
Advanced Java

JDBC & The Way Ahead

Objective

At the end of the session, you will be able to:

- Understand JDBC Architecture
- Differentiate among JDBC Driver types
- Write a database handling program using Type-1 & Type-2 JDBC drivers
- Introduce JDBC 2.0 features
- Execute Java program on UNIX
- Introduce new features in Java

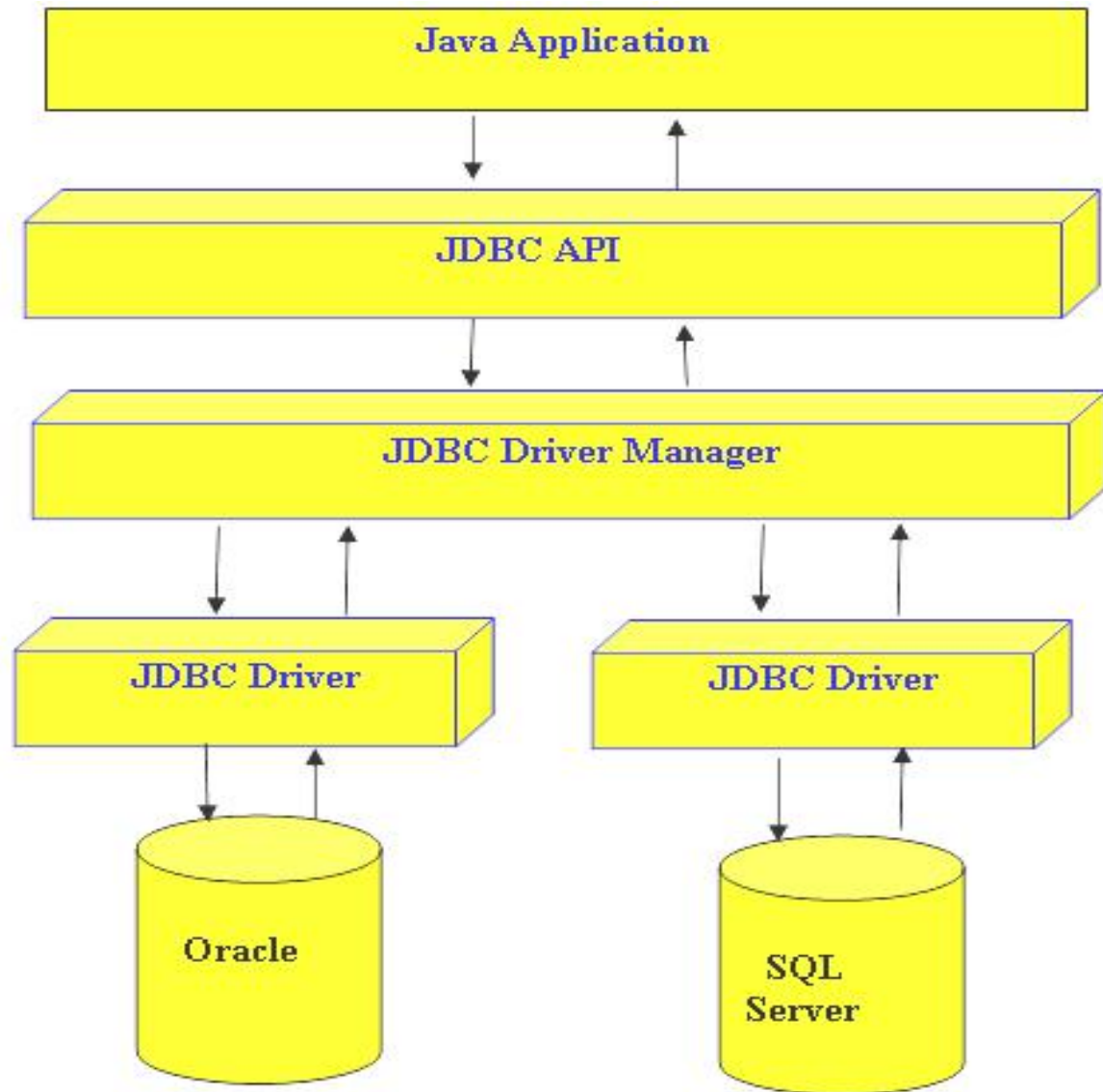
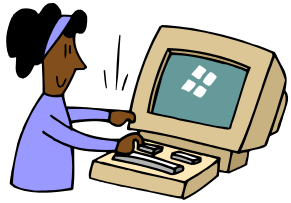
Agenda

- JDBC Architecture
- Types of JDBC Drivers
- Loading JDBC Driver
- Connecting to Database
- Manipulate data in the Database
- Transaction Management
- Feature of JDBC 2.0
- Java on UNIX
- New Features in Java

What is JDBC?

- An API specification developed by Sun Microsystems
- Defines a uniform interface for accessing various relational databases
- The JDBC API uses a Driver Manager and database-specific drivers to provide connectivity to heterogeneous databases
- Driver Manager can support multiple concurrent drivers connected to multiple heterogeneous databases
- A JDBC driver translates standard JDBC calls into a network or database protocol or into a database library API call that facilitates communication with the database

JDBC Architecture

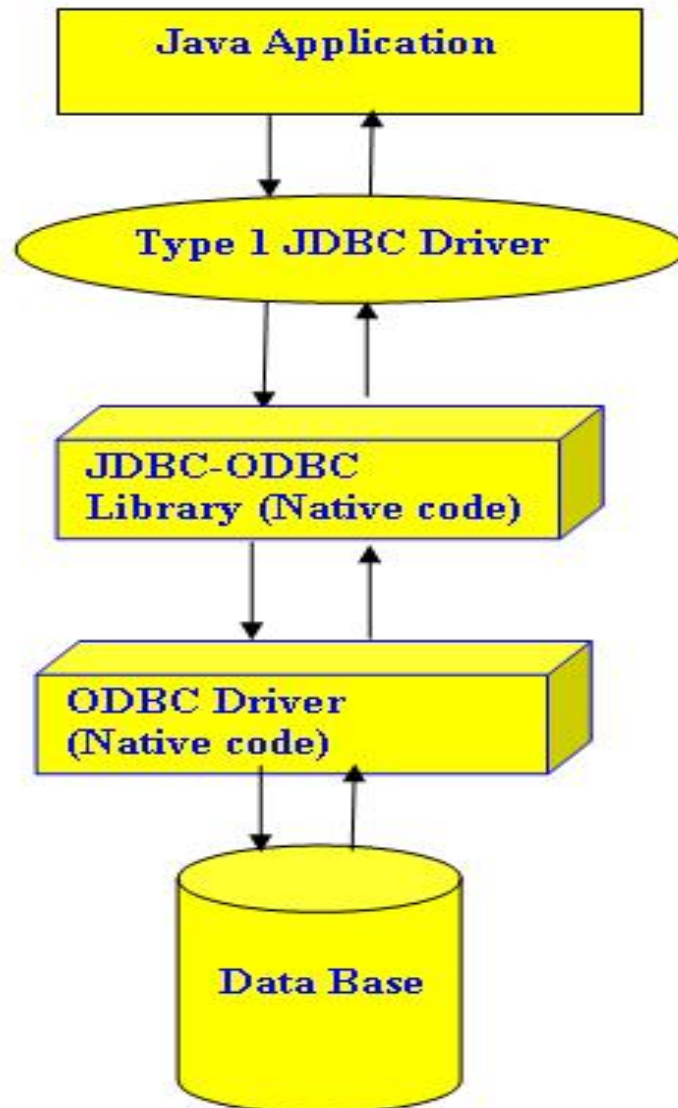


Types of JDBC Drivers

Four distinct types of JDBC drivers:

1. JDBC-ODBC Bridge Driver - Type1
2. Native API Java Driver - Type2
3. Java to Network Protocol Driver - Type3
4. Pure Java Driver - Type4

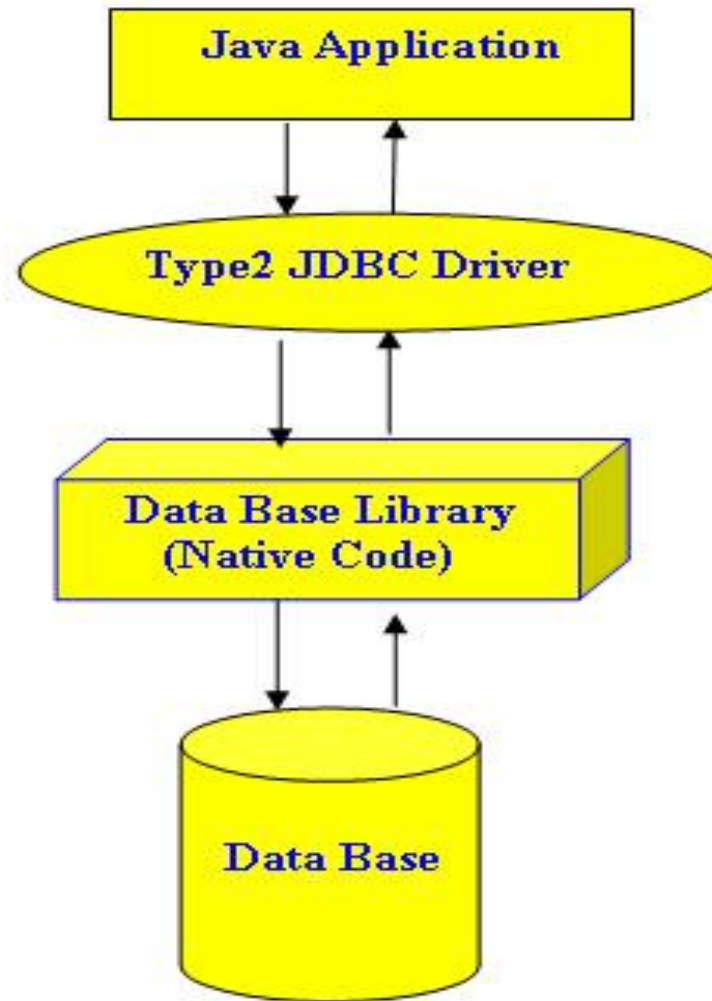
Type 1: JDBC-ODBC Bridge Driver



Type 1: JDBC-ODBC Bridge Driver (Contd...)

- Act as a "bridge" between JDBC & database connectivity mechanism like ODBC
- The bridge provides JDBC access using most standard ODBC drivers
- This driver is included in the Java 2 SDK within the *sun.jdbc.odbc* package
- JDBC statements call the ODBC by using the JDBC - ODBC Bridge and finally the query is executed by the database

Type 2: Java to Native API Driver



Type 2: Java to Native API Driver (Contd...)

- Use the Java Native Interface (JNI) to make calls to a local database library API
- Converts JDBC calls into a database specific call for databases such as SQL, ORACLE etc.
- Communicates directly with the database server & requires some native code to connect to the database
- Usually faster than Type 1 drivers
- Like Type 1 drivers, Type 2 drivers require native database client libraries to be installed and configured on the client machine

Advantages

- As there is no implementation of jdbc-odbc bridge, its considerably faster than a type 1 driver.

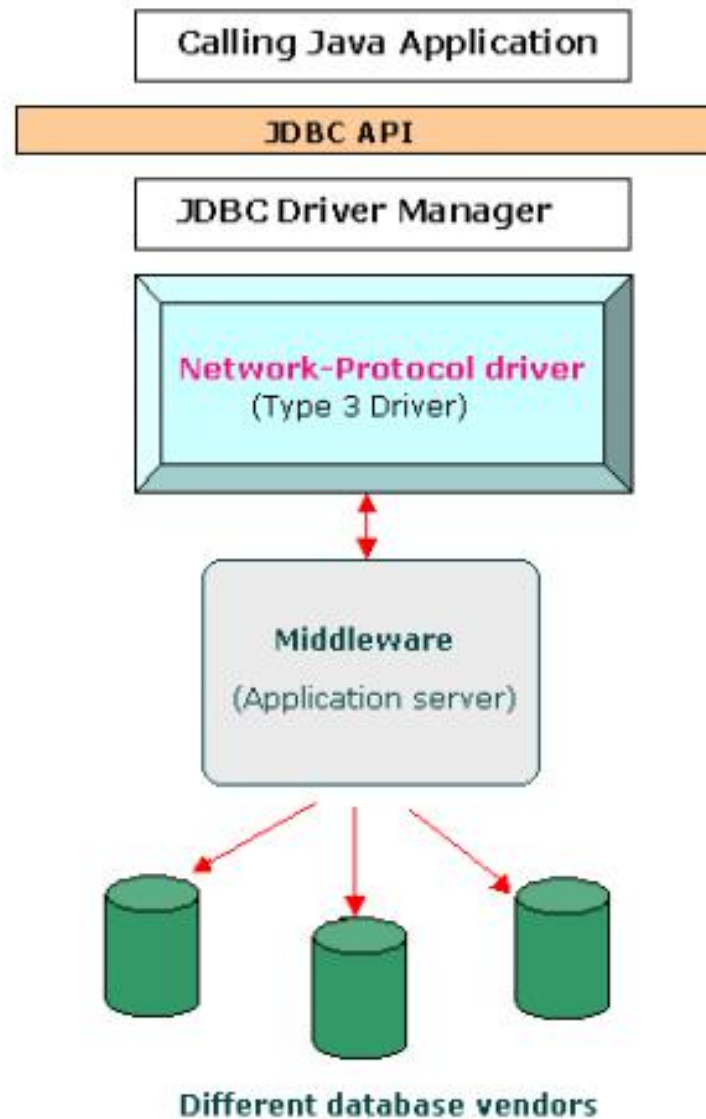
Disadvantages

- The vendor client library needs to be installed on the client machine.
- Not all databases have a client side library
- This driver is platform dependent
- This driver supports all java applications except Applets

Type 3: Java to Network Protocol Driver (Contd...)

- The JDBC type 3 driver, also known as the **Pure Java Driver for Database Middleware**, is a database driver implementation which makes use of a middle tier between the calling program and the database.
- The middle-tier (application server) converts JDBC calls directly or indirectly into the vendor-specific database protocol.
- Pure Java drivers that use a proprietary network protocol to communicate with JDBC middleware on the server
- Do not require native database libraries on the client and can connect to many different databases on the back end
- Can be deployed over the Internet without client installation

Type 3: Java to Network Protocol Driver

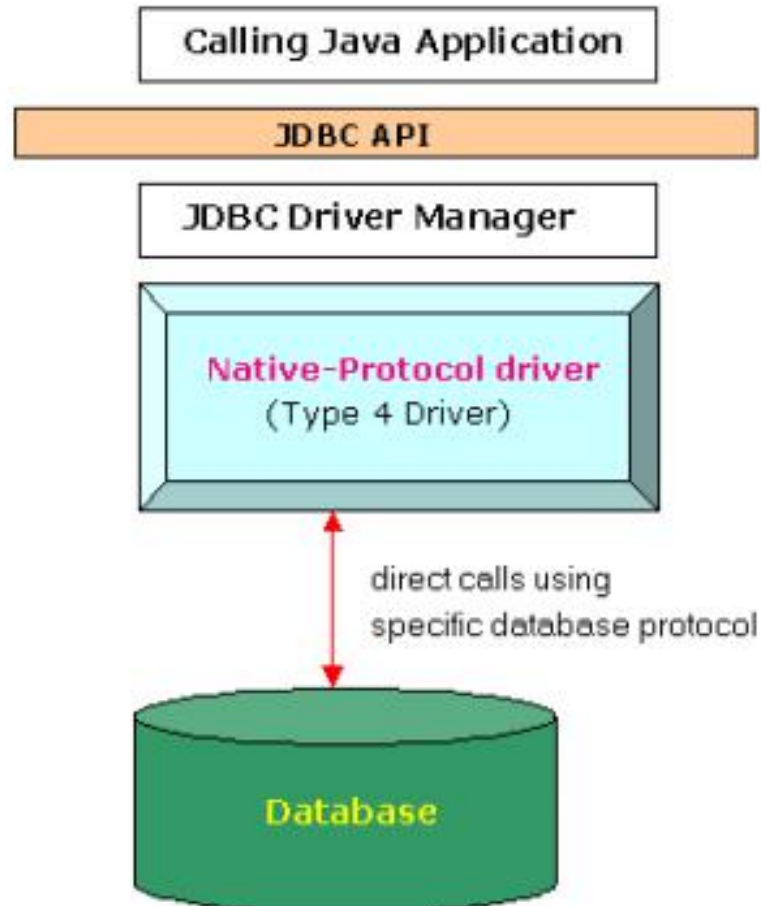


- This differs from the type 4 driver in that the protocol conversion logic resides not at the client, but in the middle-tier.
- The same driver can be used for multiple databases.
- It depends on the number of databases the middleware has been configured to support.
- The type 3 driver is platform-independent as the platform-related differences are taken care of by the middleware.
- Also, making use of the middleware provides additional advantages of security and firewall access.

Working

- Sends JDBC API calls to a middle-tier net server that translates the calls into the DBMS-specific network protocol. The translated calls are then sent to a particular DBMS.
- Follows a three tier communication approach.
- Can interface to multiple databases - Not vendor specific.
- The JDBC Client driver written in java, communicates with a middleware-net-server using a database independent protocol, and then this net server translates this request into database commands for that database.
- So the client driver to middleware communication is database independent.

Type 4: Java to Database Protocol Driver



Type 4: Java to Database Protocol Driver (Contd...)

- The JDBC type 4 driver, also known as the **Direct to Database Pure Java Driver**, is a database driver implementation that converts JDBC calls directly into a vendor-specific database protocol.
- Unlike the type 3 drivers, it does not need associated software to work.
- Communicate directly with the database engine rather than through middleware or a native library
- It is installed inside the Java Virtual Machine of the client. This provides better performance than the type 1 and type 2 drivers as it does not have the overhead of conversion of calls into ODBC or database API calls.
- One drawback is that they are database specific
- Usually the **fastest JDBC** drivers available
- Directly converts java statements to SQL statements

Using JDBC API

- The JDBC API classes and interfaces are available in the `java.sql` and the `javax.sql` packages.
 - The commonly used classes and interfaces in the JDBC API are:
 - `DriverManager` class: Loads the driver for a database.
 - `Driver` interface: Represents a database driver. All JDBC driver classes must implement the `Driver` interface.
 - `Connection` interface: Enables you to establish a connection between a Java application and a database.
 - `Statement` interface: Enables you to execute SQL statements.
 - `ResultSet` interface: Represents the information retrieved from a database.
 - `SQLException` class: Provides information about the *exceptions* that occur while interacting with databases.
-

Using JDBC API (Contd.)

- The steps to create JDBC application are:
 - Load a driver
 - Connect to a database
 - Create and execute JDBC statements
 - Handle SQL exceptions
-

Loading the Driver

- Can be done in two ways:
 - Using *DriverManager*: Uses system property jdbc.drivers to know drivers to be loaded
 - Using *Class.forName()*:
 - ◆ A fully qualified name of the driver class should be passed as a parameter
 - ◆ The following call will load the Sun JDBC-ODBC driver:

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

Using JDBC API (Contd.)

- Using the `forName()` method
 - The `forName()` method is available in the `java.lang.Class` class.
 - The `forName()` method loads the JDBC driver and registers the driver with the driver manager.
 - The method call to use the `forName()` method is:
`Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");`
-

Using JDBC API (Contd.)

- Using the `registerDriver()` method
 - You can create an instance of the `Driver` class to load a JDBC driver.
 - This instance enables you to provide the name of the driver class at run time.
 - The statement to create an instance of the `Driver` class is:

```
Driver d = new sun.jdbc.odbc.JdbcOdbcDriver();
```
 - You need to call the `registerDriver()` method to register the `Driver` object with the `DriverManager`.
 - The method call to register the JDBC-ODBC Bridge driver is:

```
DriverManager.registerDriver(d);
```
-

Connecting to Database

■ The *getConnection()* method:

- A static method of *DriverManager* class
- Provides many overloaded versions
- Returns a *Connection* object

//-----Connection with Type-1 Driver-----//

```
con=DriverManager.getConnection("jdbc:odbc:dsn1","system","manager");
```

URL

//-----Connection with Type-4 Driver-----//

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

Port

Driver
Type

Server IP

Service ID



JdbcConnect.java

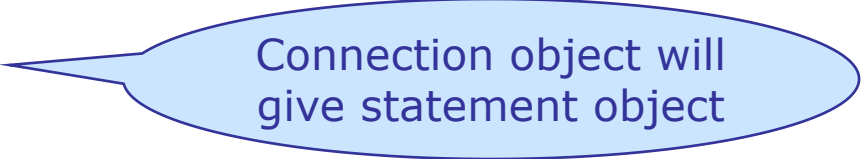
The Connection Interface

- When a Connection is opened, this represents a single instance of a particular database session
- As long as the connection remains open, SQL queries may be executed and results obtained
- Has Methods which return objects of:
 - Statement
 - PreparedStatement
 - CallableStatement

The Statement Interface

- Used pass a SQL String to the database for execution and to retrieve result from database

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



Connection object will give statement object

- The *createStatement()* method of Connection interface returns a Statement object
- Provides methods to execute SQL statements:
 - *executeQuery()*
 - ◆ Used for SQL statements such as simple SELECT, which return a single result set
 - *executeUpdate()*
 - ◆ Used for other SQL statements like INSERT, UPDATE & DELETE

The *PreparedStatement* Interface

- The *prepareStatement()* method of Connection interface returns a PreparedStatement object
- Useful for frequently executed SQL statements
- An SQL statement is pre-compiled and gets executed more efficiently than a plain statement
- For example: the statement **"INSERT INTO EMP VALUES (?, ?)"** can be used to add multiple records, with a different values each time
 - '?' acts as a placeholder
- Records are in *emp* table as follows:

```
pspmt=con.prepareStatement("insert into emp values (?,?)");  
pspmt.setInt(1,26788);  
pspmt.setString(2,"Ajay");
```



The *PreparedStatement* Interface (Contd...)

- The *PreparedStatement* Interface:
 - methods *executeQuery()* and *executeUpdate* for statement execution
 - *setXXX()* methods for assigning values for the placeholders
 - ◆ XXX stands for the data type of the value of the placeholder
 - ◆ For example: *setString()*, *setInt()*, *setFloat()*, *setDouble()* etc.
- To delete a record, the following code snippet is used:

```
pstmt = con.prepareStatement("Delete FROM emp WHERE empno =?" )  
pstmt.setString(1,eno);  
int i = pstmt.executeUpdate();
```



JdbcDelete.java

The *PreparedStatement* Interface (Contd...)

- To update records, the following code snippet can be used:

```
PreparedStatement stmt = con.prepareStatement("update  
Emp SET Ename = ? WHERE Empno = ?");  
st.setString(1, "Amit");  
st.setString(2, "50000");  
int i = stmt.executeUpdate();
```



JdbcUpdate.java

The *CallableStatement* Interface

- A *CallableStatement* object is created by calling the *prepareCall()* method on a *Connection* object
- The object provides a way to call stored procedures in a standard way for all DBMS
- *CallableStatement* inherits *Statement* methods, which deal with SQL statements in general, and it also inherits *PreparedStatement* methods, which deal with IN parameters
- All methods defined in the *CallableStatement* deal with OUT parameters or the output aspect of INOUT parameters

The *CallableStatement* Interface (Contd...)

- Syntax for a stored procedure without parameters is:

```
{call procedure_name}
```

- Syntax for procedure call with two parameters:

```
{call procedure_name[(?, ?)] }
```

The *CallableStatement* Interface (Contd...)

\\ Creating CallableStatement

```
CallableStatement cs;  
cs=con.prepareCall("{call SHOW_SUPPLIERS}");  
ResultSet rs = cs.executeQuery();
```

\\ Stored Procedures with parameters

```
int age = 39;  
String poetName = "dylan thomas";  
CallableStatement proc =  
con.prepareCall("{call set_age(?, ?)}");  
proc.setString(1, poetName);  
proc.setInt(2, age);  
cs.execute();
```

Procedure In Oracle

SQL>

```
create or replace procedure proc1(x in int) is
begin
  dbms_output.put_line('hello world');
end;
```

SQL> set serveroutput on;

SQL> execute proc1(100) ;

inout parameter in procedure

create or replace procedure emp_detail(eid in int , name1 out varchar, salary1 out int) is

 ename varchar(20);

 salary number;

begin

 select name,sal into ename,salary from employee
 where id=eid;

 name1:=ename;

 salary1:=salary;

exception

when no_data_found then

 dbms_output.put_line('no data found for the given id ');

end;

procedure calling

procedure calling

```
declare
name varchar(12);
salary int;
begin
emp_detail(101,name,salary);
dbms_output.put_line('name is '||name);
dbms_output.put_line('salary is '||salary);
end;
```

-----using bind variable -----

```
VARIABLE name      VARCHAR2(25)
VARIABLE sal NUMBER
EXECUTE emp_detail(171, :name, :sal)
print name
print sal
```

```
void dispEmp() throws SQLException {
    cst = con.prepareCall("{call emp_detail(?,?,?)}");
    System.out.println("enter employee id ");
    cst.setInt(1, sc.nextInt());
    cst.registerOutParameter(2, java.sql.Types.VARCHAR);
    cst.registerOutParameter(3, java.sql.Types.FLOAT);

    cst.execute();
    name = cst.getString(2);
    sal = cst.getFloat(3);
    System.out.println("Employee detail ");
    System.out.println(name + "\t" + sal);

}
```

function in callable

```
CREATE OR REPLACE FUNCTION cust_sal  
(id1 IN int)  
  RETURN int  
IS  
    sal1 int :=0;  
BEGIN  
  SELECT sal  
    INTO  sal1  
    FROM  customer  
    WHERE id = id1;  
  RETURN sal1;  
END;
```

```
-----  
VARIABLE g_salary NUMBER  
EXECUTE :g_salary := cust_sal(117)  
PRINT g_salary
```

```
Connection con =  
DriverManager.getConnection("jdbc:oracle:thin:@localhost:15  
21:XE", "system", "manager");
```

```
        CallableStatement cst =  
con.prepareCall("{?=call cust_sal(?)}");  
        cst.registerOutParameter(1,  
java.sql.Types.INTEGER);  
        cst.setInt(2, 1001);  
        cst.execute();  
        int p = cst.getInt(1);
```

```
System.out.println(p);
```

Give this a Try...

1. Which Driver uses JDBC-ODBC Bridge to translate JDBC statements calls to the ODBC calls?
2. Which driver converts JDBC calls into the vendor-specific database protocol so that client applications can communicate directly with the database server?
3. Which interface do you think provides a way to call stored procedures in a standard way for all DBMS?
4. Which class should be used to efficiently execute SQL statement multiple times?

The *ResultSet* Interface

- The result of an SQL statement execution can be:
 - A *ResultSet* object
 - ◆ The object returned by *executeQuery()* method
 - ◆ Contains records retrieved by the SELECT statement
 - An integer
 - ◆ Returned in case of INSERT, UPDATE, DELETE statements
 - ◆ Indicates the number of rows affected due to the statement

```
rs=stmt.executeQuery("select * from emp");
```

The *ResultSet* Interface (Contd...)

- A *ResultSet* object represents the output table of data resulted from a SELECT query statement with following features:
- The data in a *ResultSet* object is organized in rows & columns
- *Each ResultSet object maintains a cursor (pointer) to identify the current data row*
- The cursor of a newly created *ResultSet* object is positioned before the first row

The *ResultSet* Interface (Contd...)

- Movement of the cursor depends on the scrollability of the *ResultSet*
 - Non-scrollable *ResultSet*:
 - ◆ Object type is *ResultSet.TYPE_FORWARD_ONLY*, the default type
 - ◆ Supports only forward move
 - Scrollable *ResultSet*:
 - ◆ Object type is *ResultSet.TYPE_SCROLL_INSENSITIVE* (scrollable but not sensitive to changes made by others) or
 - ◆ *ResultSet.TYPE_SCROLL_SENSITIVE* (scrollable and sensitive to changes made by others)

The *ResultSet* Interface (Contd...)

- *next()* method:
 - Moves the record pointer to the next record in the result set
 - Returns a boolean value true / false, depending on whether there are records in the result set
- *getXXX()* methods:
 - XXX stands for data type of the value being retrieved
 - Retrieve the values of individual column in the result set
 - Two overloaded versions of each *getXXX()* method are provided:
 1. Accepting an *integer argument* indicating the column position
 2. Accepting a *string argument* indicating name of the column

The *ResultSet* Interface (Contd...)

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT empno, ename, sal
                                FROM emp");
while (rs.next() )
{
    //assuming there are 3 columns in the table
    System.out.println( rs.getString(1));
    System.out.println(rs.getString(2));
    System.out.println(rs.getString(3));
}
```

Don't forget to close
the Resultset,
Statement &
Connection

```
rs.close(); //---- First ----//
stmt.close(); //---- Second ----//
con.close(); //---- Last ----//
```

Maintain the
sequence as shown

```
System.out.println(" You are done");
```

The *ResultSetMetadata* Interface

- *getMetaData()* of the *ResultSet* interface returns a *ResultSetMetadata* object containing details about the columns in a result set
- Some of the methods of *ResultSetMetadata* interface are:
 - *getColumnName()*
 - ◆ Returns the name of a column by taking an integer argument indicating the position of the column within the result set
 - *getColumnType()*
 - ◆ Returns the data type of a column
 - *getColumnCount()*
 - ◆ Returns the number of columns included in the result set

The *ResultSetMetadata* Interface (Contd...)

```
rsmat=rs.getMetaData();  
  
int cols=rsmat.getColumnCount()  
  
while(rs.next()  
{  
    for(int i=1;i<=cols;i++)  
    {  
        System.out.print(rs.getString(i));  
    }  
}
```

Example

```
ResultSet rset = stmt.executeQuery("SELECT * FROM data");
ResultSetMetaData rsmeta = rset.getMetaData();

int numCols = rsmeta.getColumnCount();

for (int i=1; i<=numCols; i++) {

    int ct    = rsmeta.getColumnType(i);
    String cn = rsmeta.getColumnName(i);
    String ctn = rsmeta.getColumnTypeName(i);

    System.out.println("Column #" + i + ": " + cn +
        " of type " + ctn + " (JDBC type: " + ct + ")");
}
```

Give this a Try...

1. Which method do you think returns ResultSet?
2. Which method of ResultSet moves the cursor to the next row?
3. Which Object do you think can be used to get information about the types (tables) and properties of the columns in a ResultSet object?
4. ResultSet objects support three types of scrollabilities. Specify all types?
5. ResultSet objects support two types of update capabilities. Specify all types?

Introduction to Transaction

- An indivisible unit of work comprised of several operations
- Either all, or none of the units need to be performed in order to preserve data integrity
- A complete task which is a combination of the smaller tasks
- For a major task to be completed, smaller tasks need to be successfully completed
- If any one task fails then all the previous tasks are reverted back to the original state

Transaction Properties

■ Atomicity:

- Implies indivisibility
- Any indivisible operation (one which will either complete in totally, or not at all) is said to be atomic

■ Consistency:

- A transaction must transition persistent data from one consistent state to another
- In the event of a failure occurs during processing, data must be restored to the state it was in prior to the transaction

Transaction Properties (Contd...)

■ Isolation:

- Transactions should not affect each other
- A transaction in progress, not yet committed or rolled back, must be isolated from other transactions

■ Durability:

- Once a transaction commits successfully, the state changes committed by that transaction must be durable & persistent, despite any failures that occur afterwards

Transaction Properties (Contd...)

- **COMMIT and ROLLBACK:**

- These two keywords **Commit** and **Rollback** are mainly used for MySQL Transactions.
- When a successful transaction is completed, the COMMIT command should be issued so that the changes to all involved tables will take effect.
- If a failure occurs, a ROLLBACK command should be issued to return every table referenced in the transaction to its previous state.
- `con.commit();`
- `con.rollback();`

Transaction Properties (Contd...)

■ **AutoCommit Option**

- Instead of issuing every time commit we can use autocommit.
- It will automatically commit after every query.

It has value 1 or 0.

- 1 Is for AutoCommit is on
- 0 for AutoCommit is off.

Transaction Management

- A connection commits all the changes once it is done with a query by default
- This can be stopped by calling *setAutoCommit(false)* on the connection object
- Now for every transaction, user will have to explicitly call the *commit()* method, else the changes won't be reflected in the database
- The *rollback()* method gets called for withdrawing all the changes done by queries in the database



Transaction Management (Contd...)

- In mysql SET autocommit = 0;

By default the Connection commits all the changes once it is done with a single query.

This can be stopped by calling *setAutoCommit(false)* on Connection object.

```
con.setAutoCommit(false);  
System.out.println("Connection established");  
stmt=con.createStatement();
```

Transaction Management (Contd...)

Now for every transaction user has to explicitly call *commit()* method, only then changes get reflected otherwise changes wont be shown in the database

```
con.commit();  
stmt.close();  
con.close();
```

Transaction Management (Contd...)

```
con.commit();  
con.rollback();
```

The rollback() method gets called for withdrawing all the changes done by the queries in the database



TransactionRollBack.java

Implementing Transactions using JDBC

Step 1: Start the Transaction

// Disable auto-commit mode

cn.setAutoCommit(false);

Step 2: Perform Transactions

```
PreparedStatement updateSales = cn.prepareStatement("UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");

updateSales.setInt(1, 50);
updateSales.setString(2, "Russian");
updateSales.executeUpdate();

PreparedStatement updateTotal = cn.prepareStatement("UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");

updateTotal.setInt(1, 50);
updateTotal.setString(2, "Russian");
updateTotal.executeUpdate();
```

Step 3: Use SavePoint

```
// Create an instance of Statement object
Statement st = cn.createStatement();
int rows = st.executeUpdate("INSERT INTO EMPLOYEE (NAME) VALUES (?COMPANY?)");

// Set the Savepoint
Savepoint svpt = cn.setSavepoint("SAVEPOINT_1");
rows = st.executeUpdate("INSERT INTO EMPLOYEE (NAME) VALUES (?FACTORY?)");
```

Step 4: Close the Transaction

...

```
// End the transaction
cn.rollback(svpt);
```

OR

...

```
cn.commit();
```

Give this a Try...

1. Enlist the properties of the transaction
2. What is the purpose of `setAutocommit()`?
3. What is the use of `commit()` method?
4. Which method is used in withdrawing all the changes done by the queries in the database?

Features of JDBC 2.0

■ *ResultSet* enhancements

- Support for scrollable *resultset* capability under three major headings:
 - ◆ Forward-only
 - ◆ Scroll-insensitive
 - ◆ Scroll-sensitive
- Each of these *resultsets* can in turn be Read Only or Updatable
- Forward Only / Read Only was a feature of JDBC 1.0

■ Batch Updates

- Automatic & explicit batch update

■ Advanced Data Types

- Advanced data types like objects, object references, arrays, LOBS, SQL Data & Struct

Features of JDBC 2.0 (Contd...)

- Connection Pooling:

- A method where multiple consumers share a limited set of connections instead of each having to create new connections
- Connection caching also included

Database Metadata

- `java.sql.DatabaseMetaData`
provides information about the database (schema etc.)

- Information about the database
 - Name of database
 - Version of database
 - List of all tables
 - List of supported SQL types
 - Support of transactions

```

        DatabaseMetaData
dmeta = con.getMetaData();

        System.out.println("databas
e product name "

                                +
dmeta.getDatabaseProductName())
; // mysql

        System.out.println("databas
e product version is "

                                +
dmeta.getDatabaseProductVersion(
)); // mysql

        System.out.println("Driver
name " + dmeta.getDriverName());

        rs =
dmeta.getSchemas();

        System.out.println("schema
names are "); // in oracle

```

```

while (rs.next()) {

        System.out.println(rs.getString(1
));

        }

        rs = dmeta.getTables(null, null, "%",
null);

        System.out.println("table names
are "); // in oracle

        while (rs.next()) {

                System.out.println(rs.getString(3
));

                }

                rs =
dmeta.getCatalogs(); // database name

                System.out.println("database
catalogs are .....");

                while (rs.next()) {

```

```

                System.out.println(rs.getString(1

```

Isolation Issues

- How do different transactions interact?
- Does a running transaction see **uncommitted changes**?
- Does it see **committed changes**?
- Can two transactions update the same row?

Locking

- To avoid conflicts during a transaction, a DBMS uses locks, mechanisms for blocking access by others to the data that is being accessed by the transaction.
- After a lock is set, it remains in force until the transaction is committed or rolled back. For example, a DBMS could lock a row of a table until updates to it have been committed.
- The effect of this lock would be to prevent a user from getting a dirty read, that is, reading a value before it is made permanent.
- Accessing an updated value that has not been committed is considered a dirty read because it is possible for that value to be rolled back to its previous value. So reading a value that is later rolled back, you will have read an invalid value.

Locks are set is determined by transaction isolation level.
It decide how one process is isolated from the other.

Isolation Levels

- The transaction isolation levels in JDBC determine whether the concurrently running transactions in a database can affect each other or not.
- Some common problems that might occur when multiple transactions simultaneously access a database are:
 - Dirty reads
 - Non-repeatable reads
 - Phantom reads

Isolation Levels (Contd.)

- The isolation levels enable you to isolate the concurrently running transactions so that a transaction cannot affect the result of another transaction.
 - The `Connection` interface of the JDBC API provides the following fields as int values to set isolation levels:
 - `TRANSACTION_READ_UNCOMMITTED`
 - `TRANSACTION_READ_COMMITTED`
 - `TRANSACTION_REPEATABLE_READ`
 - `TRANSACTION_SERIALIZABLE`
-

Isolation level

- Locks are set is determined by transaction isolation level.
- It decide how one process is isolated from the other.

There four Transaction Isolation Levels

1. Read Committed

- - No two transactions can change the data at the same time. The shared lock is held for the record for the duration of the transaction and no other transaction can access the record.

2. Read Uncommitted

- - This allows least restrictions to the record, meaning no shared locks and no exclusive locks. This kind offers best data concurrency but threat to data integrity.

3. Read Table Read

- - This level allows new rows that can be inserted in the table and can even be read by the transactions.

▪ 4. Serializable

- - This applies most restrictive setting holding shared locks on the range of data.

- **Dirty reads:** A transaction reads data that is written by another, uncommitted transaction
- **Non-repeatable reads:** A transaction rereads data it previously read and finds that a committed transaction has modified or deleted that data
- **Phantom reads:** A transaction re-executes a query returning a set of rows satisfying a search condition and finds that a committed transaction inserted additional rows that satisfy the condition

READ COMMITTED vs. SERIALIZABLE

	READ COMMITTED	SERIALIZABLE
Dirty Reads	Impossible	Impossible
Non-repeatable Reads	Possible	Impossible
Phantom Reads	Possible	Impossible

Isolation Levels (Contd.)

- The `Connection` interface contains the following methods to retrieve and set the value of transaction isolation level for a database:

- `getTransactionIsolationLevel()`
- `setTransactionIsolationLevel()`

- The code snippet to use the `getTransactionIsolationLevel()` method is:

```
Connection con = DriverManager.getConnection
("jdbc:odbc:MyDatasource","administrator","");
int transLevel=getTransactionIsolationLevel();
```

- The code snippet to use the `setTransactionIsolationLevel()` method is

```
Connection con = DriverManager.getConnection
("jdbc:odbc:MyDatasource","administrator","");
int transLevel=getTransactionIsolationLevel();
con.setTransactionIsolationLevel(TRANSACTION_SERIALIZABLE);
```

Isolation Level	Transactions	Dirty Reads	Non-Repeatable Reads	Phantom Reads
TRANSACTION_NONE	Not supported	<i>Not applicable</i>	<i>Not applicable</i>	<i>Not applicable</i>
TRANSACTION_READ_COMMITTED	Supported	Prevented	Allowed	Allowed
TRANSACTION_READ_UNCOMMITTED	Supported	Allowed	Allowed	Allowed
TRANSACTION_REPEATABLE_READ	Supported	Prevented	Prevented	Allowed
TRANSACTION_SERIALIZABLE	Supported	Prevented	Prevented	Prevented

- **A non-repeatable read** occurs when transaction A retrieves a row, transaction B subsequently updates the row, and transaction A later retrieves the same row again. Transaction A retrieves the same row twice but sees different data.
- **A phantom read** occurs when transaction A retrieves a set of rows satisfying a given condition, transaction B subsequently inserts or updates a row such that the row now meets the condition in transaction A, and transaction A later repeats the conditional retrieval. Transaction A now sees an additional row. This row is referred to as a phantom.

Row Set

A JDBC **RowSet** object holds tabular data in a way that makes it more flexible and easier to use than a result set. A *Row Set* is an object which encapsulates a set of rows.

Kinds of RowSet Objects

Connected RowSet object uses a driver based on JDBC technology ("JDBC Driver") to make a connection to a relational database and maintains that connection throughout its life span.

Only one of the standard **RowSet** implementations is a connected **RowSet**: **JdbcRowSet**. Being always connected to a database, it is very similar to a **ResultSet** object and is often used as a wrapper to make an otherwise nonscrollable and read-only ResultSet object scrollable and updatable.

You can create a **JdbcRowSet** object in two ways:

- ♦ By using the reference implementation constructor that takes a ResultSet object
- ♦ By using the reference implementation default constructor

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(select * from COFFEES);
JdbcRowSet jdbcRs = new JdbcRowSetImpl(rs);
```

Batch updates

- A set of multiple update statements that is submitted to the database for processing as a batch
- Statement, PreparedStatement and CallableStatement can be used to submit batch updates

Implementing Batch Update using “Statement” interface

- ❏ Disable the auto-commit mode
- ❏ Create a Statement instance
- ❏ Add SQL commands to the batch
- ❏ Execute the batch commands
- ❏ Commit the changes in the database
- ❏ Remove the commands from the batch

Row Set

Using the Default Constructor

The following line of code creates an empty `JdbcRowSet` object.

```
JdbcRowSet jdbcRs2 = new JdbcRowSetImpl();
```

Disconnected `RowSet` objects make a connection to a data source only to read in data from a `ResultSet` object or to write data back to the data source.

The other four implementations are disconnected `RowSet` implementations.

1. A `CachedRowSet` object has all the capabilities of a `JdbcRowSet` object plus it can also do the following:
 - Obtain a connection to a data source and execute a query
 - Read the data from the resulting `ResultSet` object and populate itself with that data
 - Manipulate data and make changes to data while it is disconnected
 - Reconnect to the data source to write changes back to it
 - Check for conflicts with the data source and resolve those conflicts

Row Set

You can create a new **CachedRowSet** object in two different ways:

- By using the default constructor

```
CachedRowSet crs = new CachedRowSetImpl();
```

- By Passing a SyncProvider implementation to the constructor

```
CachedRowSet crs2 =  
    CachedRowSetImpl("com.fred.providers.HighAvailabilityProvider");
```

2. A **WebRowSet** object has all the capabilities of a **CachedRowSet** object plus it can also do the following:

- ◆ Write itself as an XML document
- ◆ Read an XML document that describes a **WebRowSet** object

```
WebRowSet pricelist = new WebRowSetImpl();
```

3. A **JoinRowSet** object has all the capabilities of a **WebRowSet** object (and therefore also a **CachedRowSet** object) plus it can also do the following:

- Form the equivalent of an SQL JOIN without having to connect to a data source

```
JoinRowSet jrs = new JoinRowSetImpl();
```

Row Set

4. A `FilteredRowSet` object likewise has all the capabilities of a `WebRowSet` object (and therefore also a `CachedRowSet` object) plus it can also do the following:

- ◆ Apply filtering criteria so that only selected data is visible. This is equivalent to executing a query on a `RowSet` object without having to use a query language or connect to a data source.

```
FilterRowSet frs = new FilteredRowSetImpl();
```

This object is initialized with the following:

- ◆ The high end of the range within which values must fall
- ◆ The low end of the range within which values must fall
- ◆ The column name or column number of the value that must fall within the range of values set by the high and low boundaries

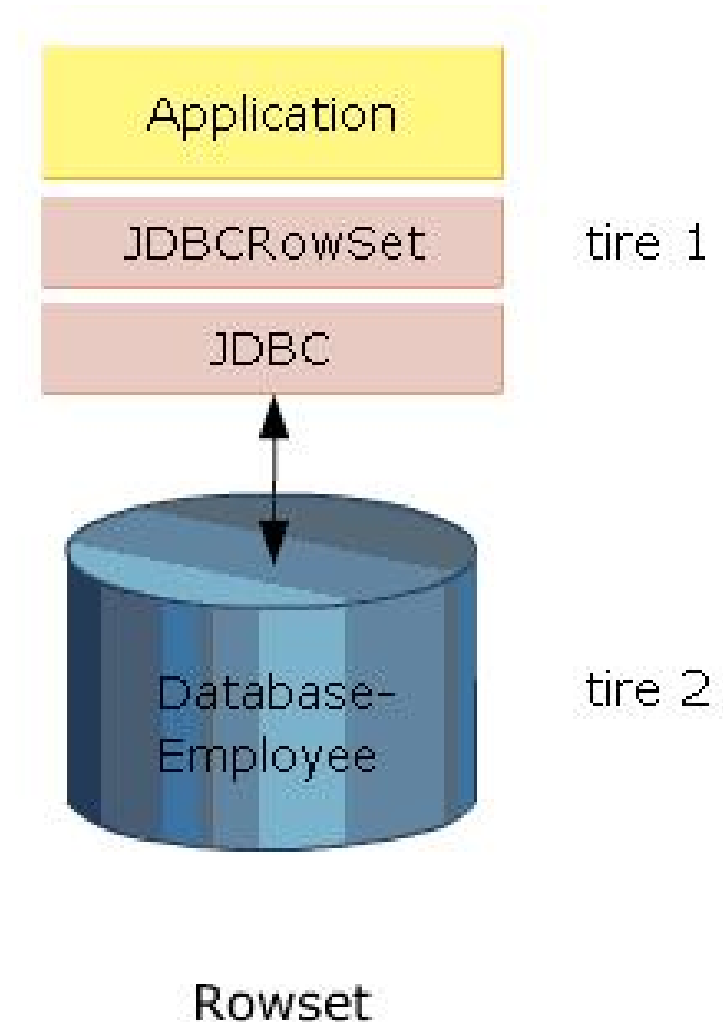
```
Filter1 range = new Filter(1000, 10999, "STORE_ID");
```

The next line of code sets *range* as the filter for *frs*.

```
frs.setFilter(range);
```

Rowsets

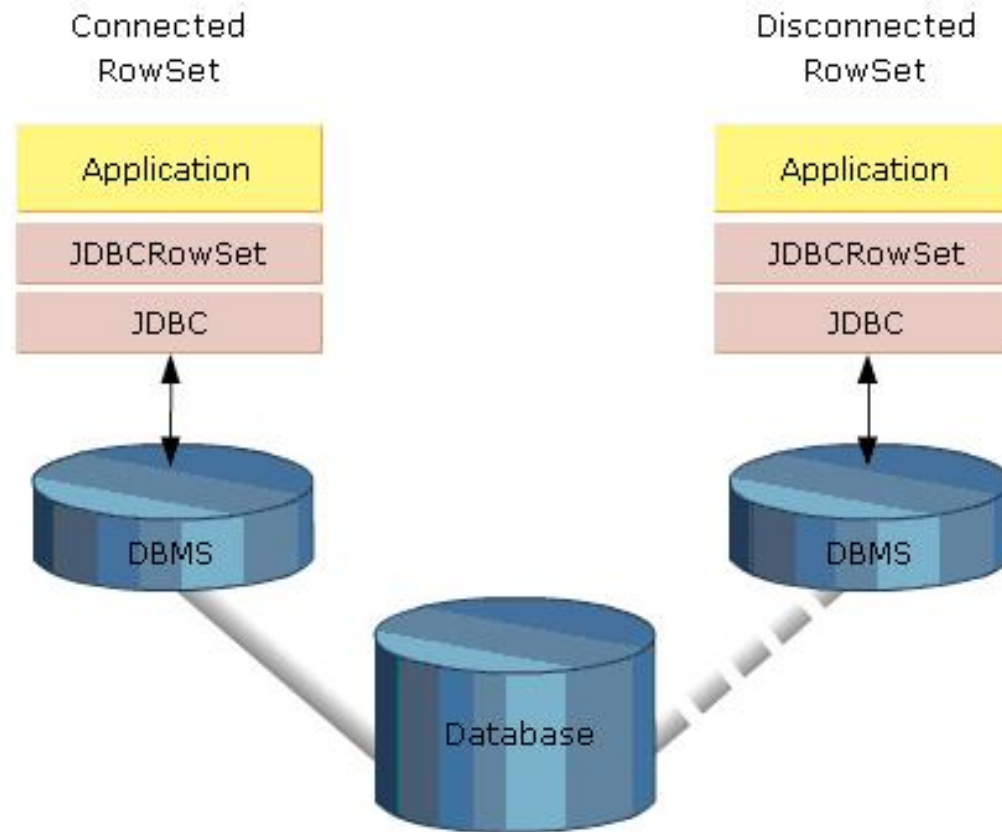
- RowSet is an interface in package javax.sql
- It is derived from the ResultSet interface.
- It typically contains a set of row from a source of tabular data like a result set.
- It is a JavaBeans component, it has features:
 - Properties
 - JavaBeans Notification Mechanism



Benefits of using "RowSet"

- The main features of using a RowSet are:
 - Scrollability
 - Updatability
- A Rowset object, being a JavaBeans component can be used to notify other registered GUI components of a change.

Different types of RowSet



Connected and Disconnected RowSets

Creating JdbcRowSet Objects

- By using the reference implementation constructor that takes a `ResultSet` object
- By using the reference implementation constructor that takes a `Connection` object
- By using the reference implementation default constructor
- By using an instance of `RowSetFactory`, which is created from the class `RowSetProvider`

Implementation of connected "RowSet"

```
RowSet rs = new JdbcRowSetImpl();
rs.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
rs.setUsername("system");
rs.setPassword("manager");
rs.setCommand("select * from dept");
rs.execute();
rs.afterLast();
while (rs.previous()) {
    System.out.println(rs.getInt(1) + "\t" + rs.getString(2));
}
```

Implementation of connected "RowSet"

```
Statement st = con.createStatement();
ResultSet rs = st.executeQuery(Select *
From Employee);
JDBCRowSet jRs = new
JDBCRowSetImpl(rs);
JDBCRowSet jrs2 = new JDBCRowSetImpl();
jrs2.setUsername("scott");
jrs2.setPassword("tiger");
jrs2.setUrl("jdbc:protocolName:datasourceName");
jrs2.setCommand("Select * from Employees");
jrs2.execute();
```

Implementation of disconnected “RowSet”

- A **CacheRowSet** object is an example of a disconnected **RowSet** object.
- There are two ways to create a **CachedRowSet** object:
 - Using a **default constructor**
 - Using the **SyncProvider** implementation

Create a CachedRowSet object: Using a default constructor

```
CachedRowSet crs = new CachedRowSetImpl();  
crs.setUsername("scott");  
crs.setPassword("tiger");  
crs.setUrl("jdbc:protocolName:datasourceName");  
crs.setCommand("Select * From Employee");  
crs.execute();
```

Create a CachedRowSet object: Using the SyncProvider implementation

```
CachedRowSet crs2 = new  
    CachedRowSetImpl("com.myJava.providers.HighAvailabilityProvid  
er");  
crs2.setUserName("scott");  
crs2.setPassword("tiger");  
crs2.setUrl("jdbc:protocolName:datasourceName");  
crs2.setCommand("Select * From Employee");  
crs2.execute();
```

Introduction to JDBCRowSet

- A JDBCRowSet object is derived from ResultSet object
- The uses of a JDBCRowSet object are:
 - To make a ResultSet object scrollable and thereby make better use of legacy drivers that do not support scrolling.
 - To use the ResultSet object as a JavaBeans component. This feature enables its use as a tool to select a JDBC driver in an application.

Using JDBCRowSet object

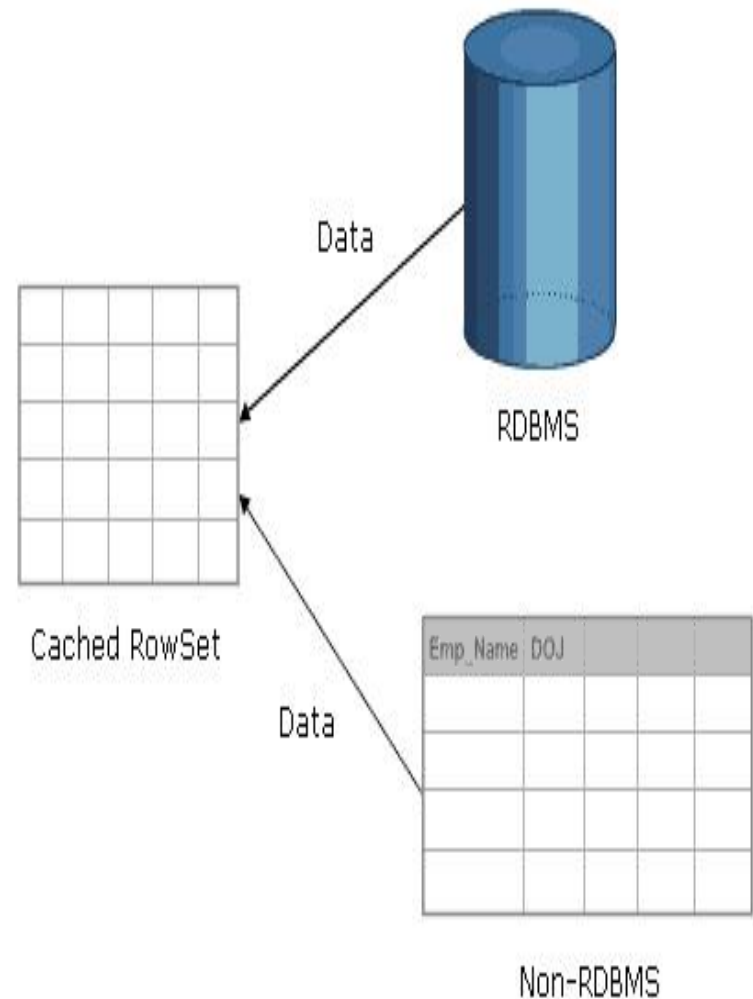
- A row of data can be updated, inserted and deleted in a way similar to an updatable ResultSet object.
- Any changes made to a JDBCRowSet object's data are also reflected on the DB.

CachedRowset (1)

- CachedRowSet stores/caches its data in memory so that it can operate on its own data rather than depending on the data stored in a DB.
- Disconnected RowSet objects are serializable. This enables a disconnected RowSet to be transmitted over a network to thin clients, such as PDA's or mobile phone.
- A CachedRowSet has all the capabilities of a connected RowSet.

CachedRowset (2)

- A row of data can be updated, inserted and deleted in a CachedRowSet object.
- Changes in data are reflected on the DB by invoking the `acceptChanges()` methods.



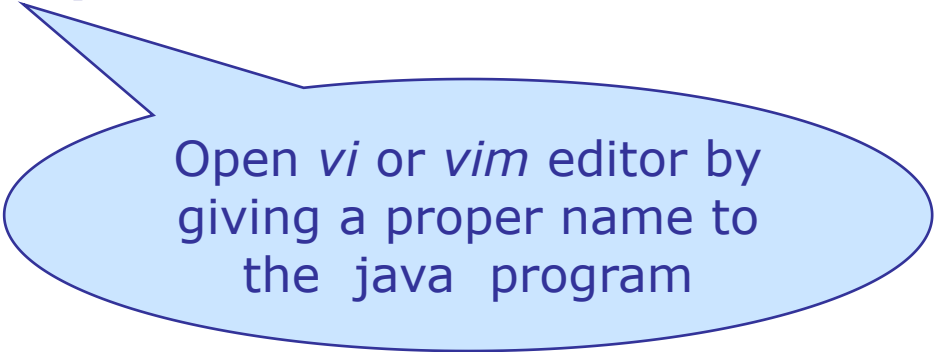
The Way Ahead

Java on Unix

Steps to be followed to run a program in Unix / Linux:

1. Go to the command prompt & type

```
$vi First.java
```



Open *vi* or *vim* editor by giving a proper name to the java program

```
class First{  
public static void main(String args[])  
{  
System.out.println("Hello Java World");  
}  
}
```

Java on Unix (Contd...)

2. Save the program and return to the command prompt
3. To compile this program, type the command:

```
$javac First.java
```

4. To run the program, type:

```
$java First
```

And, the Output will be:

```
Hello Java World
```

New features in Java

- Generics
- Enhanced for loop ("*foreach*")
- Autoboxing / Unboxing
- Type-safe Enumerations
- Variable Arguments
- Static Import

Generics

- We have to explicitly type cast objects from Collection according to requirements
- This conversion is unsafe as the compiler does not know its type and it may fail at runtime
- Generics provide a way to tell the compiler, the type of objects Collection carries
- Syntax of the Generics:

```
Collection <Type> obj = new <Type> Collectionclass();
```


Enhanced for loop - "*foreach*"

- A new *foreach* loop is introduced to simplify the iteration process for the collections

- Syntax of *foreach* loop:

```
for ( Objecttype obj : collection )
```

- All the objects of collection are assigned to *obj* in a sequence
- An Advantage - We don't have to type cast objects from the collection to the desired type

Autoboxing

Earlier Java versions

- To add an integer number to collection:
 - We have an ArrayList and 2 integers, a & b
 - We create new Integer objects
 - We then convert a & b to Integer objects and then add them in the ArrayList

Latest Java version

- This will automatically happen with *Autoboxing*
- Using collection objects like primitive variables is also possible through *Unboxing*

Enums in Java

- Enumerated types are the collection of values
- The standard way to represent an enumerated type in earlier version of Java was the `int Enum` pattern

For Example:

```
public static final int DAY_SUNDAY = 0;  
public static final int DAY_MONDAY = 1;
```

- In 5.0, the Java™ programming language gets linguistic support for enumerated types
- Java Enums look similar to C, C++ Enums
- Java Enums Syntax:

```
public enum enumName {Value1,.....ValueN;}
```

Variable Arguments in Java (varargs)

Earlier Java versions

- If a user wants to send an arbitrary number of values, there are 2 ways:
 - Create an array and put the values into it
 - Create some overloaded methods for the same

Java 5.0

- Sun has introduced the Variable Argument concept
- Java automatically takes the array of values passed to a method
- varargs syntax:

```
public returntype methodname(Object... arguments);
```

Variable Arguments in Java (varargs)

Tiger's variable arguments language feature makes it possible to call a method with a variable number of arguments

type ... variableName The ellipsis (...) identifies a variable number of arguments, and is demonstrated in the following summation method.

```
static int sum (int ... numbers)
{
    int total = 0;
    for (int i = 0; i < numbers.length; i++)
        total += numbers [i];
    return total;
}
```

Static Import

- To access static members, it is necessary to qualify references with the class they came from

Example: To access the PI member in the Math class we have to use Math.PI as shown:

```
double r = Math.PI *10;
```

- The static import declaration is analogous to normal import declaration
- Normal import declaration imports classes from packages, allowing them to be used without package qualification
- Static import declaration imports static members from classes, allowing them to be used without class qualification

Summary

In this session, we have covered:

- JDBC Architecture
- Types of Drivers
- Loading the drivers
- Connecting to Database
- Manipulate data in the Database
- Transaction Management
- Features of JDBC 2.0
- Using Java on UNIX
- New features in Java