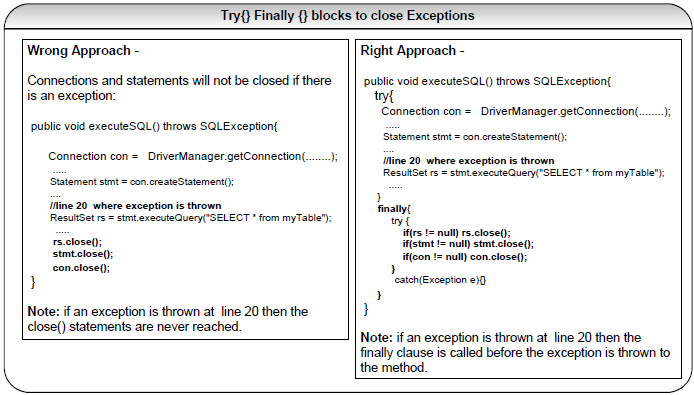
How to avoid the “running out of cursors” problem?

A database can run out of cursors if the connection is not closed properly or the DBA has not allocated enough

cursors. In a Java code it is essential that we close all the valuable resources in a try{} and **finally**{} block. The

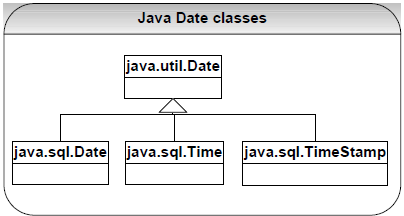
finally{} block is always executed even if there is an exception thrown from the catch {} block. So the resources like

connections and statements should be closed in a finally {} block.



Explain differences among java.util.Date, java.sql.Date, java.sql.Time, and java.sql.Timestamp?

As shown below all the sql Date classes extend the util Date class.



**java.util.Date -** class supports both the Date (ie year/month/date etc) and the Time (hour, minute, second, and

millisecond) components.

**java.sql.Date -** class supports only the Date (ie year/month/date etc) component. The hours, minutes, seconds

and milliseconds of the Time component will be set to zero in the particular time zone with which the instance is

associated.

**java.sql.Time -** class supports only Time (ie hour, minute, second, and millisecond) component. The date

components should be set to the "zero epoch" value of January 1, 1970 and should not be accessed.

**java.sql.TimeStamp –** class supports both Date (ie year/month/date etc) and the Time (hour, minute, second,

millisecond and **nanosecond**) components.

**Note:** the subtle difference between **java.util.Date** and **java.sql.Date.**

To keep track of time Java counts the number of milliseconds from January 1, 1970 and stores it as a long value in

**java.util.Date** class. The **GregorianCalendar** class provides us a way to represent an arbitrary date. The

**GregorianCalendar** class also provides methods for manipulating dates.