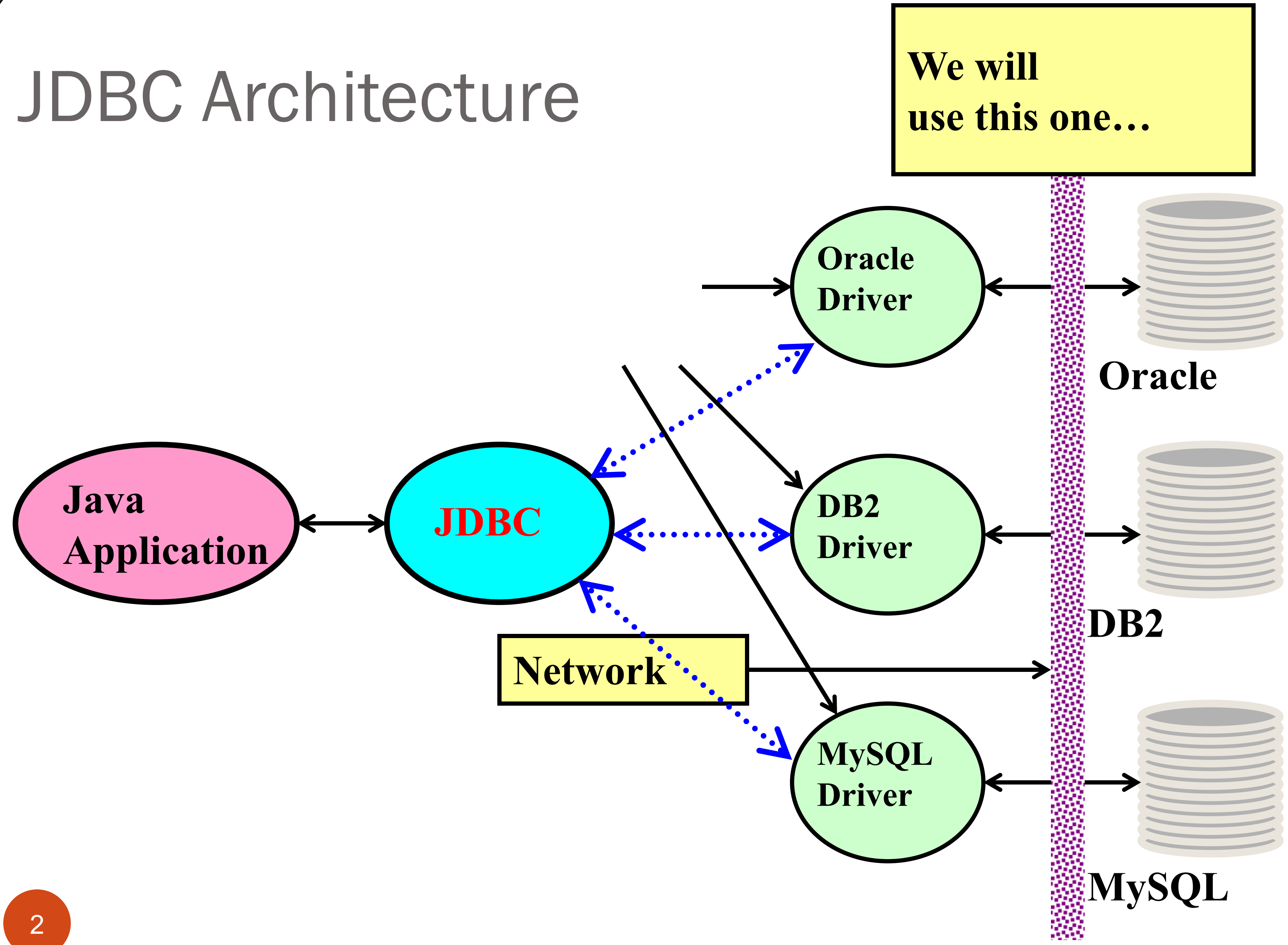
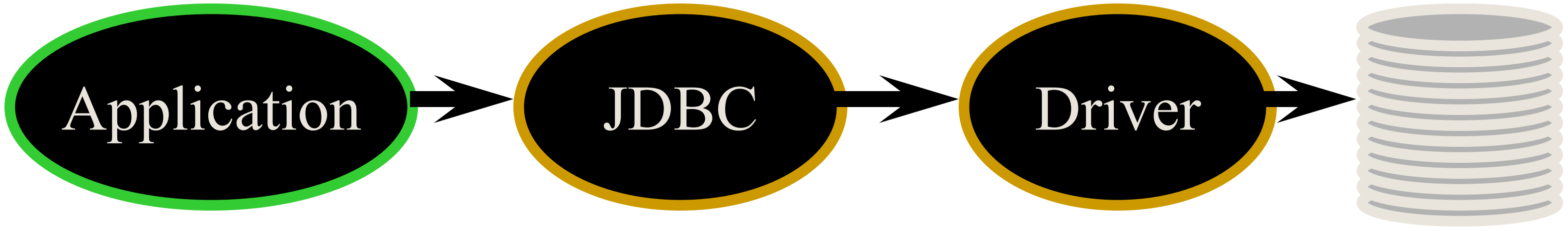
## Introduction to JDBC

- **JDBC** is used for accessing databases from Java applications
- Industrial strength DBMS – Eg: Oracle,Sybase,DB2 etc.,
- Challenge of SUN?
- 1996 – Sun developed JDBC Driver and API
- JDBC not a driver
- Only a specification (translator)

# JDBC Architecture



# JDBC Architecture

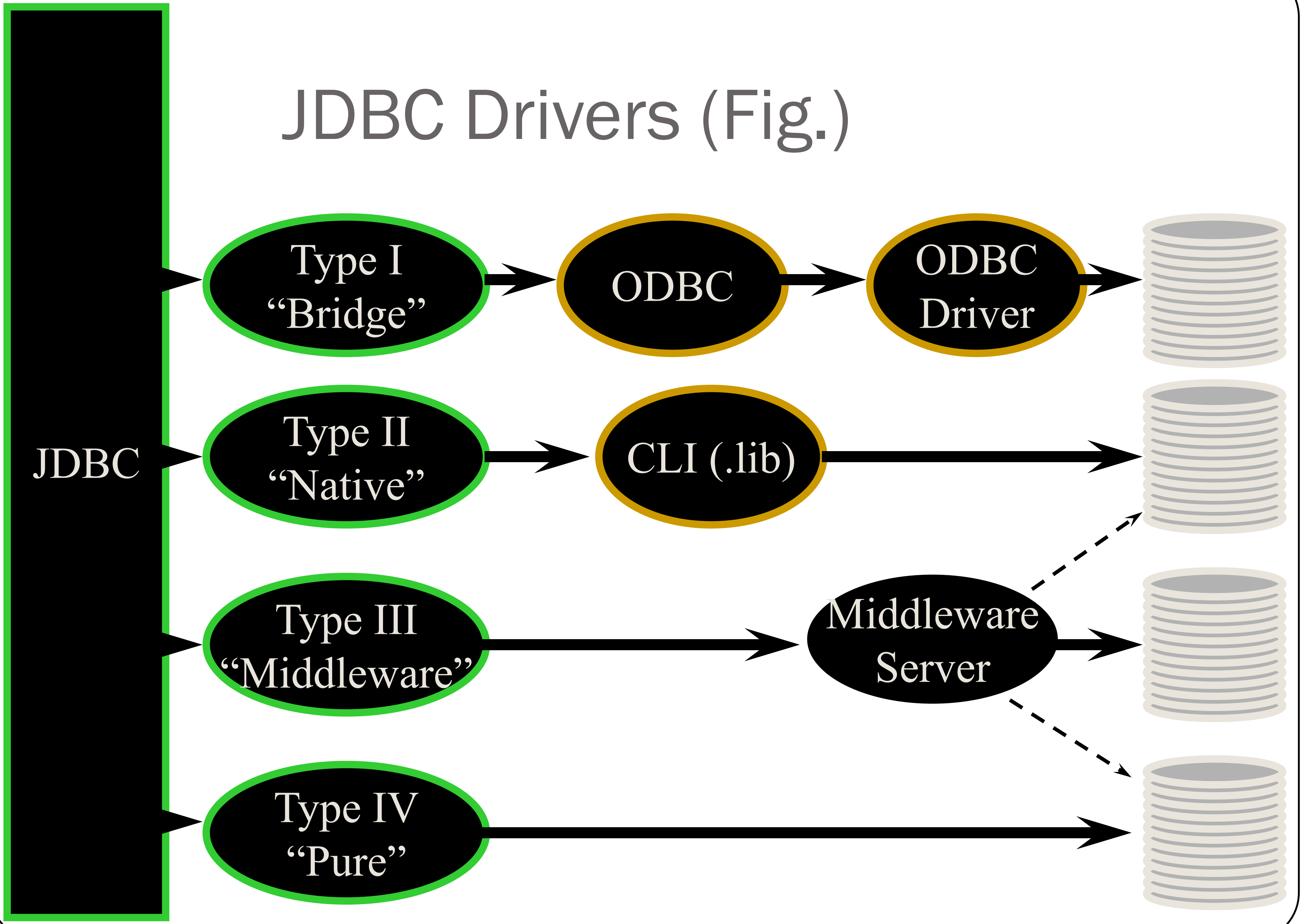


- Java code calls JDBC library
- JDBC loads a *driver*
- Driver talks to a particular database
- Can have more than one driver -> more than one database
- Ideal: can change database engines <u>without changing any application code</u>

## JDBC Drivers

- Type I: “Bridge”
- Type II: “Native”
- Type III: “Middleware”
- Type IV: “Pure”

# JDBC Drivers (Fig.)



## Packages

- Java.sql – Part of J2SE, basics for connecting to DBMS, contains core data objects of JDBC API
- Javax.sql extends java.sql – Part of J2EE, Interacts with JNDI, manages connections and other advanced features

## JDBC Process

- Loading JDBC driver
- Connecting to DBMS
- Creating and executing statement
- Process data returned by DBMS
- Terminate connection with DBMS

## DriverManager

- DriverManager tries all the drivers
- Uses the first one that works
- When a driver class is first loaded, it registers itself with the DriverManager
- Therefore, to register a driver, just load it!

## Loading/Registering a Driver

- statically load driver

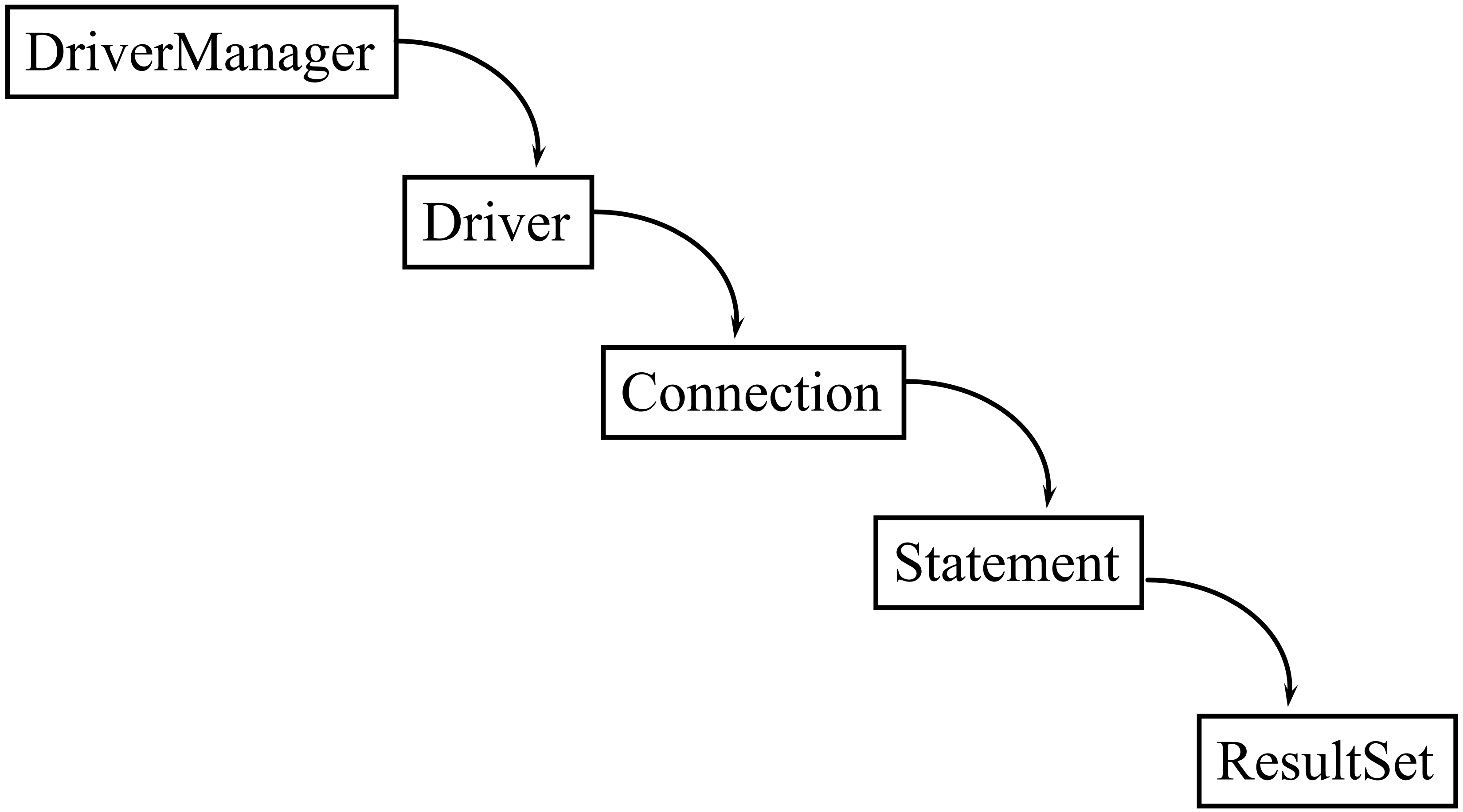
```
Class.forName("foo.bar.MyDriver");
Connection c =
  DriverManager.getConnection(...);
```

- or use the `jdbc.drivers` system property

## JDBC Object Classes

- DriverManager
  - Loads, chooses drivers
- Driver
  - connects to actual database
- Connection
  - a series of SQL statements to and from the DB
- Statement
  - a single SQL statement
- ResultSet
  - the records returned from a Statement

# JDBC Class Usage



## JDBC URLs

`jdbc:`*`subprotocol`*`:`*`source`*

- each driver has its own subprotocol
- each subprotocol has its own syntax for the source

`jdbc:odbc:`*`DataSource`*

  - e.g. `jdbc:odbc:Northwind`

`jdbc:msql://`*`host[:port]/database`*

  - e.g. `jdbc:msql://foo.nowhere.com:4333/accounting`

## DriverManager

```
Connection getConnection
  (String url, String user, String
  password)
```

- Connects to given JDBC URL with given user name and password
- Throws java.sql.SQLException
- returns a Connection object

## Connection

- A Connection represents a session with a specific database.
- Within the context of a Connection, SQL statements are executed and results are returned.
- Can have multiple connections to a database
  - NB: Some drivers don't support serialized connections
  - Fortunately, most do (now)
- Also provides "metadata" -- information about the database, tables, and fields
- Also methods to deal with transactions

## Obtaining a Connection

```
String url   = "jdbc:odbc:Northwind";
try {
  Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
  Connection con =
  DriverManager.getConnection(url,user id,password);
}
catch (ClassNotFoundException e)
  { e.printStackTrace(); }
catch (SQLException e)
  { e.printStackTrace(); }
```

## Connection Methods

**`Statement createStatement()`**

- returns a new Statement object

**`PreparedStatement prepareStatement(String sql)`**

- returns a new PreparedStatement object

**`CallableStatement prepareCall(String sql)`**

- returns a new CallableStatement object

- Why all these different kinds of statements? Optimization.

## Statement

- A Statement object is used for executing a static SQL statement and obtaining the results produced by it.

## Statement Methods

`ResultSet executeQuery(String)`

- Execute a SQL statement that returns a single ResultSet.

`int executeUpdate(String)`

- Execute a SQL INSERT, UPDATE or DELETE statement. Returns the number of rows changed.

`boolean execute(String)`

- Execute a SQL statement that may return multiple results.

- Why all these different kinds of queries? Optimization.

## ResultSet

- A ResultSet provides access to a table of data generated by executing a Statement.
- Only one ResultSet per Statement can be open at once.
- The table rows are retrieved in sequence.
- A ResultSet maintains a cursor pointing to its current row of data.
- The 'next' method moves the cursor to the next row.
  - you can’t rewind

## ResultSet Methods

- boolean next()
  - activates the next row
  - the first call to next() activates the first row
  - returns false if there are no more rows
- void close()
  - disposes of the ResultSet
  - allows you to re-use the Statement that created it
  - automatically called by most Statement methods

## ResultSet Methods

- *Type* get*Type*(int columnIndex)
  - returns the given field as the given type
  - fields indexed starting at 1 (not 0)
- *Type* get*Type*(String columnName)
  - same, but uses name of field
  - less efficient
- int findColumn(String columnName)
  - looks up column index given column name

## ResultSet Methods

- String getString(int columnIndex)
- boolean getBoolean(int columnIndex)
- byte getByte(int columnIndex)
- short getShort(int columnIndex)
- int getInt(int columnIndex)
- long getLong(int columnIndex)
- float getFloat(int columnIndex)
- double getDouble(int columnIndex)
- Date getDate(int columnIndex)
- Time getTime(int columnIndex)
- Timestamp getTimestamp(int columnIndex)

## SELECT Example

```
Connection con =
  DriverManager.getConnection(url, "alex",
  "8675309");
Statement st = con.createStatement();
ResultSet results =
  st.executeQuery("SELECT EmployeeID,
  LastName, FirstName FROM Employees");
```

## SELECT Example (Cont.)

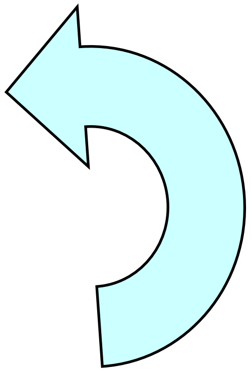
```
while (results.next()) {
  int id = results.getInt(1);
  String last = results.getString(2);
  String first = results.getString(3);
  System.out.println("" + id + ": " +
  first + " " + last);
}
st.close();
con.close();
```

## Modifying the Database

- use executeUpdate if the SQL contains “INSERT” or “UPDATE”
- executeUpdate returns the number of rows modified
- executeUpdate also used for “CREATE TABLE” etc. (DDL)

## Seven Steps

- Load the driver
- Define the connection URL
- Establish the connection
- Create a Statement object
- Execute a query using the Statement
- Process the result
- Close the connection



## Loading the Driver

- We can register the driver indirectly using the statement

  Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver");
- Class.forName loads the specified class
- When JdbcOdbcDriver is loaded, it automatically
  - creates an instance of itself
  - registers this instance with the DriverManager
- Hence, the driver class can be given as an argument of the application

## Connecting to the Database

- Every database is identified by a URL
- Given a URL, DriverManager looks for the driver that can talk to the corresponding database
- DriverManager tries all registered drivers, until a suitable one is found

## Connecting to the Database

```
String url=“jdbc:odbc:CustInfo”;
String userid=“amrita”;
String pwd=“april”
Connection con = DriverManager.
        getConnection(“url,userid,pwd");
```

**We Use:**
**DriverManager.getConnection(<URL>, <user>, <pwd>);**

## Interaction with the Database

- We use Statement objects in order to
  - Query the database
  - Update the database
- Three different interfaces are used:

  Statement, PreparedStatement, CallableStatement
- All are interfaces, hence cannot be instantiated
- They are created by the Connection

## Querying with Statement

```
String queryStr =
        "SELECT * FROM employee " +
        "WHERE lname = ‘Wong'";

Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(queryStr);
```

- The executeQuery method returns a ResultSet object representing the query result.
    - Will be discussed later…

## Process Data returned by DBMS

```
ResultSet results;
String firstname;
String lastname;
String printrow;
Boolean records=results.next();
if(!records)
{S.O.P("No data returned");
return;}
else{
do{
firstname=results.getString(firstname);
lastname=results.getString(lastname);
printrow=firstname+""+lastname
S.O.P(printrow);} while (results.next());}
```

## Changing DB with Statement

```
String deleteStr =
        "DELETE FROM employee " +
        "WHERE lname = 'Wong'";

Statement stmt = con.createStatement();
int delnum = stmt.executeUpdate(deleteStr);
```

- executeUpdate is used for data manipulation: insert, delete, update, create table, etc. (anything other than querying!)
- executeUpdate returns the number of rows modified

## About Prepared Statements

- Prepared Statements are used for queries that are executed many times
- They are parsed (compiled) by the DBMS only once
- Column values can be set after compilation
- Instead of values, use '?'
- Hence, Prepared Statements can be though of as statements that contain placeholders to be substituted later with actual values
- How precompilation works?

## Querying with PreparedStatement

```
String queryStr =
        "SELECT * FROM employee " +
        "WHERE superssn= ? and salary > ?";

PreparedStatement pstmt =
        con.prepareStatement(queryStr);

pstmt.setString(1, "333445555");
pstmt.setInt(2, 26000);

ResultSet rs = pstmt.executeQuery();
```

## Updating with PreparedStatement

```
String deleteStr =
        “DELETE FROM employee " +
        "WHERE superssn = ? and salary > ?";

PreparedStatement pstmt =
        con.prepareStatement(deleteStr);

pstmt.setString(1, "333445555");
pstmt.setDouble(2, 26000);

int delnum = pstmt.executeUpdate();
```

## Statements vs. PreparedStatements: Be Careful!

- Are these the same? What do they do?

```
String val = "abc";
PreparedStatement pstmt =
        con.prepareStatement("select * from R where A=?");
pstmt.setString(1, val);
ResultSet rs =  pstmt.executeQuery();
```

```
String val = "abc";
Statement stmt =  con.createStatement( );
ResultSet rs =
        stmt.executeQuery("select * from R where A=" + val);
```

## Statements vs. PreparedStatements: Be Careful!

- Will this work?

```
PreparedStatement pstmt =
          con.prepareStatement("select * from ?");


pstmt.setString(1, myFavoriteTableString);
```

## Timeout

- Use setQueryTimeOut(int seconds) of Statement to set a timeout for the driver to wait for a statement to be completed
- If the operation is not completed in the given time, an SQLException is thrown
- What is it good for?

## ResultSet

- ResultSet objects provide access to the tables generated as results of executing a Statement queries
- Only one ResultSet per Statement can be open at the same time!
- The table rows are retrieved in sequence
  - A ResultSet maintains a cursor pointing to its current row
  - The next() method moves the cursor to the next row

## ResultSet Methods

- boolean next()
  - activates the next row
  - the first call to next() activates the first row
  - returns false if there are no more rows
- void close()
  - disposes of the ResultSet
  - allows you to re-use the Statement that created it
  - automatically called by most Statement methods

## ResultSet Methods

- *Type* get*Type*(int columnIndex)
  - returns the given field as the given type
  - indices start at 1 and not 0!
- *Type* get*Type*(String columnName)
  - same, but uses name of field
  - less efficient
- For example: getString(columnIndex), getInt(columnName), getTime, getBoolean, getType,…
- int findColumn(String columnName)
  - looks up column index given column name

## ResultSet Methods

- JDBC 2.0 includes scrollable result sets. Additional methods included are : ‘first’, ‘last’, ‘previous’, and other methods.

## ResultSet Example

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.
executeQuery("select lname,salary from employees");
// Print the result
while(rs.next()) {
 System.out.print(rs.getString(1) + ":");
 System.out.println(rs.getDouble("salary"));
}
```

## Mapping Java Types to SQL Types

| SQL type | Java Type |
|---|---|
| CHAR, VARCHAR, LONGVARCHAR | String |
| NUMERIC, DECIMAL | java.math.BigDecimal |
| BIT | boolean |
| TINYINT | byte |
| SMALLINT | short |
| INTEGER | int |
| BIGINT | long |
| REAL | float |
| FLOAT, DOUBLE | double |
| BINARY, VARBINARY, LONGVARBINARY | byte[] |
| DATE | java.sql.Date |
| TIME | java.sql.Time |
| TIMESTAMP | java.sql.Timestamp |

## Null Values

- In SQL, NULL means the field is empty
- Not the same as 0 or ""
- In JDBC, you must explicitly ask if the last-read field was null
  - ResultSet.wasNull(column)
- For example, getInt(column) will return 0 if the value is either 0 or NULL!

## Null Values

- When inserting null values into placeholders of Prepared Statements:
  - Use the method setNull(*index*, Types.*sqlType*) for primitive types (e.g. INTEGER, REAL);
  - You may also use the set*Type*(*index*, null) for object types (e.g. STRING, DATE).

## ResultSet Meta-Data

A ResultSetMetaData is an object that can be used to get information about the properties of the columns in a ResultSet object

An example: write the columns of the result set

```
ResultSetMetaData rsmd = rs.getMetaData();
int numcols = rsmd.getColumnCount();


for (int i = 1 ; i <= numcols; i++) {
          System.out.print(rsmd.getColumnLabel(i)+" ");
}
```

## Database Time

- Times in SQL are notoriously non-standard
- Java defines three classes to help
- java.sql.Date
  - year, month, day
- java.sql.Time
  - hours, minutes, seconds
- java.sql.Timestamp
  - year, month, day, hours, minutes, seconds, nanoseconds
  - usually use this one

## Cleaning Up After Yourself

- Remember to close the Connections, Statements, Prepared Statements and Result Sets

```
con.close();
stmt.close();
pstmt.close();
rs.close()
```

## Dealing With Exceptions

- An SQLException is actually a list of exceptions

```
catch (SQLException e) {
   while (e != null) {
         System.out.println(e.getSQLState());
         System.out.println(e.getMessage());
         System.out.println(e.getErrorCode());
         e = e.getNextException();
   }
}
```

# Transactions and JDBC

- Transaction: more than one statement that must all succeed (or all fail) together
  - e.g., updating several tables due to customer purchase
- If one fails, the system must reverse all previous actions
- Also can't leave DB in inconsistent state halfway through a transaction
- COMMIT = complete transaction
- ROLLBACK = cancel all actions

## Example

- Suppose we want to transfer money from bank account 13 to account 72:

```
PreparedStatement pstmt =
        con.prepareStatement("update BankAccount
                              set amount = amount + ?
                              where accountId = ?");
pstmt.setInt(1,-100);
pstmt.setInt(2, 13);
pstmt.executeUpdate();
pstmt.setInt(1, 100);
pstmt.setInt(2, 72);
pstmt.executeUpdate();
```

**What happens if this update fails?**

## Transaction Management

- Transactions are not explicitly opened and closed
- The connection has a state called AutoCommit mode
- if AutoCommit is true, then every statement is automatically committed
- if AutoCommit is false, then every statement is added to an ongoing transaction
- Default: true

## AutoCommit

```
setAutoCommit(boolean val)
```

- If you set AutoCommit to false, you must explicitly commit or rollback the transaction using `Connection.commit()` and `Connection.rollback()`
- Note: DDL statements (e.g., creating/deleting tables) in a transaction may be ignored or may cause a commit to occur
  - The behavior is DBMS dependent

## Scrollable ResultSet

- Statement createStatement( int resultSetType, int resultSetConcurrency)
- **resultSetType**:
- ResultSet.TYPE_FORWARD_ONLY
- -default; same as in JDBC 1.0
- -allows only forward movement of the cursor
- -when rset.next() returns false, the data is no longer available and the result set is closed.
- ResultSet.TYPE_SCROLL_INSENSITIVE
- -backwards, forwards, random cursor movement.
- -changes made in the database are not seen in the result set object in Java memory.
- ResultSetTYPE_SCROLL_SENSITIVE
- -backwards, forwards, random cursor movement.
- -changes made in the database are seen in the
- result set object in Java memory.

# Scrollable ResultSet (cont'd)

- **resultSetConcurrency:**
- ResultSet.CONCUR_READ_ONLY
- This is the default (and same as in JDBC 1.0) and allows only data to be read from the database.
- ResultSet.CONCUR_UPDATABLE
- This option allows for the Java program to make changes to the database based on new methods and positioning ability of the cursor.
- **Example:**
- `Statement stmt = conn.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);`
- `ResultSetrset= stmt.executeQuery( "SHOW TABLES");`

# Scrollable ResultSet (cont'd)

**`public boolean absolute(int row) throws SQLException`**

- -If the given row number is positive, this method moves the cursor to the given row number (with the first row numbered 1).
- -If the row number is negative, the cursor moves to a relative position from the last row.
- -If the row number is 0, an SQLException will be raised.

`public boolean relative(int row) throws SQLException`

- This method call moves the cursor a relative number of rows, either positive or negative.
- An attempt to move beyond the last row (or before the first row) in the result set positions the cursor after the last row (or before the first row).

```
public boolean first() throws SQLException
public boolean last() throws SQLException
public boolean previous() throws SQLException
public boolean next() throws SQLException
```

## Scrollable ResultSet (cont'd)

```
public void beforeFirst() throws SQLException
public void afterLast() throws SQLException
public boolean isFirst() throws SQLException
public boolean isLast() throws SQLException
public boolean isAfterLast() throws
  SQLException
public boolean isBeforeFirst() throws
  SQLException
public int getRow() throws SQLException
```

- getRow() method retrieves the current row number: The first row is number 1, the second number 2, and so on.