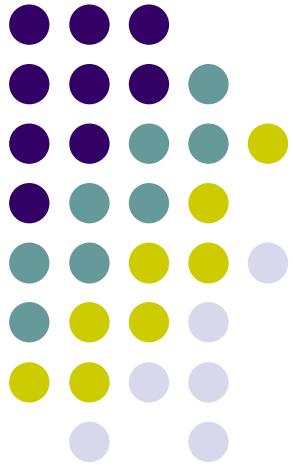
JDBC –
Java DataBase Connectivity

Ms. Deepthi S Narayan
Assistant Professor
Department of CA
PES University



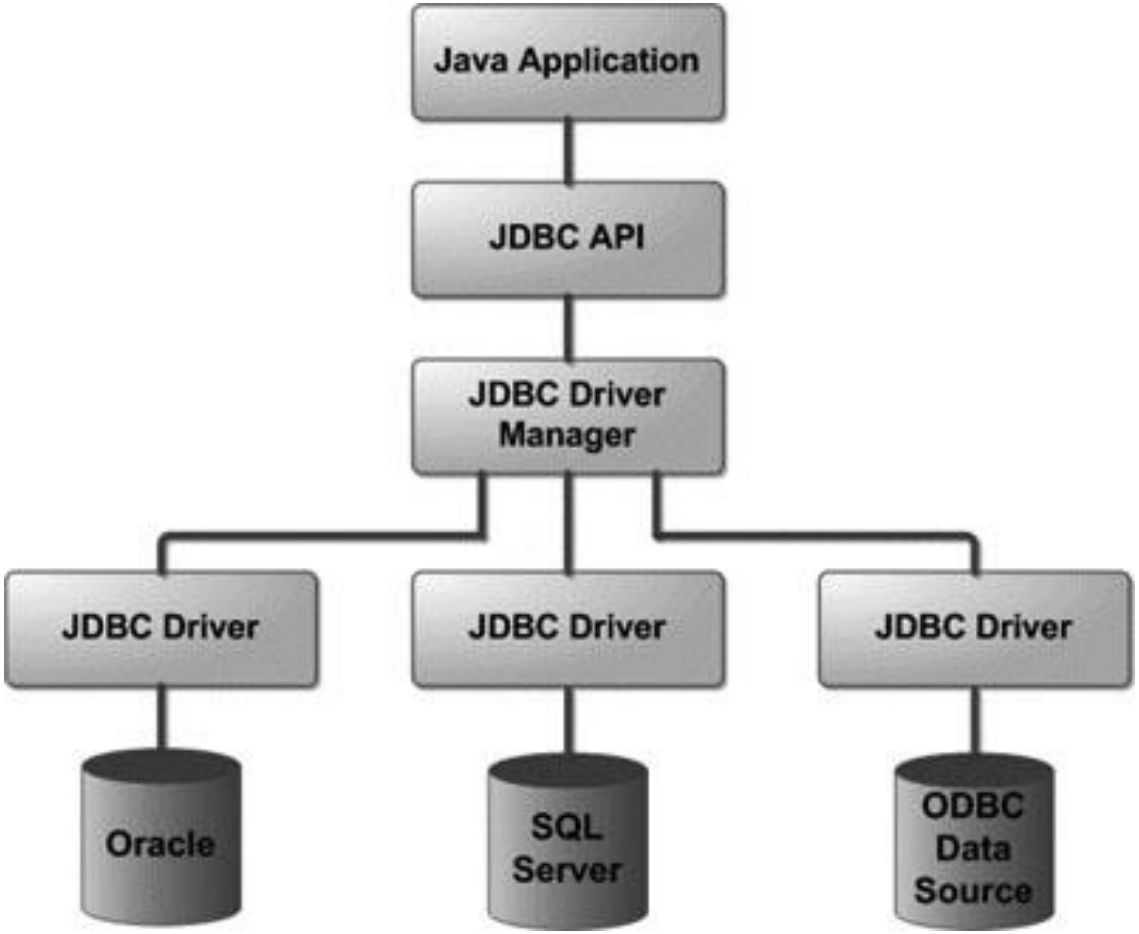
Pre-requisite

- Core Java Programming
- SQL - MySQL Database

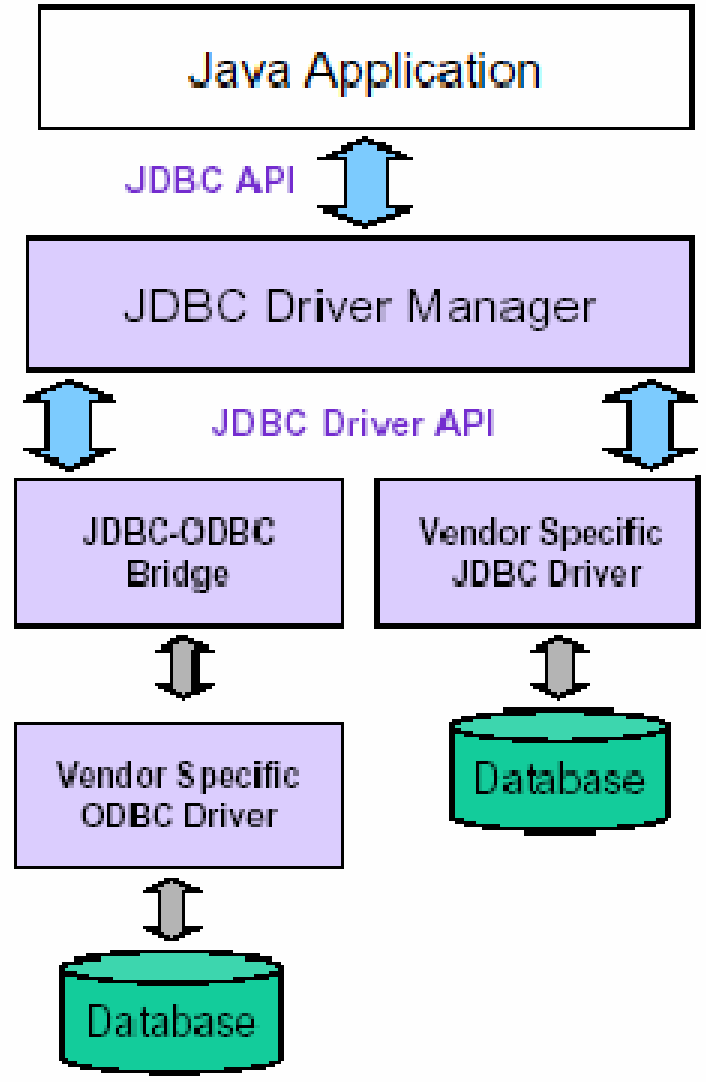
What is JDBC?

- Java Database Connectivity is a Java API that helps us to achieve the connectivity between Java and the database.
- Lets you access virtually any tabular data source from the Java programming language
- It is the **one and only** API in Java that helps us to achieve database connectivity.

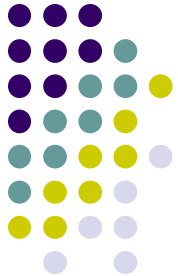
Architecture Diagram



General Architecture



Advantages of JDBC

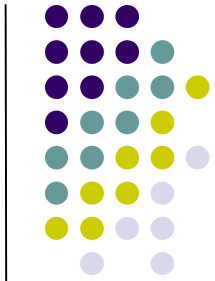


- It inherits all the advantages of Java
- JDBC is Database – independent
- Interaction with multiple databases simultaneously – possible
- JDBC helps us to acheive high performance through PreparedStatement and CallableStatement
- JDBC supports stored procedure.

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 the SQL queries in the database
- Viewing & Modifying the resulting records

JDBC Pre-requisites



1. Install DB Server(MySQL)
2. Create a DB in the above DB Server(Schema)

Ex: student

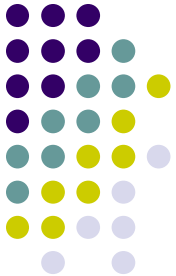
3. Create a table in the above database.

Ex: student

4. Insert some data into the above table.

5. Ready for DB interaction.

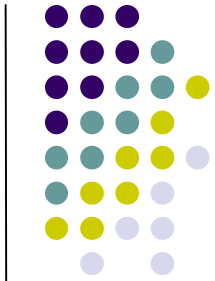
Necessary steps to work with JDBC



- 1.Load the **Driver**
- 2. Get the **DB Connection** via **Driver**
- 3. Execute **SQL Queries** via **Connection**
- 4. **Process results** returned by **SQL Queries**
- 5.**Close all JDBC objects.**

JDBC and ODBC Comparison

Two-tier & Three-tier models



There are 2 layers present in JDBC.

- JDBC API is the top layer. The API communicates with the JDBC driver manager API and sends it to various SQL statements.
- The manager should communicate with various third party drivers that actually communicate with the drivers.

JDBC Two-Tier Model

A Java application or applet talks directly to database in the two-tier model.

It requires a JDBC driver that can help to communicate with the particular database.

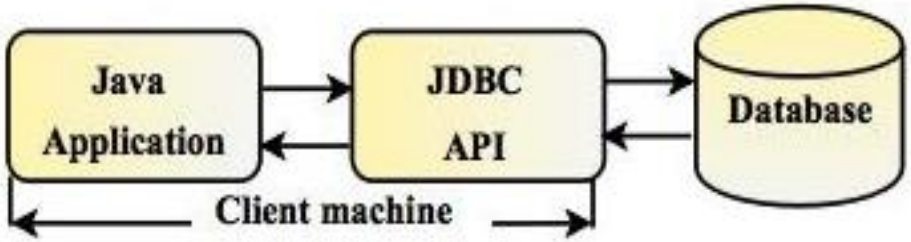


Fig: Two-tier Architecture of JDBC

The JDBC Package

import java.sql.*

Driver (I)

DriverManager (C)

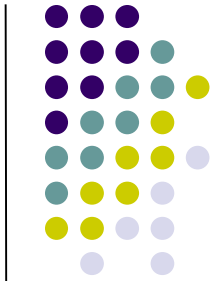
Connection (I)

ResultSet (I)

Statement (I)

PreparedStatement (I)

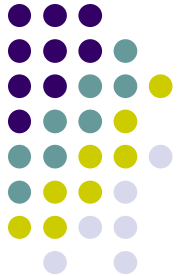
CallableStatement (I)



Three types of JDBC statements are there :

- Statement
- PreparedStatement
- CallableStatement

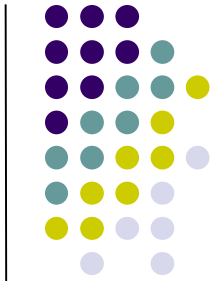
Types of network connections



Direct – It is a connection that a JDBC client makes directly to the DBMS server, which may be remote

Indirect – It is a connection that a JDBC client makes to a middleware process that acts as a bridge to the DBMS server.

JDBC Drivers



There are 4 types of JDBC drivers :

- Type 1 – JDBC-ODBC bridge

Provides JDBC access via most ODBC drivers.

- Type 2 – Native API driver

Converts JDBC calls into calls on the client API for Oracle, Sybase, DB2, MYSQL, etc.

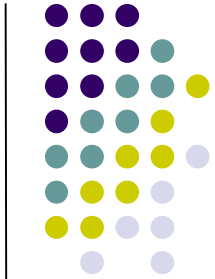
- Type 3 – Network protocol driver

Translates JDBC calls into DBMS independent net protocol, which is then translated to a DBMS protocol by a server.

- Type 4 – Native protocol driver (Middleware driver)

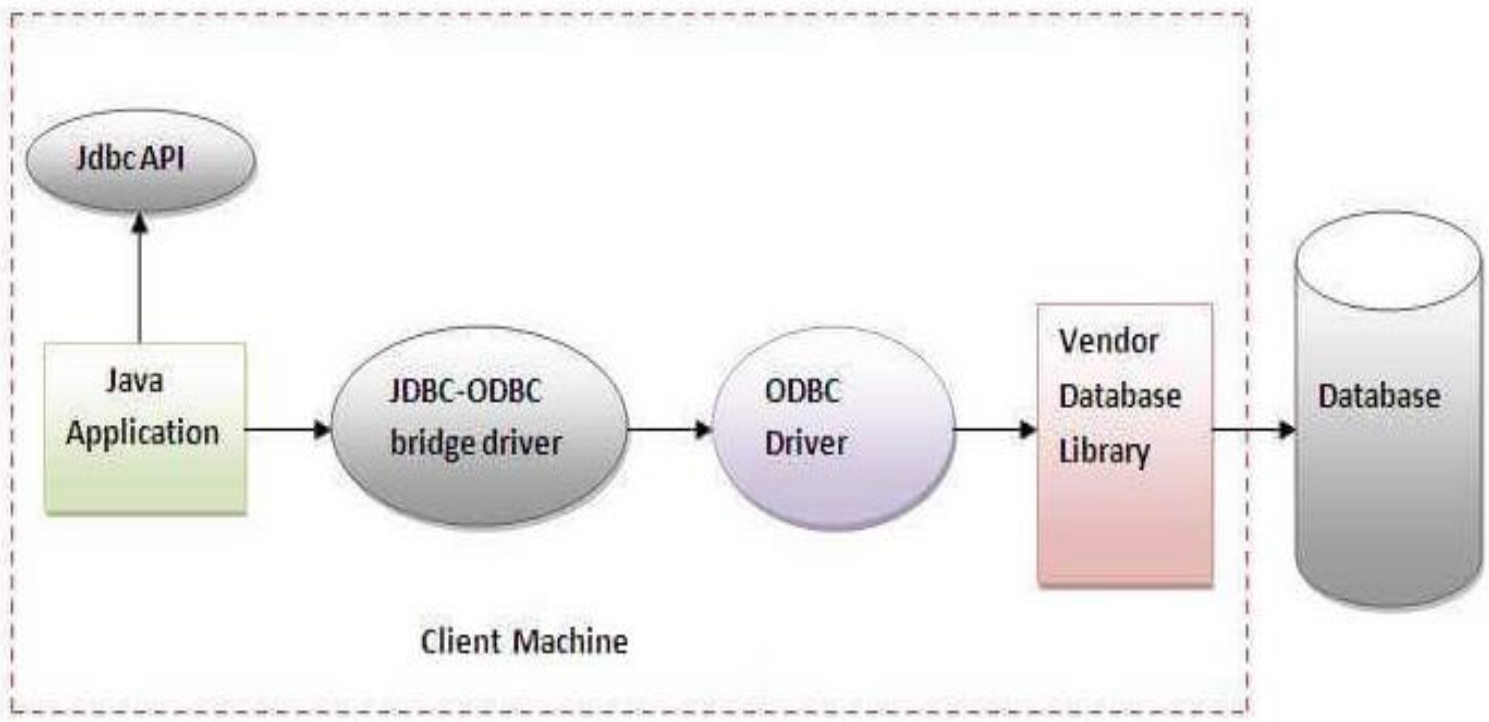
Converts JDBC calls into network protocol used by DBMS directly.

The four categories of drivers and their properties

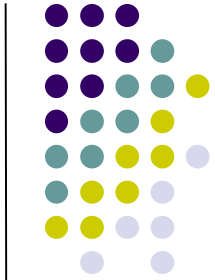


Driver Category	All Java	Network Connection
JDBC-ODBC bridge	No	Direct
Native API	No	Direct
Net Protocol	Client – Yes, Server – may be	Indirect
Native Protocol	Yes	Direct

Type 1 - JDBC-ODBC bridge



Type 1 - JDBC-ODBC bridge



PROS

Easy to use

Any DB is supported

CONS

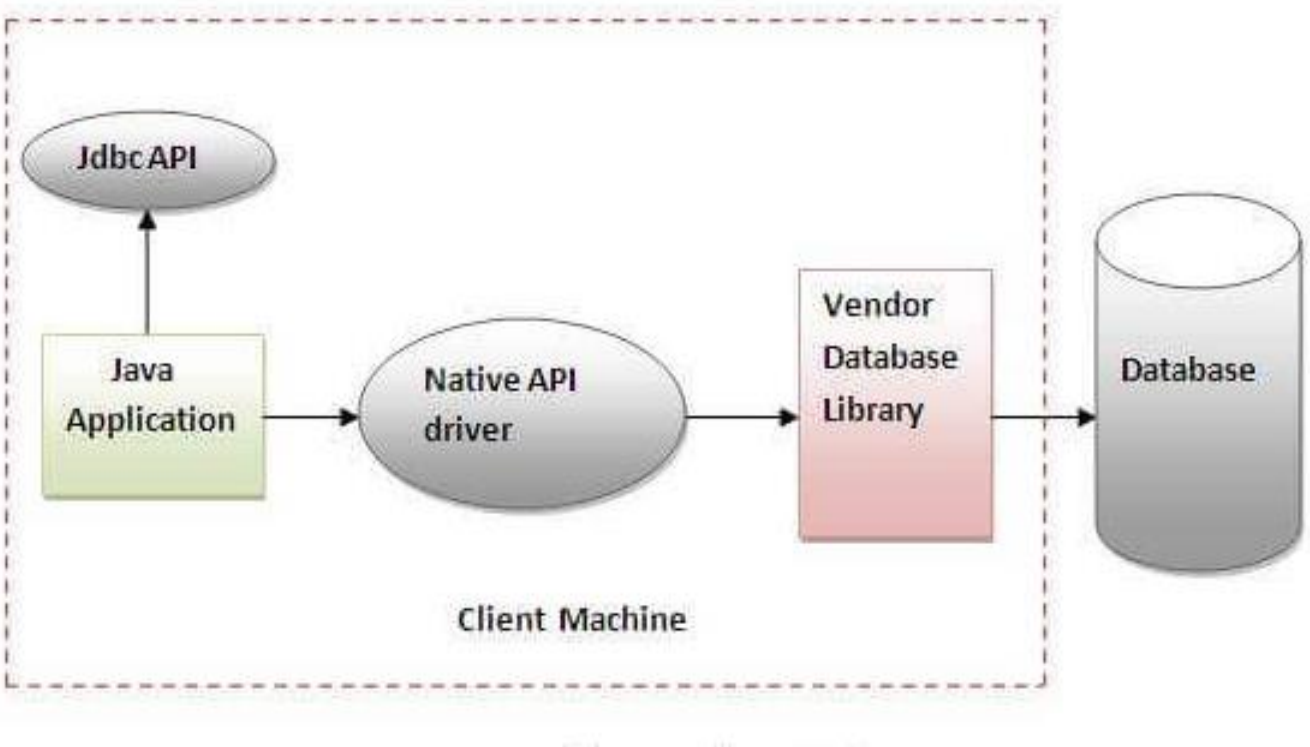
Performance in not efficient

Client-side installation required (ODBC Driver)

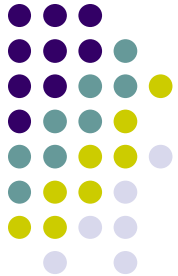
Platform dependent

Not suitable for Applets(Why?)

Type 2 - Native API driver



Type 2 - Native API driver



PROS

Faster than Type 1 driver

CONS

Client-side library is not available for all DB’s

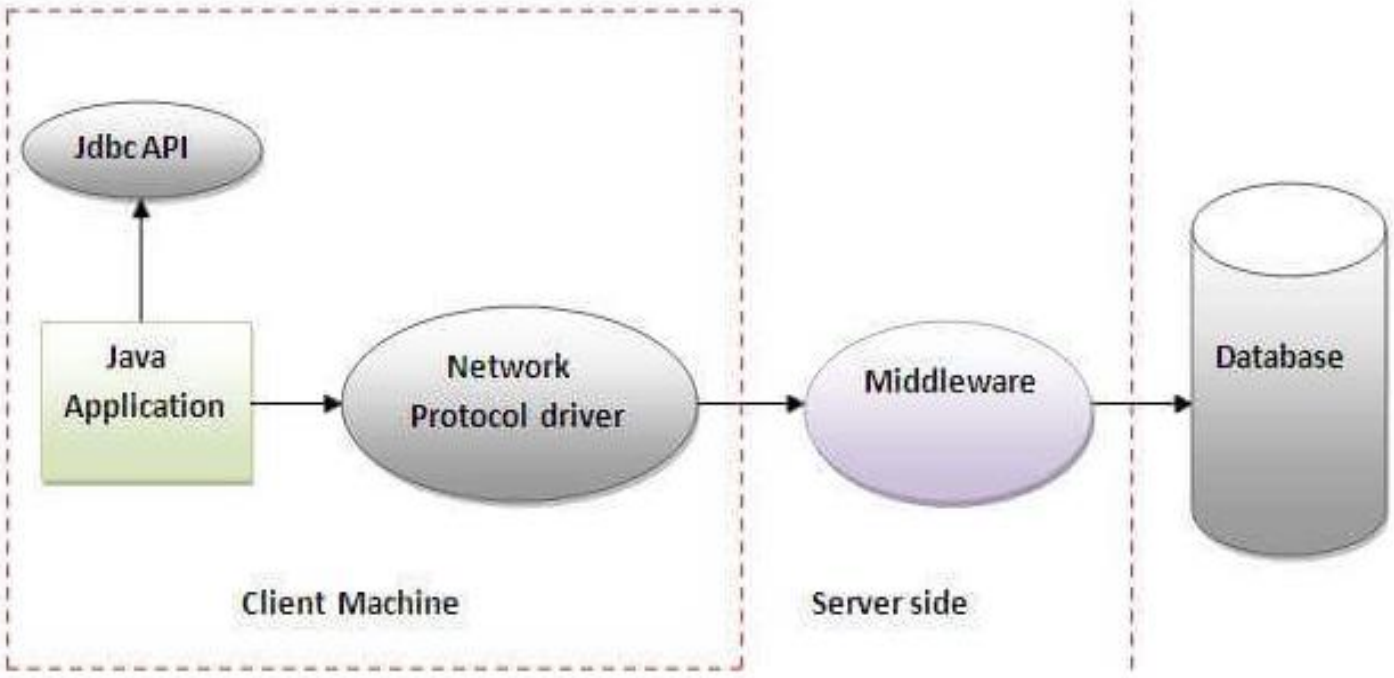
Vendor client library needs to be installed

Platform dependent

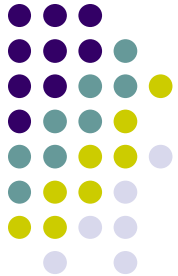
Not thread-safe

Comparitively low performance

Type 3: Network protocol driver



Type 3: Network protocol driver



PROS

No additional library installation is required on client machine.

No changes are required at client for any DB

Single driver can handle any DB, provided the middleware supports it

Platform independent.

CONS

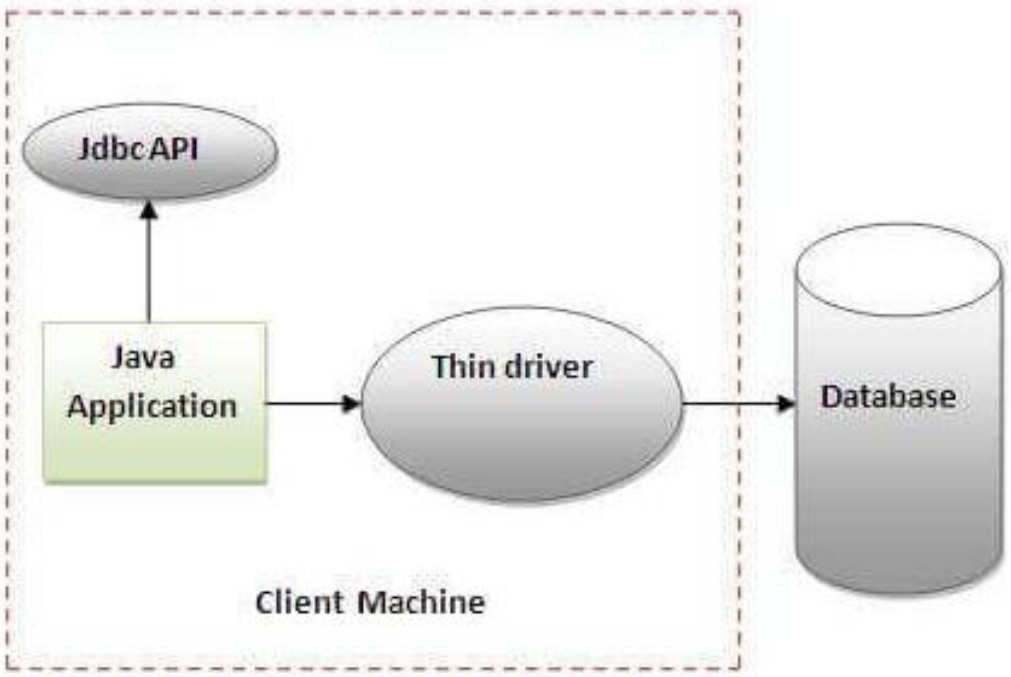
Server-side installation is required

Performance will be slow (Why?)

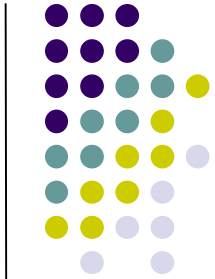
Maintenance of Network Protocol Driver is costly

Requires DB-specific coding in middleware

Type 4: Native protocol driver (Middleware driver)



Type 4: Native protocol driver (Middleware driver)



PROS

Platform independent

No client-side or server-side installation is required

Application connects directly to the DB Server

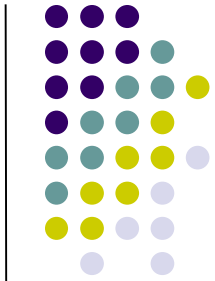
Performance is faster

JVM manages every aspect

CONS

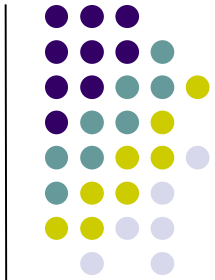
Drivers are DB-dependent

When-What?



- If you are accessing one type of database such as Oracle, SQL Server, MySQL etc. then the preferred type is 4
- If your Java application is accessing multiple types of databases at the same time, type 3 is 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 it is typically used for development and testing purposes only

Driver Interface



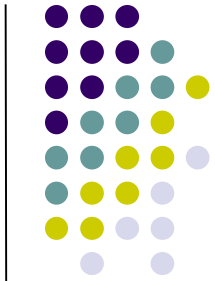
Used to create connection objects

Driver is an additional software component required by JDBC to interact with DB.

All drivers supply a class that implements **Driver** interface.

Drivers are provided by DB vendor and they are DB dependent.

Driver Interface - Methods



boolean acceptsURL(String url)

Connection connect(String url, Properties info)

int getMajorVersion()

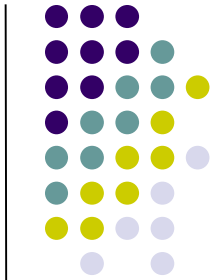
int getMinorVersion()

Logger getParentLogger()

DriverPropertyInfo[] getPropertyInfo(String url, Properties info)

boolean jdbcCompliant()

DriverManager



DriverManager manages the drivers.

DriverManager Class helps to load Driver for any given connection request.

When a **Driver** class is loaded, it creates an instance of itself and registers it with the **DriverManager** class.

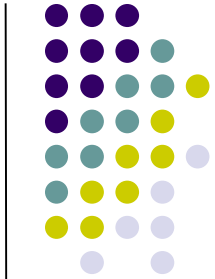
getConnection() – overloaded versions??

2 ways to load the driver into the program

1. Create Driver object and then register using DriverManager
2. Using Class.forName(“…..”)

Which one is better and why?

Connection Interface



extends Wrapper, AutoCloseable

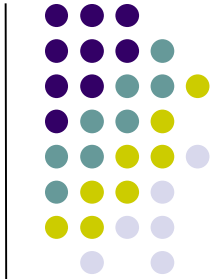
A connection (session) with a specific database.

SQL statements are executed and results are returned within the context of a connection.

Makes a connection to a specific database using a specific driver

con = DriverManager.*getConnection(" …")*

Statement Interface



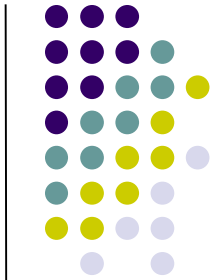
extends Wrapper, AutoCloseable

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

Incase of queries, executing a statement brings data into a ResultSet

stmt = con.createStatement();

PreparedStatement interface



extends Statement

An object that represents a precompiled SQL statement.

Used to execute dynamic queries.

Used in a scenario, where same SQL query needs to be executed many times – high performance

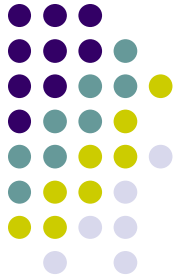
Callable Statement

extends PreparedStatement

The interface used to execute SQL stored procedures.

A Stored procedure is a group of SQL queries that performs a particular task. As the name implies, it is stored in the database – high performance

Callable Statements



Predefined sequences of SQL commands

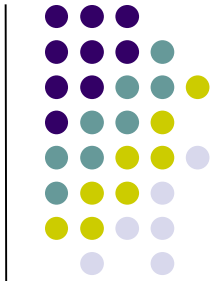
Used to execute Stored Procedures

Extends PreparedStatement

Executed as -

```
String query = "{call storedProcedure}";
CallableStatement cstmt =
   con.prepareCall(query);
```

What is a Stored Procedure?



A stored procedure is a subroutine available to applications that access a relational database system.

Similar to user-defined function

Stored Procedures are batch of SQL statements compiled into a single execution plan.

Syntax

delimiter $$

create procedure procedureName()

begin

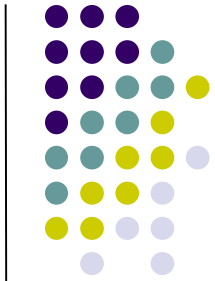
SQL query

end$$

Called using -

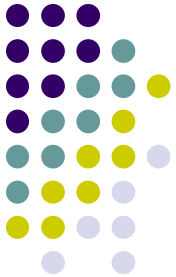
call procedureName()

Stored Procedure - Example



```
use STUDENTS DB
delimiter &
create procedure getAllStudentsInfo()
begin
select * from studenttab;
end &
call getStudentsInfo()
```

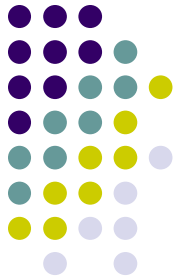
Another Example



```
DELIMITER $$
DROP PROCEDURE IF EXISTS
   `getStudentName` $$
CREATE PROCEDURE `getStudentName`
 (IN sid INT, OUT sname VARCHAR(255))
BEGIN
   SELECT studentname INTO sname
   FROM student_table
   WHERE studentid = sid;
END $$
```

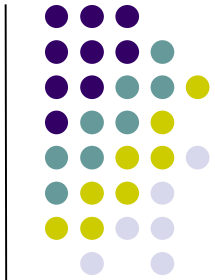
Example for insert and update

```
delimiter &
create procedure studentUpsert
(in_regno int, in_fname varchar(50), in_lname
   varchar(50))
begin
declare regno_count int;
select count(*) into regno_count from
   students_info where regno=in_regno;
```



```
if regno_count > 0 then
update studentinfo set firstname = in_fnm,
   middlename = in_mnm, lastname = in_lnm
   where regno =  in_regno;
else
 insert into students_info values(in_regno,
   in_fnm, in_mnm, in_lnm);
end if;
end$
```

ResultSetMetaData Interface



An object that can be used to get information about the types and properties of the columns in a **ResultSet** object

Ex:

```
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM TABLE2");
    ResultSetMetaData rsmd = rs.getMetaData();
    int numberOfColumns = rsmd.getColumnCount();
    boolean b = rsmd.isSearchable(1);
```

DatabaseMetaData Interface

This interface provides information regarding the database itself.

Ex: Version, Table names, Supported functions etc.,.

Ex: A tool might use the method getTypeInfo to find out what data types can be used in a CREATE TABLE statement

Commonly used methods in DatabaseMetaData Interface

public String **getDriverName**()throws SQLException: it returns the name of the JDBC driver.

public String **getDriverVersion**()throws SQLException: it returns the version number of the JDBC driver.

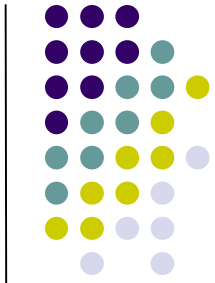
public String **getUserName**()throws SQLException: it returns the username of the database.

public String **getDatabaseProductName**()throws SQLException: it returns the product name of the database.

public String **getDatabaseProductVersion**()throws SQLException: it returns th

public ResultSet **getTables**(String catalog, String schemaPattern, String tableNamePattern, String[] types)throws SQLException: it returns the description of the tables of the specified catalog. The table type can be TABLE, VIEW, ALIAS, SYSTEM TABLE, SYNONYM etc.e product version of the database.

JDBC URL Examples



1) Oracle

jdbc:oracle:thin:myuser/mypwd@myserver:1521:mydb

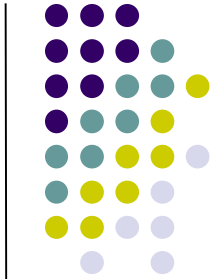
2) Microsoft SQL Server

jdbc:microsoft:sqlserver://myserver:1433;databasename=mydb;user:myuser;password:mypassword

3) MySQL

jdbc:mysql://myserver:3306/mydb?user=myuser&password=mypassword

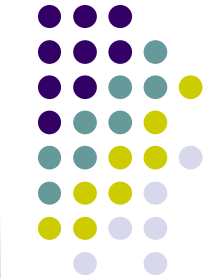
1. Load the Driver and establish a connection



- **import java.sql.*;**
- **Load the vendor specific driver**

  Class.forName("com.mysql.jdbc.Driver");
  - What do you think this statement does, and how?
  - Dynamically loads a driver class, for MySql database
- **Make the connection**

  Connection con = DriverManager.getConnection(
  jdbc:mysql://localhost:3306","username", "password");
  - What do you think this statement does?
  - Establishes connection to database by obtaining a *Connection* object



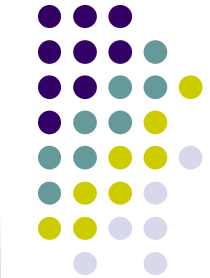
```
try {
Connection con;
String dburl = “jdbc:mysql://localhost:3306”
String username =” ”;
String password=“ “;
con = DriverManager.getConnection(dburl, username, password);
System.out.println(“*****Connected to Database***”);
} catch(SQLException e) {
System.out.println(“SQLException:” + e.getMessage());
System.out.println(“SQLState:” + e.getSQLState());
System.out.println(“Vendor Error:” + e.getErrorCode());
}
```

2. Create JDBC statement(s)

- Statement stmt = con.createStatement() ;

Creates a Statement object for sending SQL statements to the database

3. Executing SQL Statements



- String query = "insert into student(usn,fname) values(123,Akash)"

  //What does this statement do?

stmt.**executeUpdate**(query);

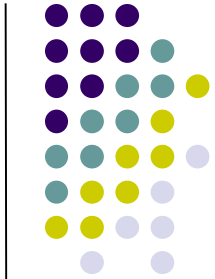
Differentiate between executeUpdate(), executeQuery() and execute()

4. Get and Process the ResultSet

```
String query = "select * from student where
   usn=123";

ResultSet rs = stmt.executeQuery(query);
//What does this statement do?
while (rs.next()) {
   int ssn = rs.getInt("SSN");
   String name = rs.getString("NAME");
   int marks = rs.getInt("MARKS");
}
```

5. Close all JDBC objects



- rs.close();
- stmt.close();
- con.close();

MyFirstJDBC.java

Using Statement Object/ Interface

Refer API

1. executeQuery()
2. executeUpdate()
3. execute()

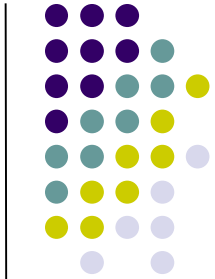
Using a PreparedStatement Object

PreparedStatement contains an SQL statement that has been pre-compiled.

Most commonly used for SQL statements with parameters

PreparedStatement pstmt = con.prepareStatement("update Student set sname=? where usn like ?");

ResultSet



SQL Query always produces two kinds of results

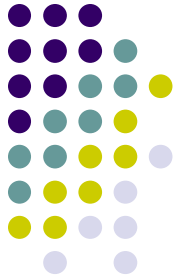
1) No. of rows affected count
2) DB Results

ResultSet can be extracted as follows –

- Move to desired row by calling necessary resultset methods
- Retrieve the desired column values

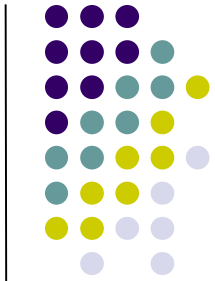
NOTE: By default, ResultSet object can be moved forward only and it is not updatable.

Scrollable ResultSets



```
ResultSet rs = stmt.executeQuery("select * from
    EMPLOYEE");
while(rs.next()) {}
rs.absolute(5);
rs.relative(-2);
rs.relative(4);
rs.previous();
int rownumber = rs.getRow();
rs.moveAfterLast();
while(previous()) { }
```

Batch Updates



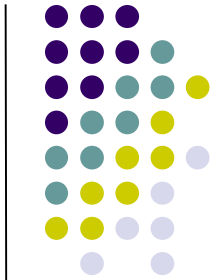
A ***JDBC batch update*** is a batch of updates grouped together, and sent to the database in one *batch*, rather than sending the updates one by one.

There are two ways to execute a *JDBC batch update*:

Using a **Statement**

Using a **PreparedStatement**

Batch Updates

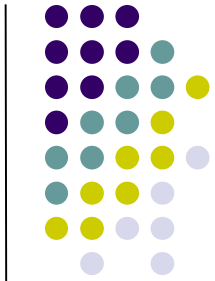


JDBC API provides Batch Processing feature through which we can execute bulk of queries in one go for a database.

JDBC API supports batch processing through **Statement** and **PreparedStatement** **addBatch**() and **executeBatch**() methods.

Batch Processing is **faster** than executing one statement at a time as the number of database calls are less.

Statement Batch Updates



1) Add the SQL statements to be executed in the batch, using the **addBatch**() method.
2) Execute the SQL statements using the **executeBatch().** The **int[] array** returned by the executeBatch() method is an array of int telling how many records were affected by each executed SQL statement in the batch.

Statement Batch Updates Example

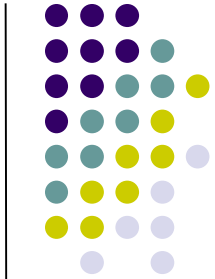
Improves Performance. How?

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.addBatch("Insert into employees values(1000,
   'Joe Jones')");
stmt.addBatch("Insert into department values(260,
   'Shoe)");
stmt.addBatch("Insert into emp_dept(1000, 260)");
int [] updateCount = stmt.executeBatch();
```

PreparedStatement Batch Updates

1) A PreparedStatement is created from an SQL statement with question marks.
2) Each set of parameter values are inserted into the preparedStatement, and the **addBatch()** method is called.
3) The **executeBatch()** method is called, which executes all the batch updates.

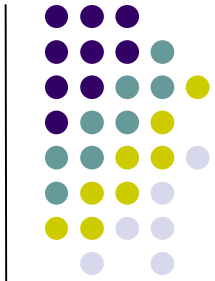
PreparedStatement Batch Update Example



```
String sql = "update student set firstname=? , lastname=?
    where usn=?";
PreparedStatement preparedStatement = null;
try{
    preparedStatement = connection.prepareStatement(sql);
    preparedStatement.setString(1, "Gary");
    preparedStatement.setString(2, "Larson");
    preparedStatement.setLong(3, 123);
    preparedStatement.addBatch();
```

cont…

PreparedStatement Batch Update Example



```
preparedStatement.setString(1, "Stan");
    preparedStatement.setString(2, "Lee");
    preparedStatement.setLong  (3, 456);
    preparedStatement.addBatch();
    int[] affectedRecords =
    preparedStatement.executeBatch();
} finally {
    if(preparedStatement != null) {
        preparedStatement.close();
    }
```

Data Types and JDBC

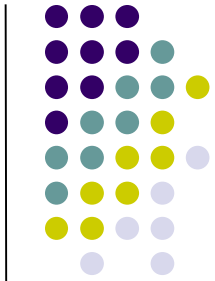
java.sql package and javax.sql package

1. Convenient use of advanced data types.
2. ResultSet with scrollable cursors.
3. Batch updates

Mapping Relational Data to Java Objects

1. Methods on the ResultSet class for retreiving SQL select results as Java types.
2. Methods on the PreparedStatement class for sending Java types as SQL statement parameters.
3. Methods on the CallableStatement class for retrieving SQL parameters as Java types.

Mapping SQL types to Java types



Ex: Large binary values

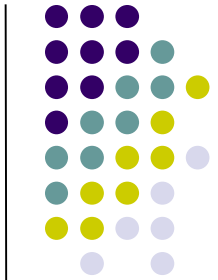
Oracle – LONG RAW

Sybase – IMAGE

Informix – BYTE

DB2 – LONG VARCHAR FOR BITDATA

java.sql.Types API to be referred



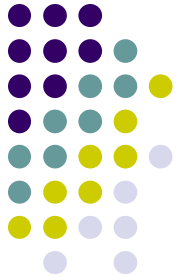
A standard mapping from the JDBC database types to Java types is defined by the JDBC.

Standard mappings -

Ex: JDBC INTEGER – Java int

CHAR, VARCHAR, LONG VARCHAR - String

Basic JDBC Datatypes

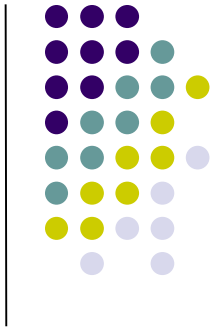


CHAR, VARCHAR and LONG VARCHAR

CHAR – small, fixed-length character string.

VARCHAR – small, variable length character string.

LONG VARCHAR – large, variable length character string.

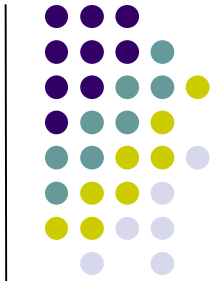


How to handle fixed length SQL strings of type CHAR(n)?

The ResultSet.getString() allocates and returns a new String object.

This is used to handle CHAR, VARCHAR and LONGVARCHAR fields.

LONG VARCHAR - ResultSet.getAsciiStream() and ResultSet.getCharacterStream()



BINARY, VARBINARY & LONG VARBINARY

BINARY – small, fixed-length binary value.

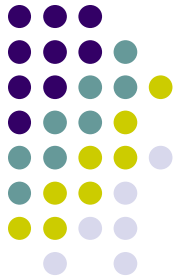
VARBINARY – small, variable length binary value.

LONG VARBINARY – large, variable length binary value.

Ex: A 12-byte binary type is defined by

BINARY(12).

Limits to 254 bytes only.



NOTE: The consistency between SQL type name corresponding to JDBC LONGVARBINARY type is not found.

BINARY, VARBINARY, LONGVARBINARY can all be expressed identically as byte arrays.

ResultSet.getBytes() is used to retreive BINARY & VARBINARY values.

ResultSet.getBinaryStream() helps to handle LONGVARBINARY values.

BIT – A single bit value is represented by JDBC type BIT that can be zero or one.

Java mapping – boolean

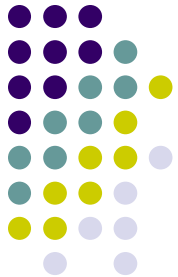
TINYINT – An 8-bit integer value between 0 & 255 that may be signed or unsigned.

Java mapping – byte or short

Note : 8-bit byte represents signed value from

-128 to 127

16-bit short represents TINYINT values better



SMALLINT – 16-bit signed integer value between -32768 to 32767

Java mapping – short

INTEGER – 32-bit signed integer value between -2147483648 & 2147483647

Java mapping – int

BIGINT – 64-bit signed integer value between -9223372036854775808 to 9223372036854775807

Java mapping - long

REAL – A single precision floating point number that supports 7 digits of mantissa.

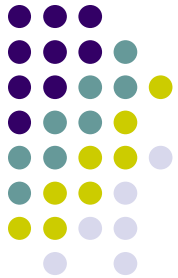
Java mapping – float

DOUBLE – A double precision floating point number that supports 15 digits of mantissa.

Java mapping – double

FLOAT – Same as DOUBLE

Java mapping - double



DECIMAL & NUMERIC

Both represent fixed precision values.

Java mapping – java.math.BigDecimal

ResultSet.getBigDecimal is recommended method.

DATE, TIME & TIMESTAMP

DATE – Represents a date consisting of day, month & year.

TIME – Represents a time consisting of hours, minutes and seconds

TIMESTAMP – DATE + TIME + nano second

Java mapping – represented by three sub classes of java.util.Date

DATE – java.util.Date with hour, minute, second and millisecond set to zero.

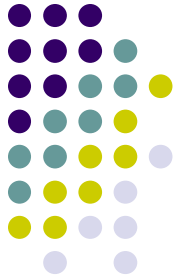
TIME – java.sql.Time (1970 January 1st)

TIMESTAMP – java.sql.Timestamp

National CharacterSet Support

The JDBC driver provides support for the JDBC 4.0 API, which includes new national character set conversion API methods.

This support includes new setter, getter, and updater methods for NCHAR, NVARCHAR, LONG NVARCHAR, and NCLOB JDBC types.



NCHAR – The JDBC type NCHAR is equivalent to SQL type NCHAR (2000 characters)

The NCHAR datatype stores fixed-length character strings that correspond to the national character set

public static final int NCHAR

NVARCHAR - The JDBC type NVARCHAR is equivalent to SQL type NVARCHAR (4000 characters)

public static final int NVARCHAR

LONGVARCHAR - The JDBC type LONGVARCHAR is equivalent to SQL type LONGVARCHAR

public static final int LONGVARCHAR

NCLOB – The JDBC type NCLOB is basically equivalent to the SQL type NCLOB

public static final int NCLOB

SQLXML – The JDBC type SQLXML is basically equivalent to SQL type SQLXML

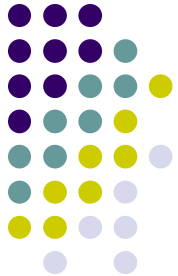
public static final int SQLXML

ROWID – equivalent to SQL type ROWID

public static final int ROWID

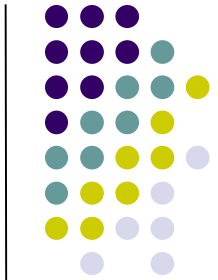
A RowId object represents an address to a row in a database table

Advanced JDBC Data types



Predefined types – BLOB, CLOB, ARRAY and REF

UDT – STRUCT, DISTINCT, JAVA_OBJECT

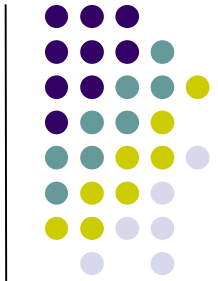


BLOB (Binary Large Object) – Maps to BLOB interface in Java.

A BLOB can hold a variable amount of data. The four BLOB types are TINYBLOB, BLOB, MEDIUMBLOB, and LONGBLOB.

CLOB (Character Large Object) – Maps to CLOB interface in Java

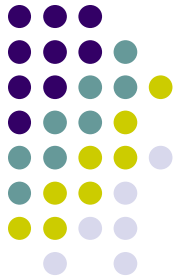
This data type is appropriate for storing text-oriented information where the amount of information can grow beyond the limits of a regular VARCHAR data type (upper limit of 32K bytes).



ARRAY - Maps to ARRAY interface in Java.

DISTINCT – The standard mapping for a DISTINCT type is the Java type to which the base type of a DISTINCT object would be mapped.

Ex: DISTINCT type based on CHAR would be mapped to a String object.



STRUCT – The attributes of STRUCT may be SQL data-type, built-in or user-defined.

Maps to Struct Object in Java

Example -

```
CREATE TYPE EMP_DATA(
    SSN Number(9),
    FirstName VARCHAR(20),
    LastName VARCHAR(20),
    Salary NUMBER(9,2));
```

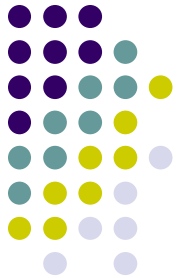
REF – REF<structured type> - logically points to an instance of SQL struct type

Maps to Ref interface in Java

JAVA_OBJECT – Makes it easier to use objects in the Java Programming language as values in a database

ResultSet.getObject, ResultSet.updateObject

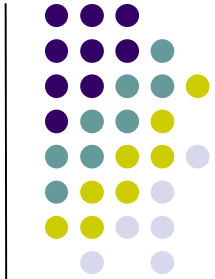
PreparedStatement.setObject



Ex:

```
ResultSet rs = stmt.executeQuery("Select
   ENGINEERS from PERSONNEL");
while(rs.next()) {
Engineer eng =
   (Engineer)rs.getObject("Engineers");
System.out.println(eng.lastName+
   ","+eng.firstName);
```

New Features introduced in JDBC 4.0



Auto-loading of JDBC Drivers

Enhancements in Connection Management

Support for ROWID SQL Type

DataSet Implementation of SQL using Annotations

Enhancements in SQLExceptions Handling

Support for SQLXML

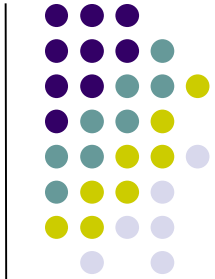
Retrieving Column Data for Specified Data types

Two Overloaded forms -

xxxType rs.getXXX(String columnname)

XxxType rs.getXXX(int columnPosition)

Working with null values



NULL is a special value in SQL.

Its not empty string for text or zero for a numeric

Null = No data

Example -

select * from STUDENTS.studenttab where age IS NULL

Working with Special Data types

ResultSet class has various methods to access the special data types

Date

Time

Timestamp

BigNumbers – NUMERIC and DECIMAL