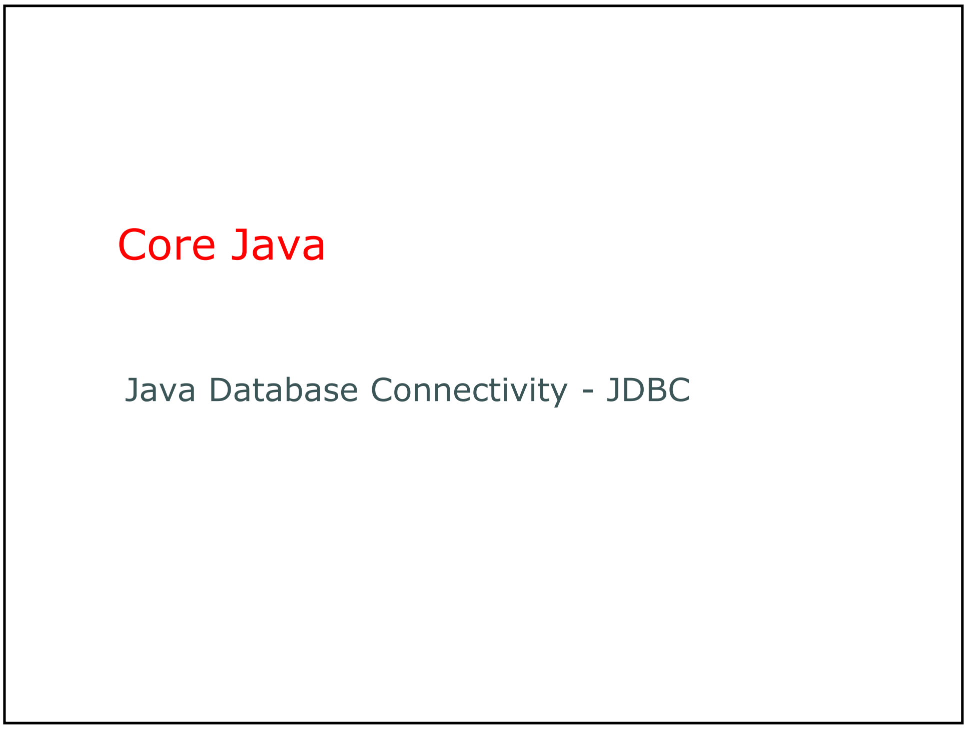
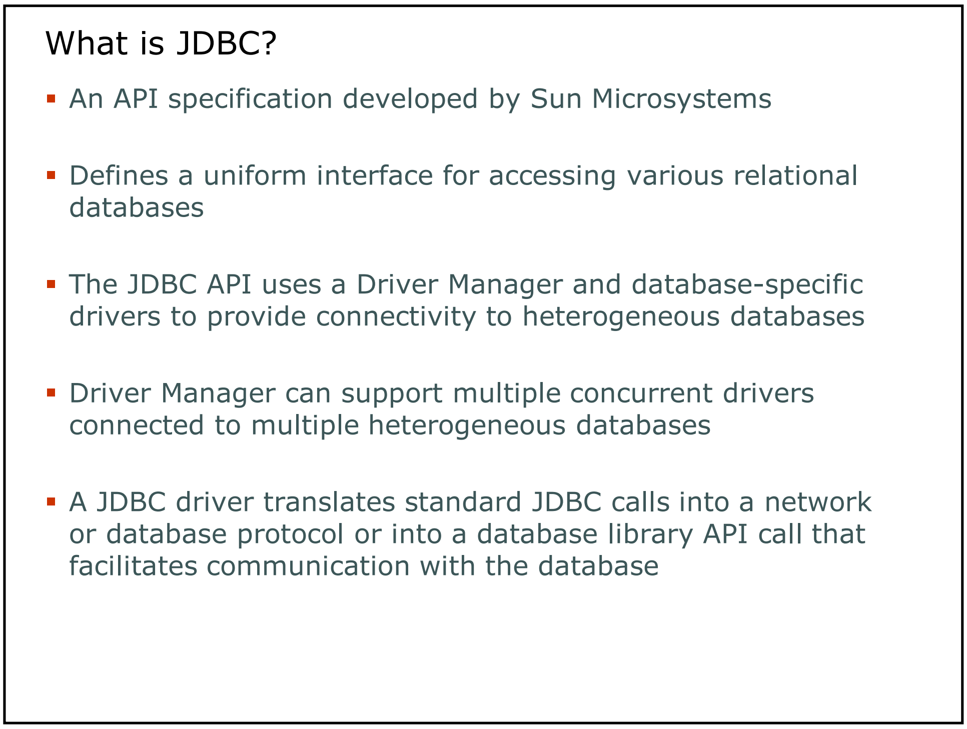


Core Java

Java Database Connectivity - JDBC

1

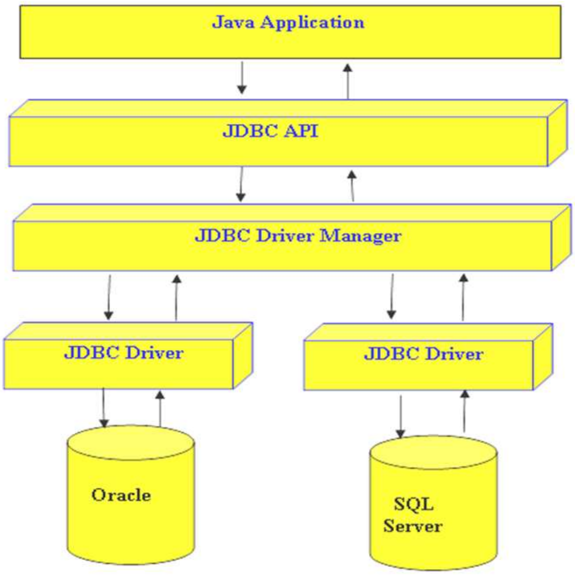


What is JDBC?

- 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

2

JDBC Architecture



3

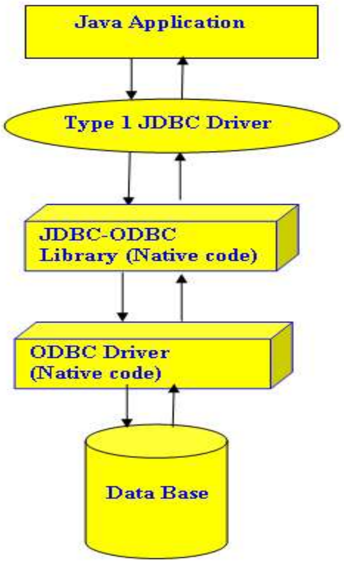
Types of JDBC Drivers

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

4

Type 1: JDBC-ODBC Bridge Driver



5

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

- Act as a "bridge" between JDBC & database connectivity mechanism like ODBC

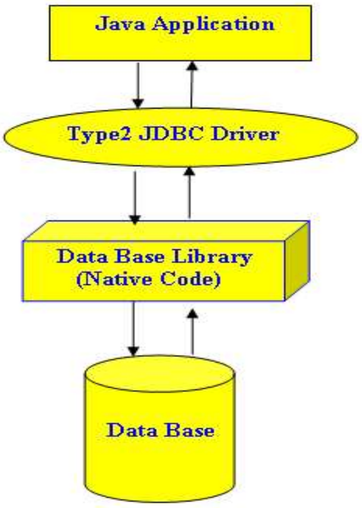
- The bridge provides JDBC access using most standard ODBC drivers

- 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

6

Type 2: Java to Native API Driver



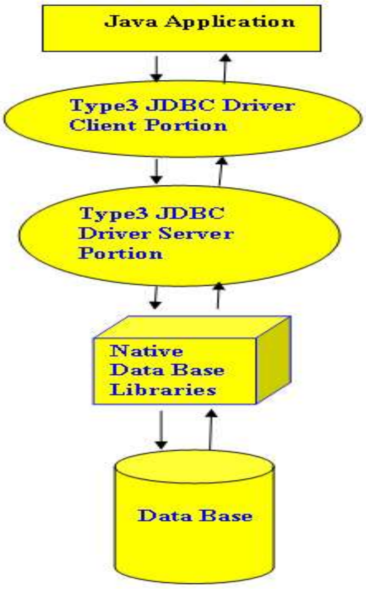
7

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

8

Type 3: Java to Network Protocol Driver


Java Application
Type3 JDBC Driver Client Portion
Type3 JDBC Driver Server Portion
Native Data Base Libraries
Data Base

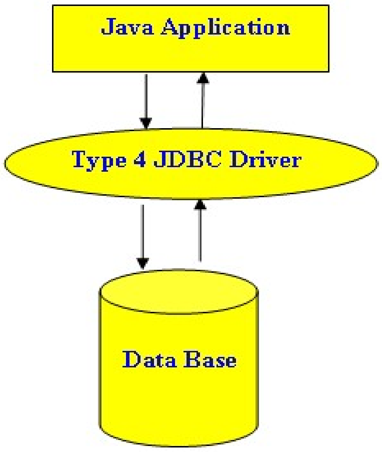
9

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

- Pure Java drivers that use a proprietary network protocol to communicate with JDBC middleware on the server
- The middleware then translates the network protocol to database-specific function calls
- 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

10

Type 4: Java to Database Protocol Driver



Java Application

Type 4 JDBC Driver

Data Base

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

- Pure Java drivers that implement a proprietary database protocol to communicate directly with the database
- Like Type 3 drivers, they do not require native database libraries and can be deployed over the Internet without client installation
- One drawback is that they are database specific
- Communicate directly with the database engine rather than through middleware or a native library
- Usually the fastest JDBC drivers available
- Directly converts java statements to SQL statements

Loading the Driver

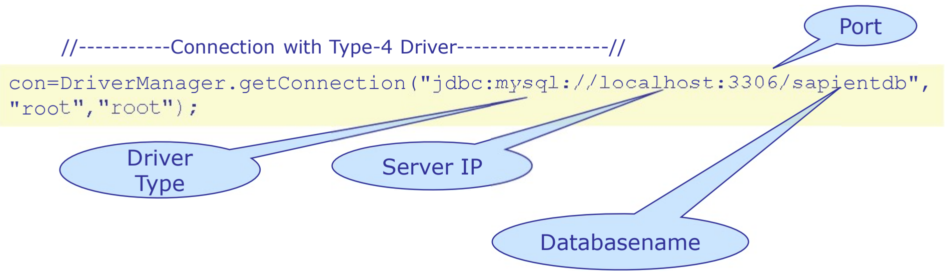
- Can be done in two ways:
  - Using *DriverManager*:
    using registerDriver() method of DriverManager
  - Using *Class.forName():*
    - A fully qualified name of the driver class should be passed as a parameter
    - The following call will load the MySql driver:

```
Class.forName("com.mysql.jdbc.Driver");
```

13

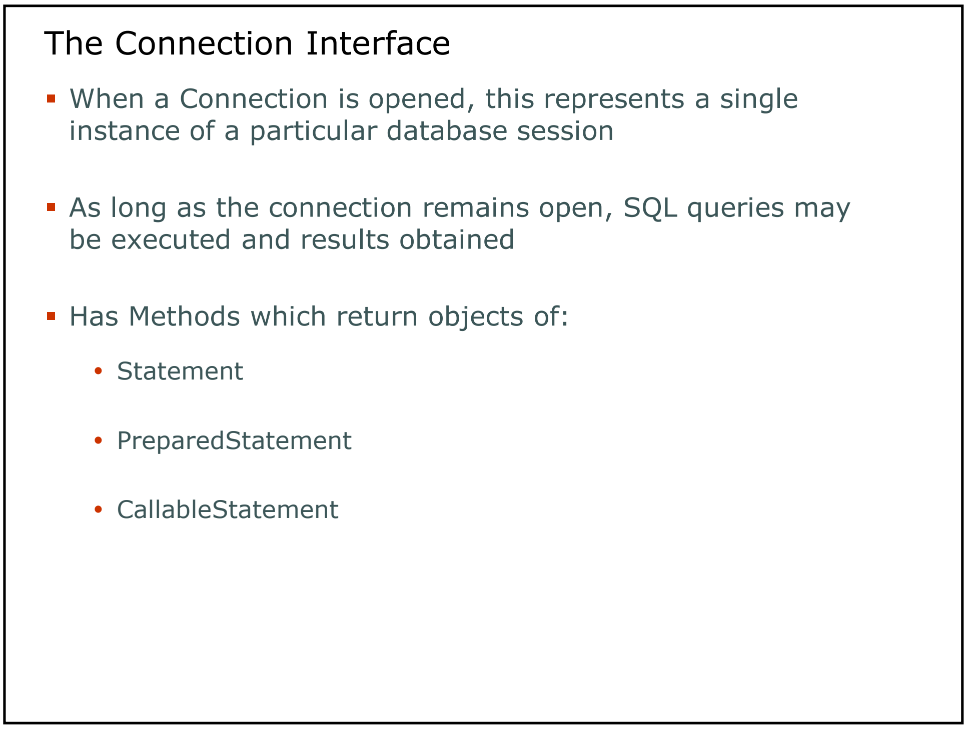
Connecting to Database

- The *getConnection()* method:
  - A static method of *DriverManager* class
  - Provides many overloaded versions
  - Returns a *Connection* object



```
//-----------Connection with Type-4 Driver------------------//
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/sapientdb",
"root","root");
```

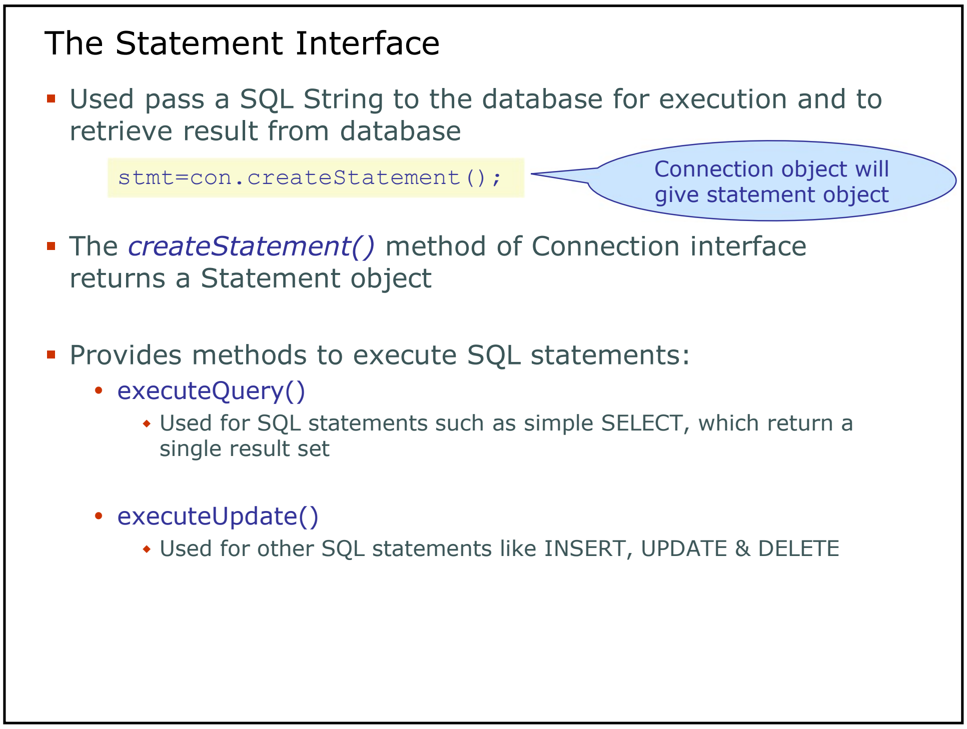
14



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

15



The Statement Interface

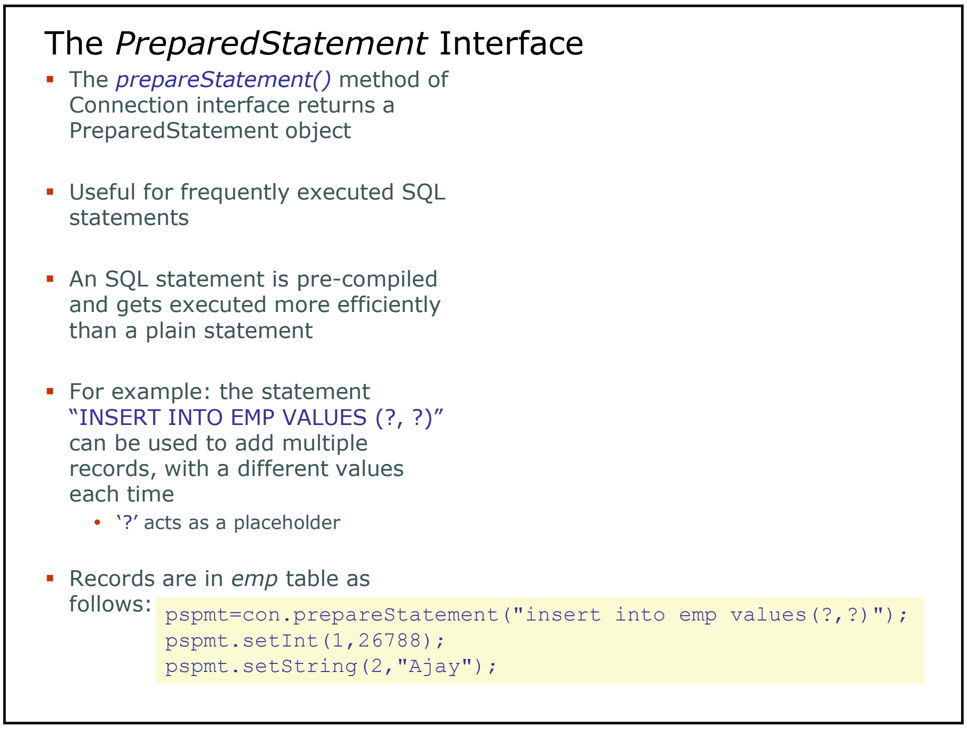
- Used pass a SQL String to the database for execution and to retrieve result from database

```
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

16

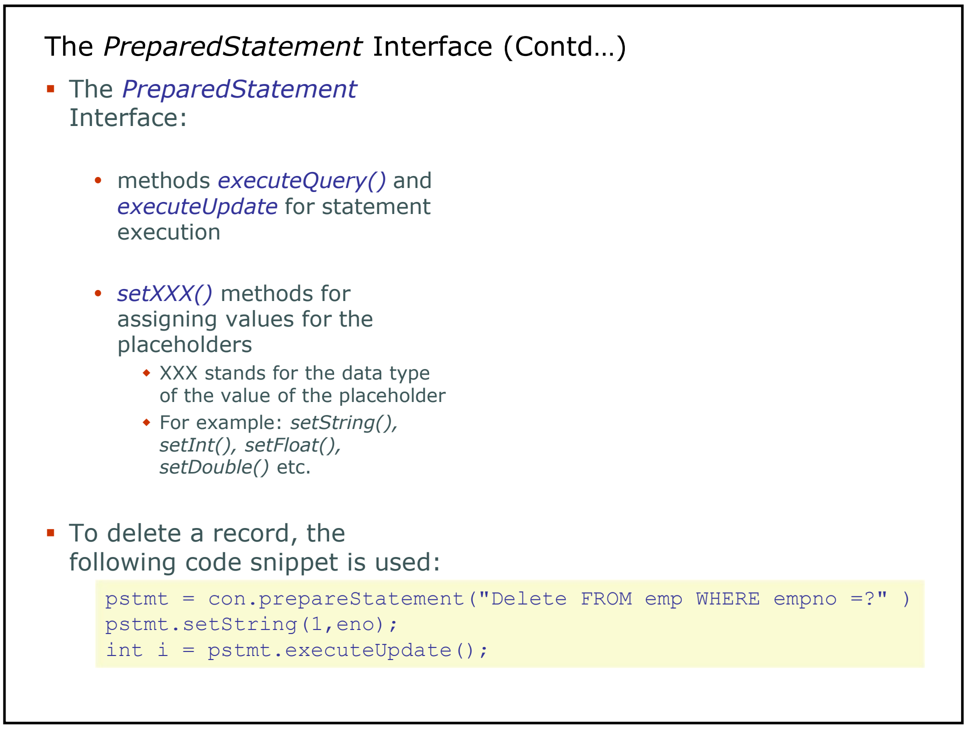


The *PreparedStatement* Interface

- 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");
```

17



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();
```

18

The *PreparedStatement* Interface (Contd…)

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

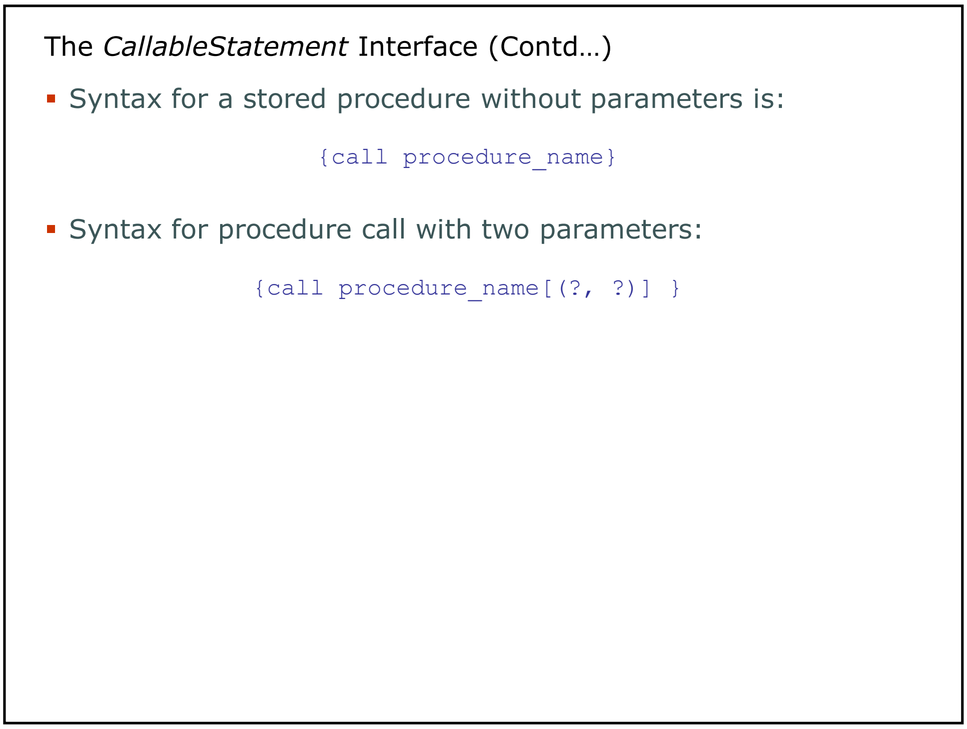
```
PreparedStatement stmt = con.prepareStatement("update
Emp SET Ename = ? WHERE Empno = ?");
st.setString(1,"Amit");
st.setString(2,“50000");
int i = stmt.executeUpdate();
```

19

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

20



The *CallableStatement* Interface (Contd…)

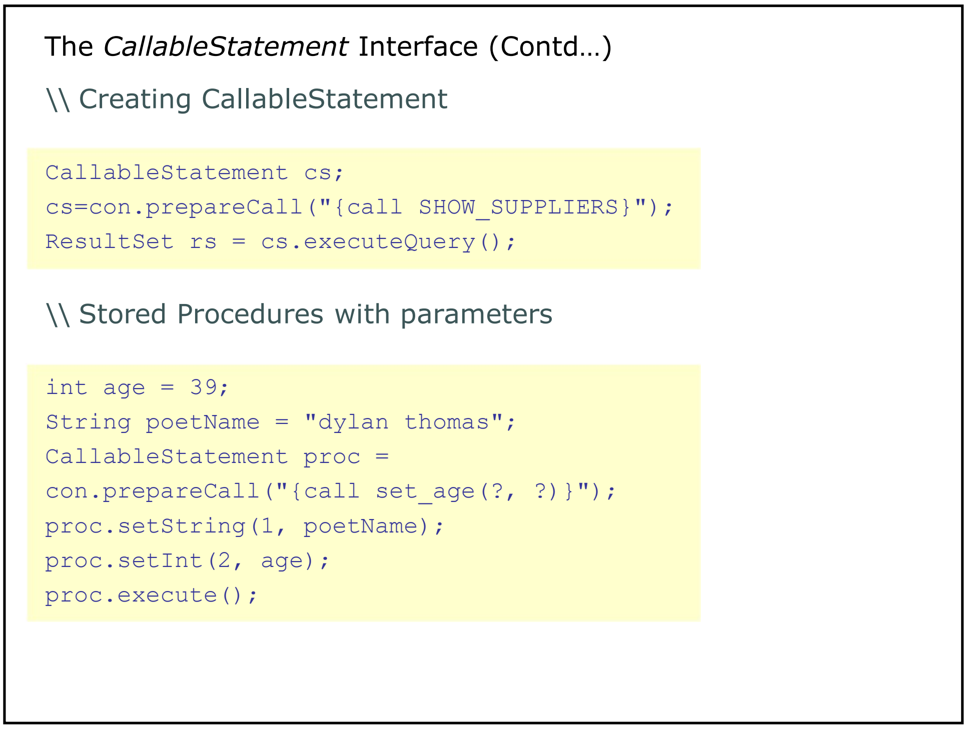
- Syntax for a stored procedure without parameters is:

```
{call procedure_name}
```

- Syntax for procedure call with two parameters:

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

21



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);
proc.execute();
```

22

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

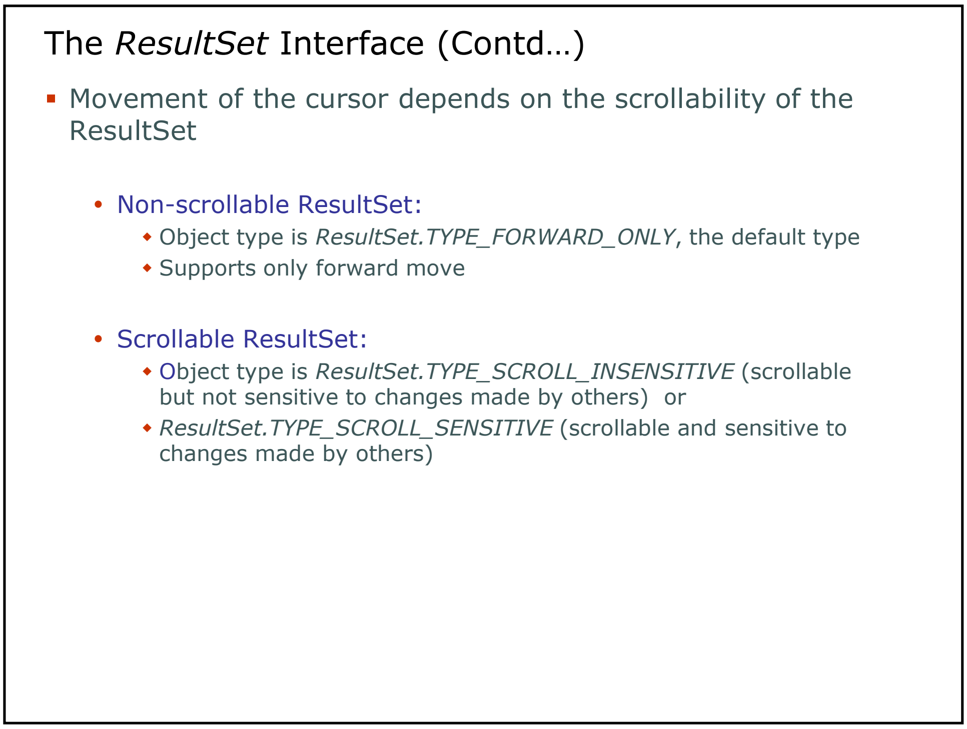
```
rs=stmt.executeQuery("select * from emp");
```

23

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

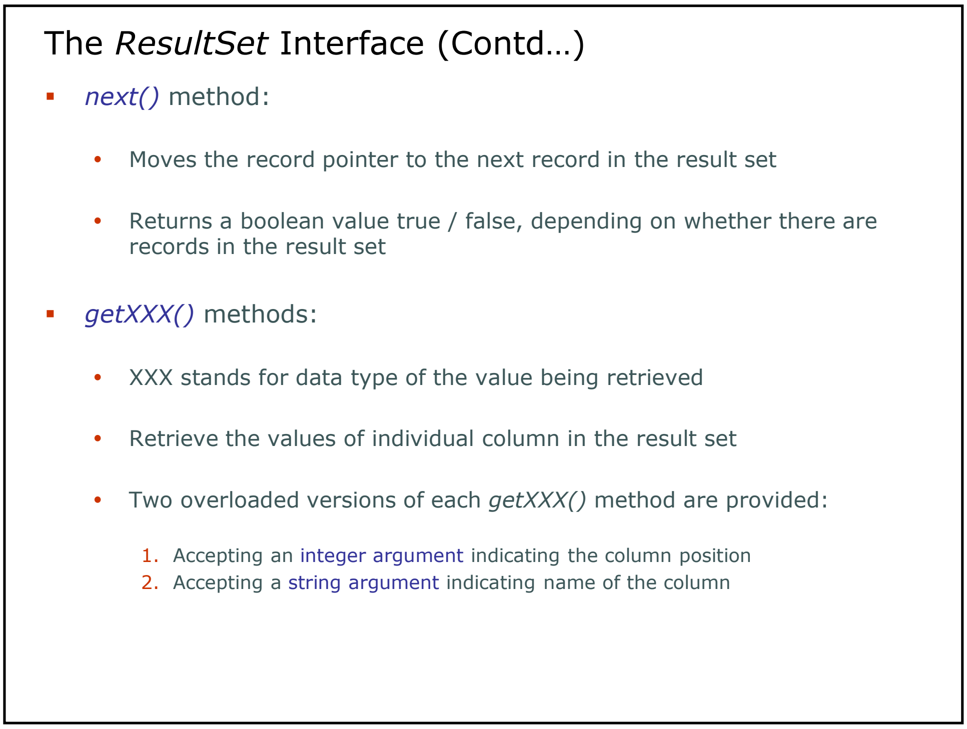
24



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)

25



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

26

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));

}

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

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

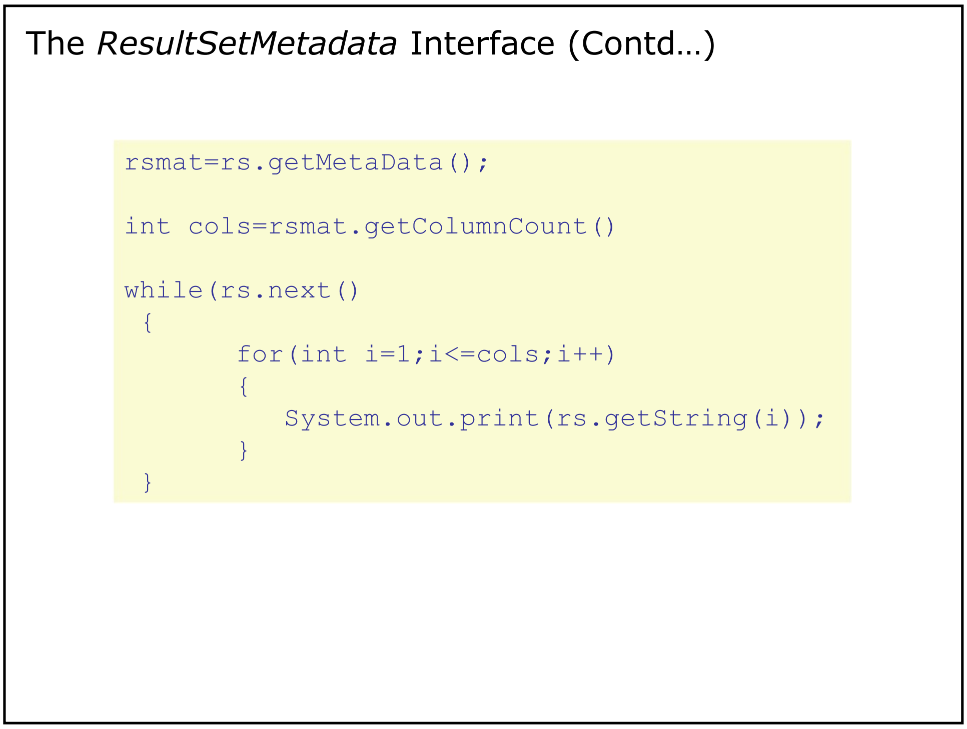
Don’t forget to close the Resultset, Statement & Connection

Maintain the sequence as shown

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



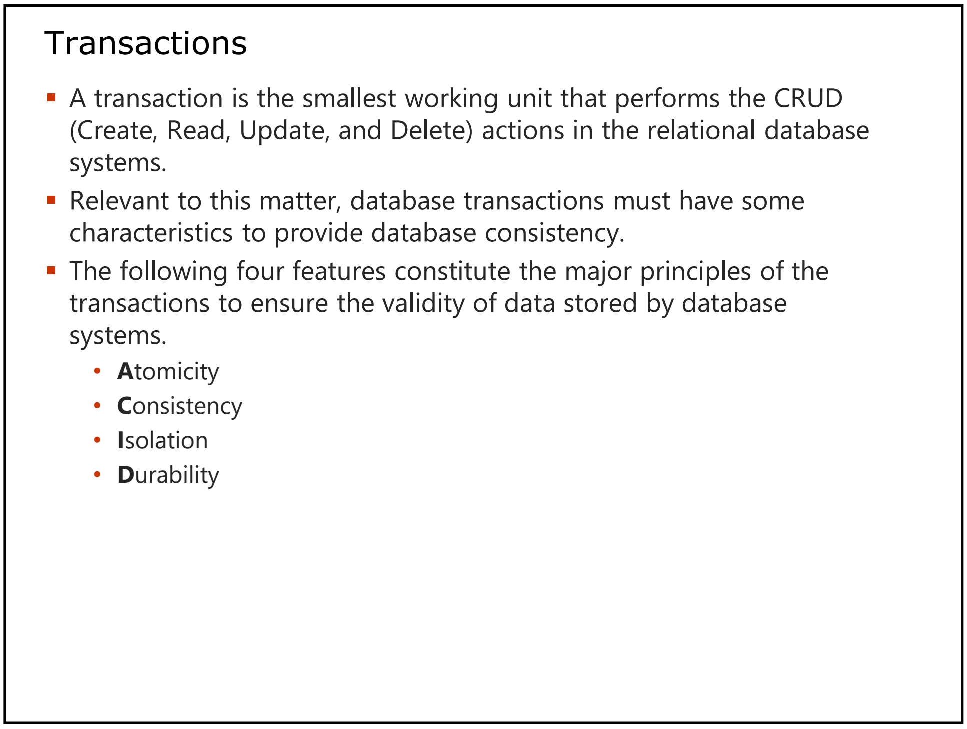
The *ResultSetMetadata* Interface (Contd…)

```
rsmat=rs.getMetaData();

int cols=rsmat.getColumnCount()

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

29



Transactions

- A transaction is the smallest working unit that performs the CRUD (Create, Read, Update, and Delete) actions in the relational database systems.
- Relevant to this matter, database transactions must have some characteristics to provide database consistency.
- The following four features constitute the major principles of the transactions to ensure the validity of data stored by database systems.
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

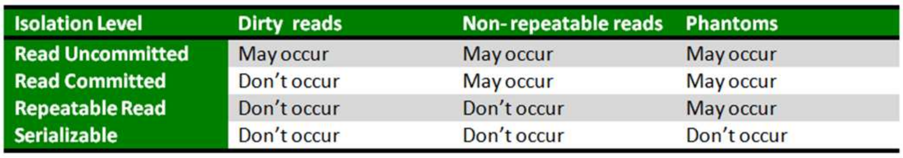
30

Transaction Phenomena

- Dirty Reads
- Non-Repeatable Reads
- Phantom Reads

Transaction Isolation Levels

- Read Uncommitted
- Read Committed
- Repeatable Read
- Serializable



Isolation Level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	May occur	May occur	May occur
Read Committed	Don't occur	May occur	May occur
Repeatable Read	Don't occur	Don't occur	May occur
Serializable	Don't occur	Don't occur	Don't occur

31

Thank You

32