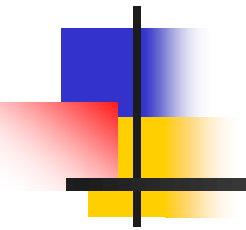
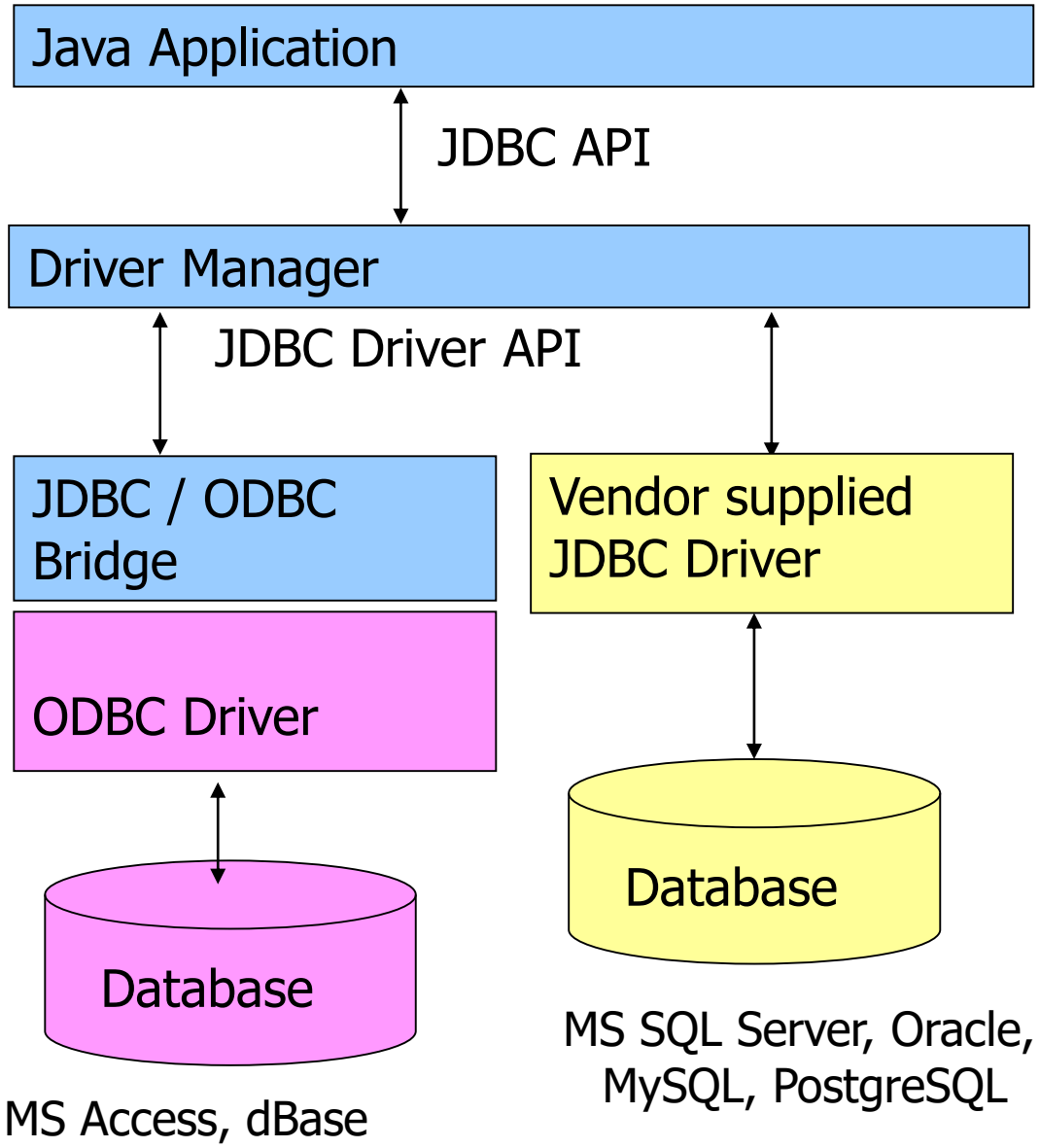


Java Database Connectivity (JDBC)

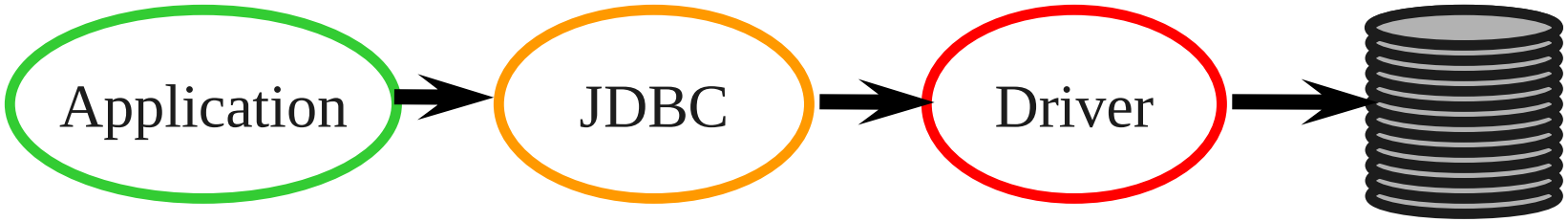
Java DataBase Connectivity

- **JDBC** (Java DataBase Connectivity) - provides access to **relational database** systems
- **JDBC** is a **vendor independent API** for accessing relational data from different vendors (Microsoft Access, Oracle) in a consistent way
- The language **SQL** (Structured Query Language) is normally used to make queries on relational data
- **JDBC** API **provides methods for executing SQL** statements and obtaining results: *SELECT, UPDATE, INSERT, DELETE* etc.
- Provides portability (eliminates rewriting code for different databases and recompiling for different platforms) and faster, reusable object developing environment
- **JDBC** API is part of core Java

JDBC-to-database communication



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
- **Advantage**: can change database engines without changing any application code

Vendor specific APIs - JDBC Drivers

- Database vendors provide proprietary APIs for accessing data managed by the server.
- JDBC aims at providing an API that eliminates vendor specific nature in accessing a database

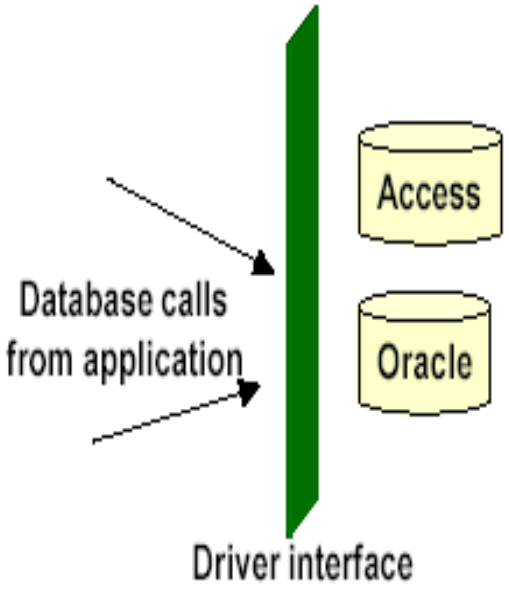
- However, JDBC still requires a vendor-specific driver for accessing database from a particular vendor
- The driver provides interface between JDBC API and vendor database by converting calls from JDBC API to vendor's database calls
- Example drivers:

- JDBC/ODBC driver:

  sun.jdbc.odbc.JdbcOdbcDriver

- Oracle driver:

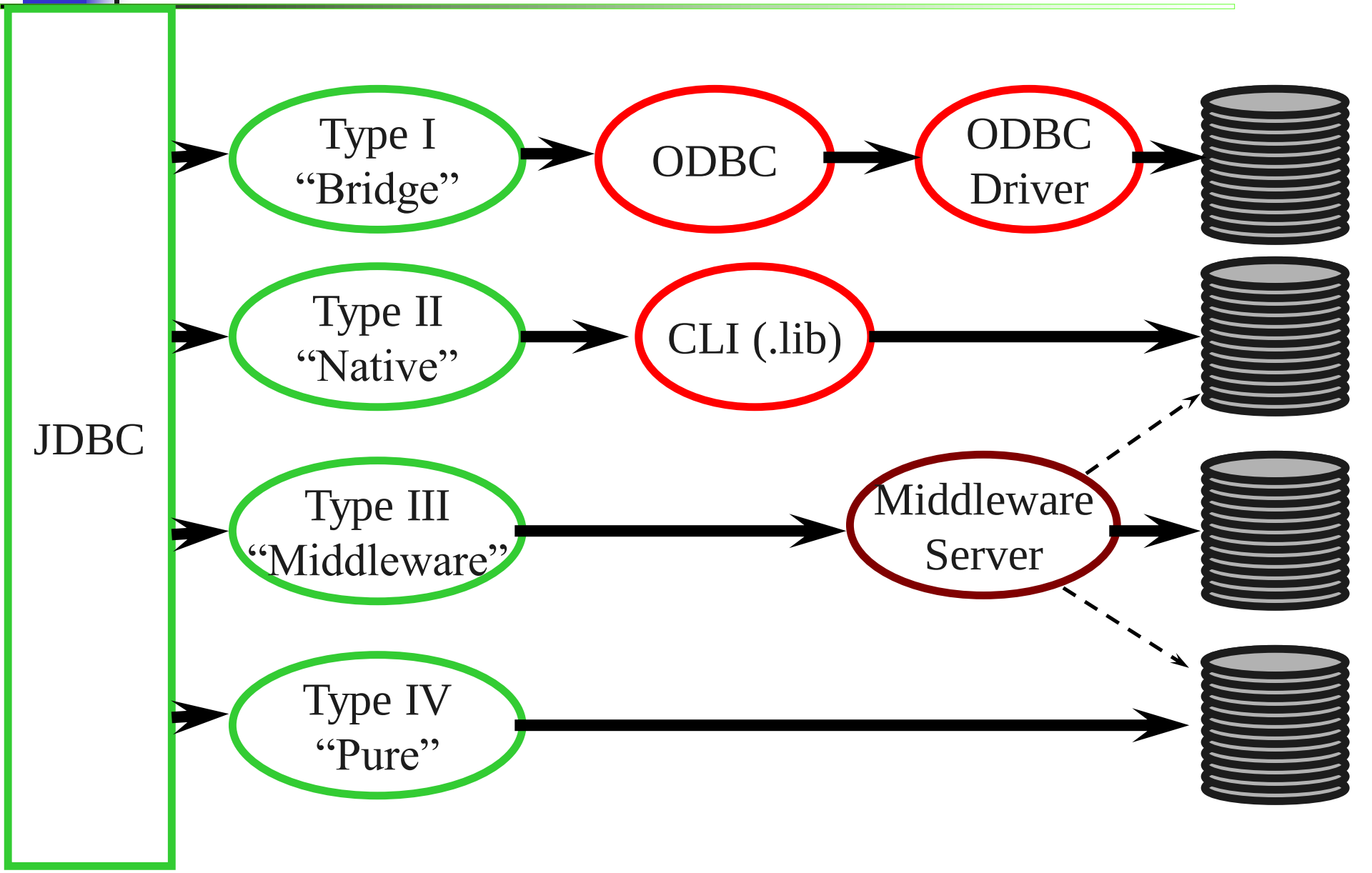
  oracle.jdbc.driver.OracleDriver



JDBC Drivers

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

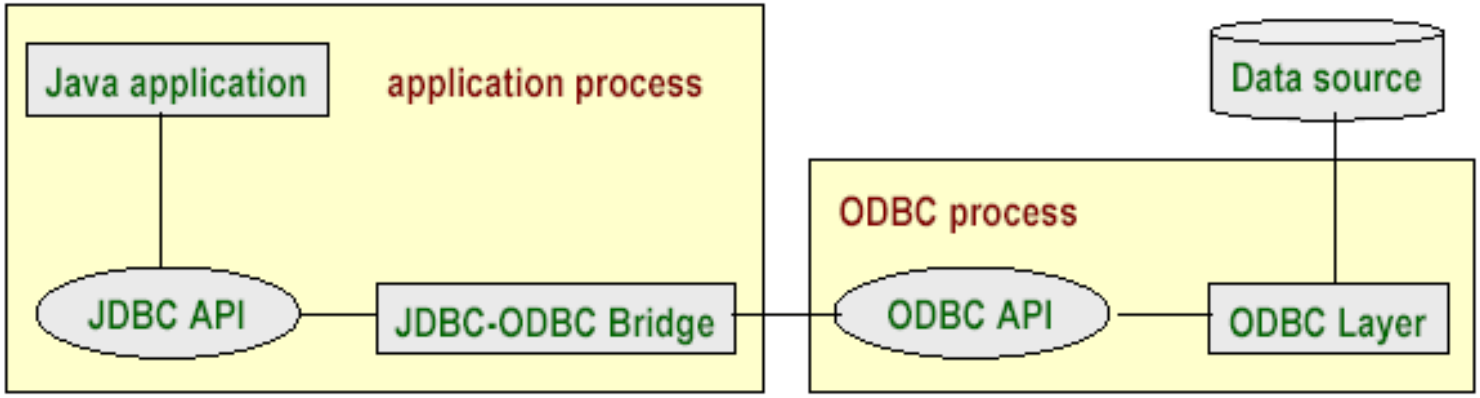
JDBC Drivers



JDBC Driver Types

Type 1: JDBC-ODBC Bridge

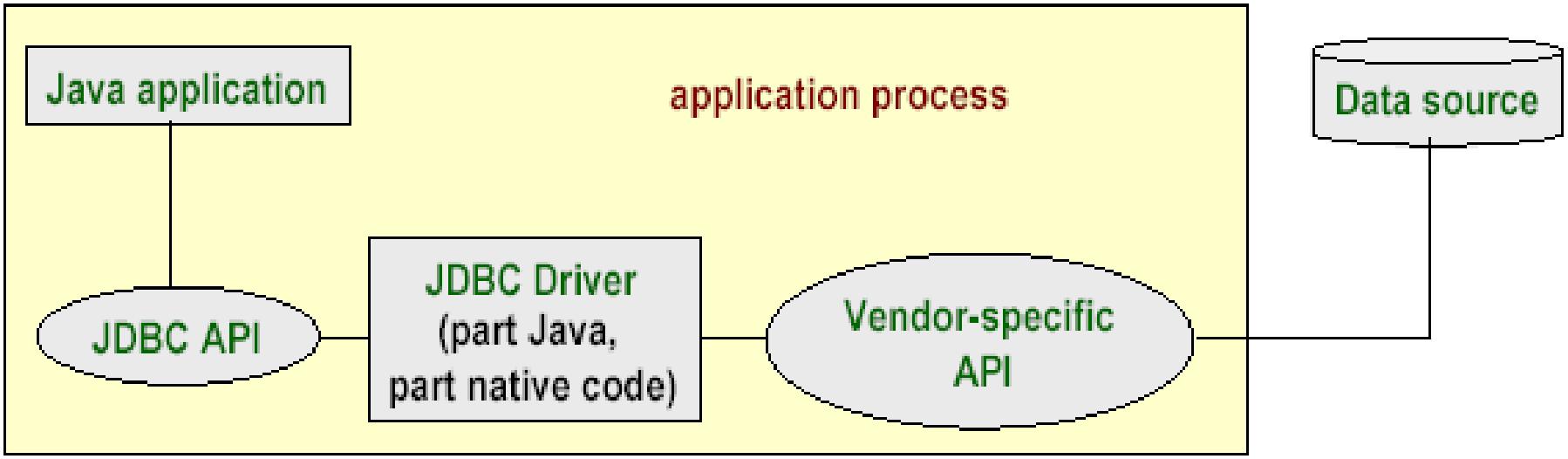
- ODBC (Open Database Connectivity) is Microsoft's API for SQL; popular on Windows platform
- ODBC API provides a set of functions for accessing a database
- JDBC drivers of this type translate calls from JDBC into corresponding ODBC calls



- The database access may be inefficient due to many layers of calls involved
- Limited to what ODBC API provides even if database vendor's API has superior functionality
- This is the only way Microsoft Access database can be accessed from Java

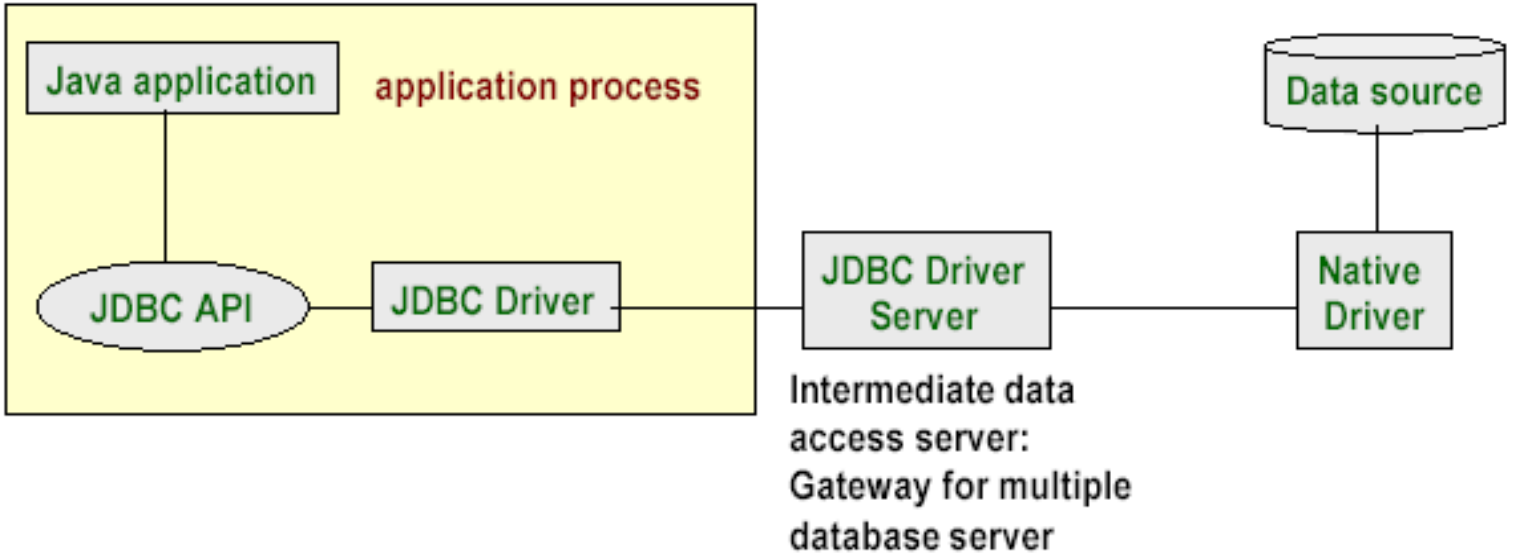
Type 2 - Part Java, Part Native Driver

- JDBC driver consists of java code and native code which uses vendor-specific API for accessing databases
- More efficient than JDBC-ODBC bridge due to fewer layers of communication
- Typical of this type of driver is the driver provided by IBM for its DB2 Universal Database (UDB).



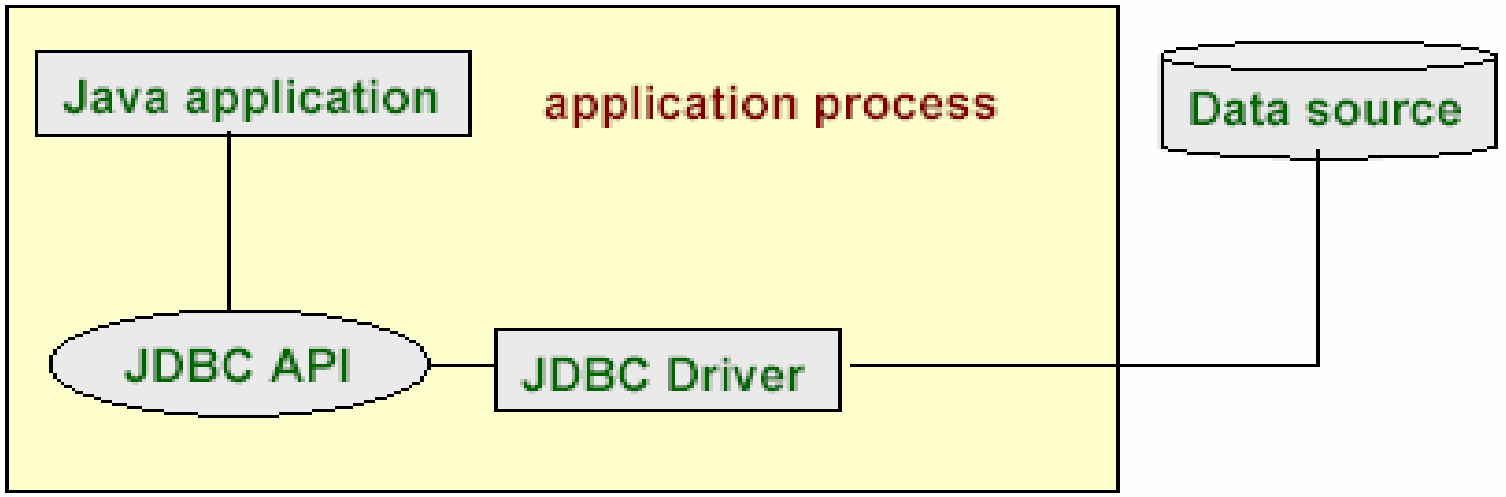
Type 3 - Intermediate Database Access Server

- Calls **middleware** server, usually on database host
- Very flexible -- allows access to multiple databases using one driver
- Only need to download one driver
- But it's another server application to install and maintain



Type 4 - Pure Java Driver

- 100% **Pure Java**
- Use Java networking libraries to talk directly to database engines
- **Only disadvantage**: need to download a new driver for each database engine
- JDBC calls are directly translated to database calls specific to vendor
- Very efficient in terms of performance and development time



JDBC Overview

Core JDBC Components

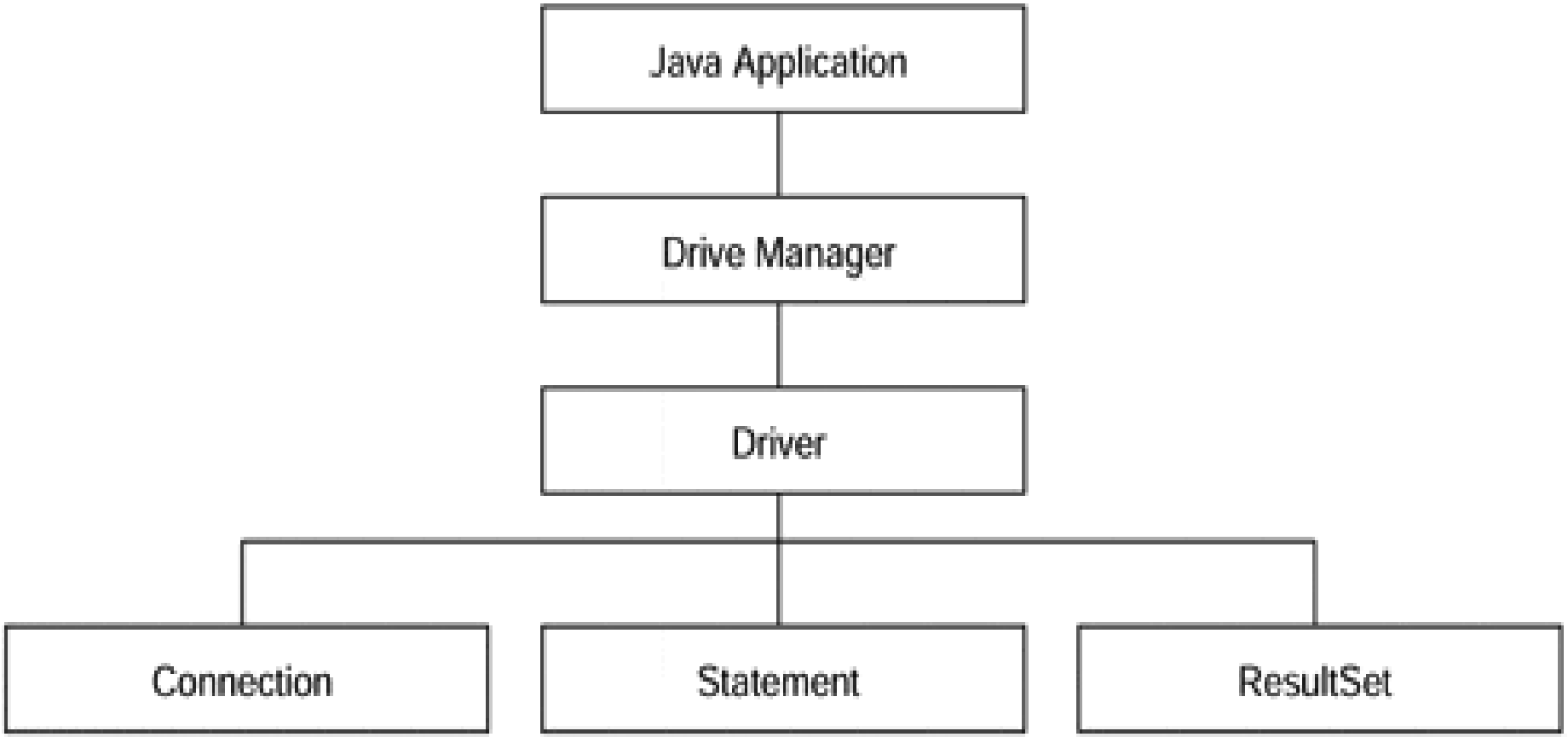
- JDBC Drivers
- Connections
- Statements
- Result Sets

Common JDBC Use Cases

- Query the database (read data from it).
- Query the database meta data.
- Update the database.
- Perform transactions.

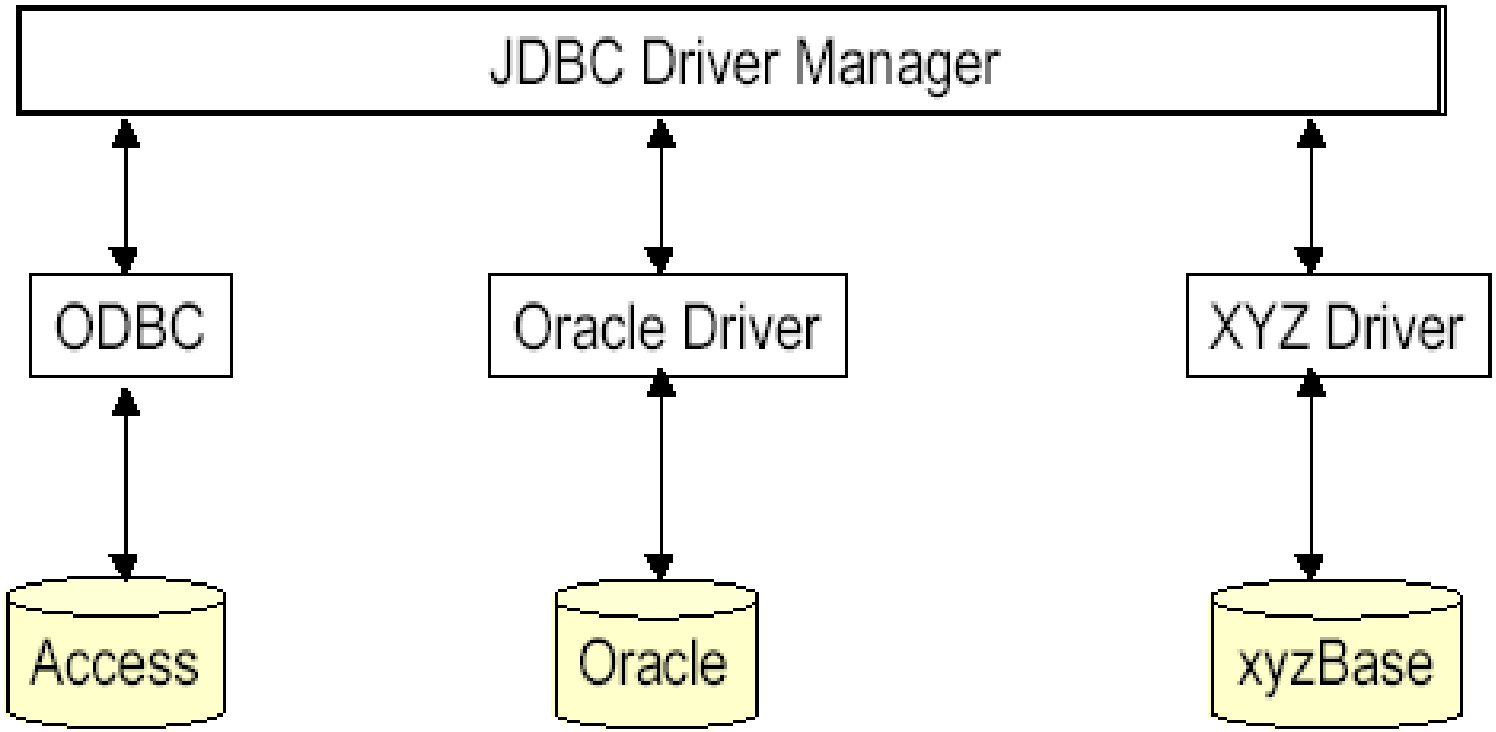
JDBC Key Components

- JDBC key components: DriverManager, Connection, Statement, ResultSet



The DriverManager Object

- **DriverManager** handles communication with different drivers that conform to JDBC Driver API
- The static class ***DriverManager*** manages the loaded drivers and contains methods for accessing connections to the databases



The DriverManager Object.

- Once a driver is installed, you need to load it into your Java object by using the **`DriverManager`**. It provides a common interface to a JDBC driver object without having to delve into the internals of the database itself
- The driver is responsible for creating and implementing the **`Connection`**, **`Statement`**, and **`ResultSet`** objects for the specific database.
- **`DriverManager`** then is able to acquire those object implementations for itself. In so doing, applications that are written using the **`DriverManager`** are isolated from the implementation details of databases.

Database Connection Interface

- The `Connection` object is responsible for establishing the link between the Database Management System and the Java application.
- The `Connection.getConnection` method accepts a URL and enables the JDBC object to use different drivers depending on the situation, isolates applets from connection-related information, and gives the application a means by which to specify the specific database to which it should connect. The URL takes the form of

```
jdbc:<subprotocol>:<subname>.
```

The subprotocol is a kind of connectivity to the database

Database Statement Object.

- A **`Statement`** encapsulates a **`query`** written in `Structured Query Language` and enables the **`JDBC`** object to compose a series of steps to look up information in a database.
- Using a **`Connection`**, the **`Statement`** can be forwarded to the database and obtain a **`ResultSet`**

ResultSet Access Control.

- A `ResultSet` is a container for a series of rows and columns acquired from a `Statement` call. Using the `ResultSet's` iterator routines, the JDBC object can step through each row in the result set. Individual column fields can be retrieved using the get methods within the `ResultSet.` Columns may be specified by their field name or by their index.

JDBC Operations

- **Query the database**
  - One of the most common use cases is to read data from a database. Reading data from a database is called querying the database.

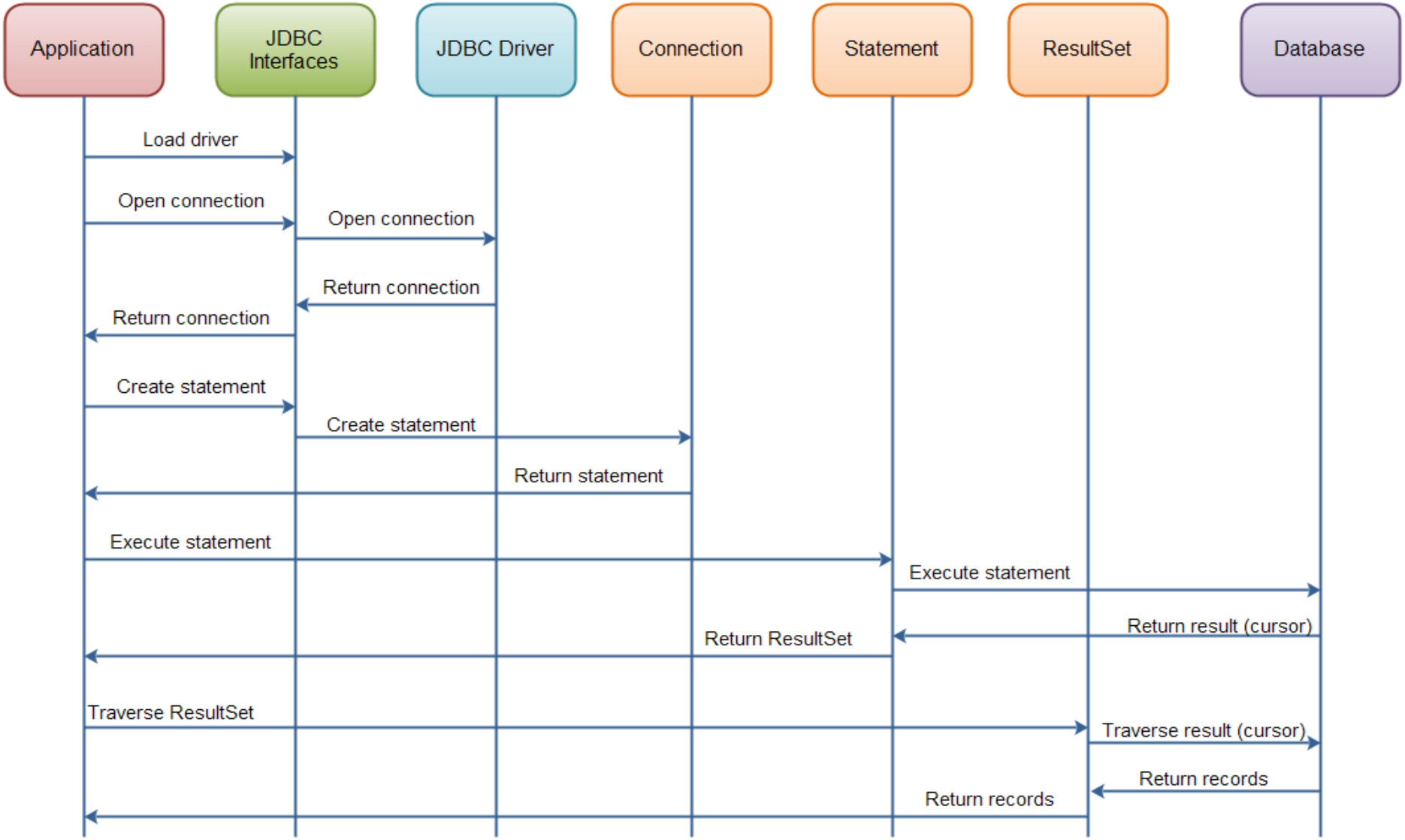
- **Query the database meta data**
  - Another common use case is to query the database meta data. The database meta data contains information about the database itself. For instance, information about the tables defined, the columns in each table, the data types etc.

JDBC Operation

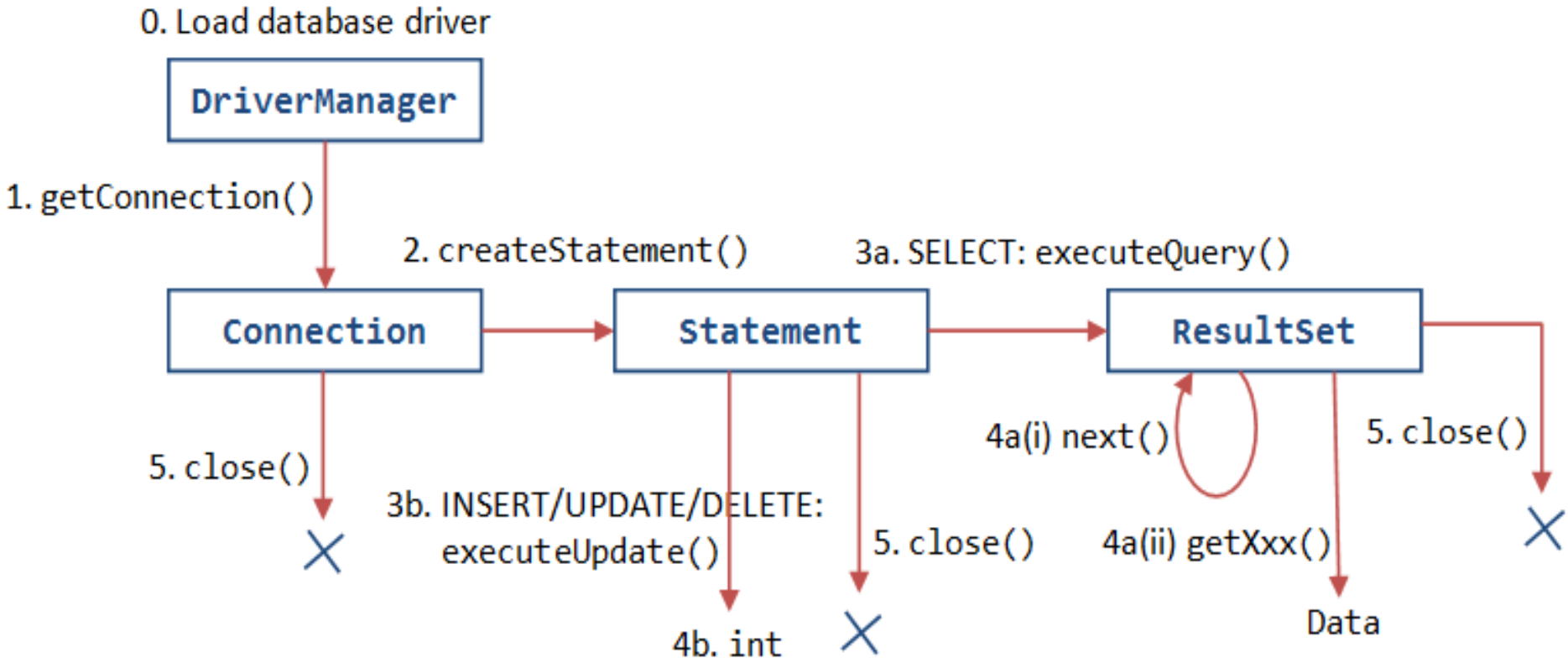
- **Update the database**
  - Another very common JDBC use case is to update the database. Updating the database means writing data to it. In other words, adding new records or modifying (updating) existing records.

- **Perform transactions**
  - Transactions is anoter common use case. A transaction groups multiple updates and possibly queries into a single action. Either all of the actions are executed, or none of them are.

JDBC Interaction Diagram



JDBC Cycle


0. Load database driver
DriverManager
1. getConnection()
2. createStatement()
3a. SELECT: executeQuery()
Connection
Statement
ResultSet
5. close()
3b. INSERT/UPDATE/DELETE:
executeUpdate()
5. close()
4a(i) next()
5. close()
4a(ii) getXxx()
4b. int
Data

Four steps in creating a database application

Step 1: Load a database **driver**

Step 2: Establish a database **connection**

Step 3: Create and execute SQL **statement**

Step 4: Get and process the **result set**, if necessary

Step 5: Close connection

Step 1: Loading a Driver

- Loading a driver requires class name of the driver.
  For JDBC-ODBC driver the class name is:
  sun.jdbc.odbc.JdbcOdbcDriver
  Oracle driver is: **oracle.jdbc.driver.OracleDriver**
- The class definition of the driver is loaded using forName static method of the class Class (in package java.lang)

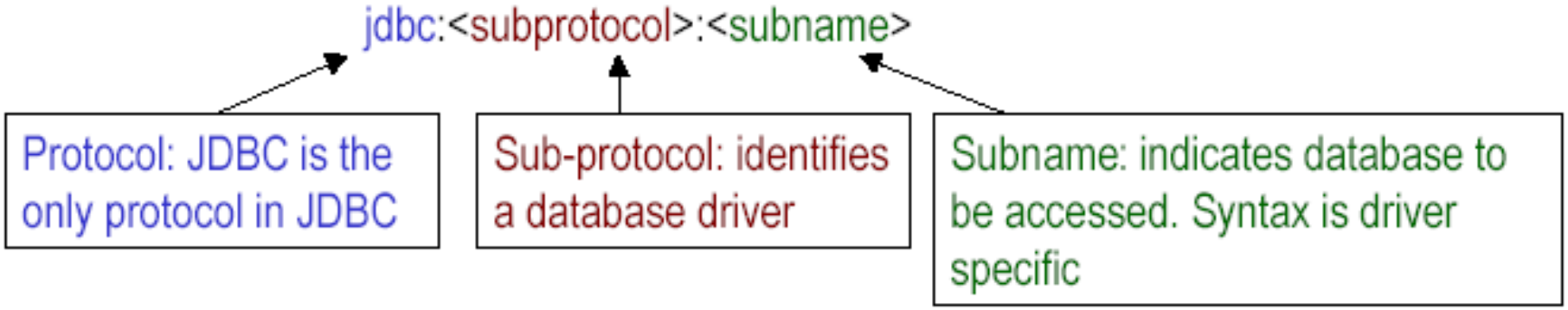
```
try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch (Exception e) {
        out.println( e.getMessage() + "\n Class not found Exception.");
}
```

- The class *DriverManager* manages the loaded drivers

Step 2: Opening a Database Connection

- ***getConnection*** method of ***DriverManager*** class returns a connection
- When there are several databases used by the same application, the driver required to access the database is uniquely identified using JDBC URLs

JDBC URL: Represents a driver and has following three-part syntax



Examples:

"**jdbc:odbc:books**" : specifies database *books* as ODBC data source (books is a logical name linked to actual database)

Step 2: Opening a Database Connection(contd.)

- Getting a connection to ODBC data source *books (MS Access database)*

```
private String url = "jdbc:odbc:books";
private String userName = "anonymous";
private String password = "guest";
Connection connection = DriverManager.getConnection(url, userName, password);
```

- ***DriverManager*** has other variants of ***getConnection*** method that accept different set of parameters
- ***Connection*** is an interface defined in ***java.sql*** package. A Connection object represents a connection with the database. The interface has methods to create statements which can be used to manipulate the database

Before the database *books* can be used here, it must be registered as an ODBC source

Step 3: executing SQL Statements

- Connection objects can be used to create statement objects.

  statement = connection.createStatement();
- Statement is an interface that contains methods for executing SQL queries

```
String sqlStr = "INSERT INTO Authors VALUES('5' , 'Walter', 'Lippman', 'Journalist')";
int rows = statement.executeUpdate(sqlStr);
```

- sqlStr contains a string which is an SQL statement for inserting a new record in the table *Authors* in the *books* database
- The SQL statement is executed using *executeUpdate* method of the statement object
- The method is used to execute statements like **INSERT**, **UPDATE**, **DELETE** that do not return any results. It returns **number of rows** affected

Step 4: Enquiring the Database

- The **Statement** object returns a **java.sql.ResultSet** object upon executing an SQL statement using **executeQuery** method

```
ResultSet rs = statement.executeQuery("SELECT * FROM Authors");
```

- The method returns all rows in *Authors* table as a *ResultSet*
- The **next()** method of **ResultSet** allows to move from one row to the next row

```
while (rs.next()) {
 // rs stands for a row in each iteration; print author details
}
```

- ***ResultSet*** contains several methods for extracting various fields in a row. The methods require column name or column index as an argument.

```
rs.getString( "FirstName" )
```

FirstName is the colum name. The method returns author's first name which is a string

JDBC Example

```
public static void main(String[] args) {
   String url = "jdbc:odbc:sach";
   String userName = "ltmang";
   String password = "ltmang";
   try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection connection =
        DriverManager.getConnection(url,userName,password);
        Statement statement = connection.createStatement();
        String sql = "select * from sach";
        ResultSet rs = statement.executeQuery(sql);
        while (rs.next()) {
          System.out.println(rs.getString( "tens" ));
      }
```

JDBC Example

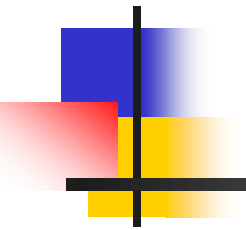
```
        sql = "insert into sach values('P6','Tu hoc Internet'," +
                                        "'Nha xuat ban lao đong')";
        statement.executeUpdate(sql);
        sql = "select * from sach";
        rs = statement.executeQuery(sql);
        while (rs.next()) {
              System.out.println(rs.getString( "tens" ));
        }
        rs.close();
        statement.close();
        connection.close();
    }
    catch (Exception e) {
          System.out.println( e.getMessage());
    }
}
```

Register a database as an ODBC Data Source

- The computer should have Microsoft Access installed
- Invoke *ODBC Data Source Administrator:*
  - In Windows Control Panel, double click "*ODBC Data Sources*". • The dialog is used to register *user data source name*. The tab User DSN must be selected
  - Since a new data source is to be created, click Add in the dialog. *Create New Data Source* dialog appears.
  - There are several drivers listed in the dialog including dBase, Oracle, Microsoft Access etc. *books* is a Microsoft Access database. So, select Microsoft Access Driver and click Finish.
  - Another dialog ODBC Microsoft Access Setup appears.
  - In the field Data Source Name enter a name by which the database is to be referred in JDBC program. In the example, the name *books* is used.
  - This name should be associated with an actual database. Click Select button. This displays Select Database dialog which allows to select a database on the local file system or across the network.
  - Click OK to dismiss the Select Database dialog and return to ODBC Microsoft Access Setup

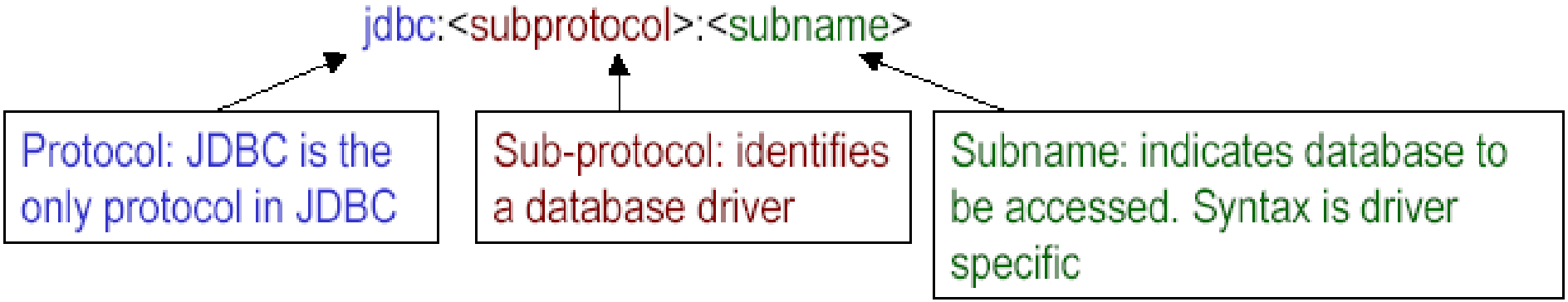
Registering as an ODBC Data Source

• Click **Advanced** button in ODBC Microsoft Access Setup dialog. Set Advanced Options dialog appears
• Enter authorisation information **Login name** and **Password**. In this example, the login name is *anonymous* and password is *guest*
• Click OK to dismiss the dialog
• Dismiss the ODBC Microsoft Access Setup dialog by clicking OK.
• Now ODBC Data Source Administrator dialog contains data source *books* with *Microsoft Access Driver* associated with it.
• Now the database can be accessed using JDBC-ODBC bridge driver



Basic JDBC Programming Concepts

JDBC URLs



- For databases on the Internet/intranet, the subname can contain the Net URL //hostname:port/... The <subprotocol> can be any name that a database understands. The odbc subprotocol name is reserved for ODBC-style data sources. A normal ODBC database JDBC URL looks like: jdbc:odbc:<>;User=<>;PW=<>
- **jdbc:postgresql://www.hcmuaf.edu.vn/ts**

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://example.com:4333/empl`**

Making the Connection

- **Loading a driver:**
  - **Class.forName(className)**
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  - **System.setProperty("jdbc.drivers", "className");**
    System.setProperty("jdbc.drivers",
    "sun.jdbc.odbc.JdbcOdbcDriver");
- **Making a connection**
  - **Connection conn = DriverManager.getConnection(...);**

```
static Connection getConnection(String url);
static Connection getConnection(String url,
                                String user,String password);
static int getLoginTimeout(); // seconds
static void setLoginTimeout(int seconds);
.....
```

JDBC URL Formats for Oracle

- The Thin driver offers several kinds of JDBC URL formats:

1. Connect to Oracle Database SID

```
jdbc:oracle:thin:[<user>/<password>]@<host>[:<port>]:<SID>
```

```
String oracleJdbcUrl =
"jdbc:oracle:thin:@myoracle.db.server:1521:my_sid“;
      String username = "dbUser";
      String password = "1234567";
      Connection conn = DriverManager.getConnection(
                  oracleJdbcUrl, username, password)
```

JDBC URL Formats for Oracle

- The Thin driver offers several kinds of JDBC URL formats:

2. Connect to Oracle Database Service Name

```
jdbc:oracle:thin:[<user>/<password>]@//<host>[:<port>]/<service>
```

```
String oracleJdbcUrl =
"jdbc:oracle:thin:@//myoracle.db.server:1521/servicename";

...

Connection conn = DriverManager.getConnection(
                  oracleJdbcUrl, username, password)
```

JDBC URL Formats for MySQL

- protocol//[hosts][/database][?properties]

```
Connection conn;
String jdbcUrl =
  "jdbc:mysql://mysql.db.server:3306/
  my_db?useSSL=false&serverTimezone=UTC";
String username = "dbUser";
String password = "1234567";
conn = DriverManager.getConnection(
            jdbcUrl, username, password)
```

Executing SQL Commands

- JDBC supports three types of statements:
  - `Statement` : the SQL is prepared and executed in one step (from the application program point of view)
  - `PreparedStatement`: the driver stores the execution plan handle for later use (SQL command with a parameters).
  - `CallableStatement`: the SQL statement is actually making a call to a stored procedure
- The `Connection` object has the `createStatement`(), `prepareStatement`(), and `prepareCall`() methods to create these `Statement` objects.

Creating and Using Direct SQL Statements

- A **`Statement`** object is created using the **`createStatement`**() method in the **`Connection`** object.
- resultSet executeQuery (String sql)
- int executeUpdate (String sql)
- boolean execute(String sql)
- The **`executeUpdate`** method can execute actions such as INSERT, UPDATE, and DELETE as well as data definition commands such as CREATE TABLE, and DROPTABLE. The **`executeUpdate`** method returns a count of the rows that were affected by the SQL command.
- The **`executeQuery`** method is used to execute SELECT queries. The **`executeQuery`** object returns an object of type **`ResultSet`** that you use to walk through the result a row at a time.

Using executeUpdate method

```
try {
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  Connection connection =
      DriverManager.getConnection(url,userName,password);
  Statement statement = connection.createStatement();
  String sqlCommand ="CREATE TABLE Books(Title CHAR(60),"+
                              "ISBN CHAR(13),Price CURRENCY)";
  statement.executeUpdate(sqlCommand);
  sqlCommand =" INSERT INTO Books VALUES(" +
                        "'Beyond HTML', '0-07-882198-3', 27.95)";
  statement.executeUpdate(sqlCommand);
  statement.close();
  connection.close();
}  catch (Exception e) {
       System.out.println( e.getMessage() );
}
```

Using executeQuery method

```
try {
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  Connection connection =
        DriverManager.getConnection(url, userName,password);
  Statement statement = connection.createStatement();
  String sqlCommand = "SELECT * FROM BOOKS";
  ResultSet rs = statement.executeQuery(sqlCommand);
  while (rs.next()) {
        String title = rs.getString(1);
        String isbn = rs.getString(2);
        float price = rs.getFloat("Price");
        System.out.println(title + ", " + isbn + " , " + price);
  }
  rs.close();
} catch (Exception e){};
```

SQL data types and corresponding Java types

SQL data type	Java data type
INTEGER, INT, SMALLINT	int
DOUBLE, FLOAT	double
CHARACTER(*n*) or CHAR(*n*)	String
VARCHAR(*n*)	String
TIMESTAMP	java.sql.Timestamp
DATE	java.sql.Date
DEC, DECIMAL, NUMBER, NUMERIC	java.math.BigDecimal
NCHAR, NVARCHAR2	oracle.sql.NString
BOOLEAN	Boolean

PreparedStatement

- The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query: "`insert into emp values(?,?,?)`"
- You can use a **PreparedStatement** instead of a **Statement** and benefit from the features of the PreparedStatement.
  - Easy to insert parameters into the SQL statement.
  - Easy to reuse the PreparedStatement with new parameters.
  - May increase performance of executed statements.
  - Enables easier batch updates.

Creating and Using PreparedStatement

- The **PreparedStatement**, the driver actually sends only the execution plan ID and the parameters to the DBMS.
- The **PreparedStatement** should be used when you need to execute the SQL statement many times in a Java application.
- The DBMS discards the execution plan at the end of the program.
- The **PreparedStatement** object achieves faster SQL execution performance than the simple Statement object, as the DBMS does not have to run through the steps of creating the execution plan.

Creating and Using PreparedStatement

- Notice that the **executeQuery**(), **executeUpdate**(), and **execute**() methods do not take any parameters.
- ***Return Type Method Name Parameter***

```
ResultSet   executeQuery  ( )
int         executeUpdate( )
boolean     execute       ( )
```

- One of the major features of a `PreparedStatement` is that it can handle `IN` types of parameters. The parameters are indicated in a SQL statement by placing the **?** as the parameter marker instead of the actual values.

Creating and Using PreparedStatement

- In the Java program, the association is made to the parameters with the **`setXXXX`**(..,..) methods. All of the **`setXXXX`**() methods take the parameter index, which is 1 for the first "?," 2 for the second "?,"and so on.
- void setBoolean (int parameterIndex, boolean x)
- void setByte (int parameterIndex, byte x)
- void setDouble (int parameterIndex, double x)
- void setFloat (int parameterIndex, float x)
- void setInt (int parameterIndex, int x)
- void setLong (int parameterIndex, long x)
- void setNumeric (int parameterIndex, Numeric x)
- void setShort (int parameterIndex, short x)
- void setString (int parameterIndex, String x)

Creating and Using PreparedStatement

```
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection connection =
      DriverManager.getConnection(url,userName,password);
     PrepStmt = connection.prepareStatement("SELECT * FROM
                                          BOOKS WHERE ISBN = ?");
     PrepStmt.setString(1,"0-07-882198-3");
     ResultSet rs = PrepStmt.executeQuery();
     while (rs.next()) {
          title = rs.getString(1);
          isbn = rs.getString(2);
          price = rs.getFloat("Price");
          System.out.println(title + ", " + isbn + " , " + price );
      }
      rs.close();
      PrepStmt.close();
      connection.close();
}
```

java.sql.DriverManager API

- static Connection getConnection(String url, String user,String password)
  establishes a connection to the given database and returns a Connection object.

java.sql.Connection API

- **Statement createStatement()**
  creates a statement object that can be used to execute SQL queries and updates without parameters.
- **PreparedStatement prepareStatement(String sql)**
  returns a PreparedStatement object containing the precompiled statement. The string sql contains a SQL statement that can contain one or more parameter placeholders denoted by ? characters.
- **void close()**
  immediately closes the current connection.

java.sql.Statement API

- **close()**
  immediately closes the current result set.
- **ResultSet executeQuery(String sql)**
  executes the SQL statement given in the string and returns a ResultSet to view the query result.
- **int executeUpdate(String sql)**
  executes the SQL INSERT, UPDATE, or DELETE statement specified by the string. Also used to execute Data Definition Language (DDL) statements such as CREATE TABLE. Returns the number of records affected, or -1 for a statement without an update count.

java.sql.Statement API

- **boolean execute(String sql)**
  executes the SQL statement specified by the string. Returns true if the statement returns a result set, false otherwise. Use the getResultSet or getUpdateCount method to obtain the statement outcome.
- **int getUpdateCount()**
  Returns the number of records affected by the preceding update statement, or -1 if the preceding statement was a statement without an update count. Call this method only once per executed statement.
- **ResultSet getResultSet()**
  Returns the result set of the preceding query statement, or null if the preceding statement did not have a result set. Call this method only once per executed statement.

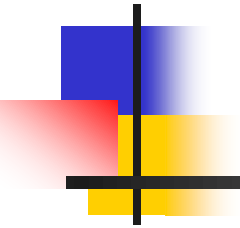
java.sql.PreparedStatement API

- **`void setXxx(int n, Xxx x)`**
  (Xxx is a type such as int, double, String, Date, etc.) sets the value of the n-th parameter to x.
- **`void clearParameters()`**
  clears all current parameters in the prepared statement.
- **`ResultSet executeQuery()`**
  executes a prepared SQL query and returns a ResultSet object.
- **`int executeUpdate()`**
  executes the prepared SQL INSERT, UPDATE, or DELETE statement represented by the PreparedStatement object. Returns the number of rows affected, or 0 for DDL statements.

java.sql.ResultSet API

- **`boolean next()`**
  makes the current row in the result set move forward by one. Returns false after the last row. Note that you must call this method to advance to the first row.
- **`xxx getXxx(int columnNumber)`**
- **`xxx getXxx(String columnName)`**

  (xxx is a type such as int, double, String, Date, etc.) return the value of the column with column index columnNumber or with column names, converted to the specified type. Not all type conversions are legal. See documentation for details.
- **`int findColumn(String columnName)`**
  gives the column index associated with a column name.



Scrollable and Updatable Result Sets

JDBC versions

- **Version 1.0 (1997)**: Include how to create a driver instance, create a database connection (via Connection object) execute SQL statements (via Statement object), return the results (through a ResultSet object), and metadata of the database and the ResultSet.
- **Version 2.0/2.1** (integrated into JDK 1.2): JDBC 2.0 introduces many new features to improve the performance and functionality. For example, it introduces scrollable and updatable ResultSet, batching processing.

JDBC versions

- **Version 3.0** (integrated into JDK 1.4): Transaction Savepoints
- **Version 4.0** (integrated into JDK 6): Autoloading of JDBC drivers. In earlier versions of JDBC, applications had to manually register drivers before requesting Connections. With JDBC 4.0 and above, applications no longer need to issue a Class.forName() on the driver name; instead, the DriverManager will find an appropriate JDBC driver when the application requests a Connection.

Scrollable and Updatable Result Sets

- You usually want the user to be able to move both forward and backward in the result set. But in JDBC 1.0, there was no previous method.The scrolling result set in JDBC 2.0 lets you **move forward** and **backward** through a result set and **jump** to **any position in the result set**.
- Furthermore, once you display the contents of a result set to users, they may be tempted to edit it. If you supply an editable view to your users, you have to make sure that the user edits are posted back to the database. In JDBC 1.0, you had to program UPDATE statements. In **JDBC 2.0**, you can simply **update the result set entries**, and **the database is automatically updated**.

Scrollable Result Sets (JDBC 2.0)

To obtain scrolling result sets from your queries, you must obtain a **different Statement** object with the method

- Statement stat = **conn.createStatement(type, concurrency);**
- PreparedStatement stat = **conn.prepareStatement(command, type, concurrency**);
- **ResultSet type values**:

TYPE_FORWARD_ONLY : The result set is not scrollable.

TYPE_SCROLL_INSENSITIVE: The result set is scrollable but not sensitive to database changes.

TYPE_SCROLL_SENSITIVE : The result set is scrollable and sensitive to database changes.

- **ResultSet concurrency values:**

CONCUR_READ_ONLY :The result set cannot be used to update the database.

CONCUR_UPDATABLE :The result set can be used to update the database.

Scrollable Result Sets (JDBC 2.0)

- For example, if you simply want to be able to scroll through a result set but you don't want to edit its data, you use:
- `Statement stat = conn.createStatement(`
  `TYPE_SCROLL_INSENSITIVE,CONCUR_READ_ONLY);`
- All result sets that are returned by method calls
  `ResultSet rs = stat.executeQuery(query)`
  are now scrollable. A scrolling result set has a *cursor* that indicates the current position.
- Scrolling is very simple. You use

  ```
  if (rs.previous()) . . .
  ```

  to scroll backward. The method returns true if the cursor is positioned on an actual row; false if it now is positioned before the first row.

Scrollable Result Sets (JDBC 2)

- Move the cursor backward or forward by a number of rows with the command

```
rs.relative(n);
```

  - If *n* is positive, the cursor moves forward
  - If *n* is negative, it moves backwards.
  - If *n* is zero, the call has no effect.
  - If you attempt to move the cursor outside the current set of rows, then, the method returns `false` and the cursor does not move. The method returns `true` if the cursor landed on an actual row.
- Set the cursor to a particular row number:

```
rs.absolute(n);
```

- **The first row in the result set has number 1**. If the return value is 0, the cursor is not currently on a row—it is either before the first or after the last row.

Updatable Result Sets (JDBC 2)

- If you want to be able to edit result set data and have the changes automatically reflected in the database, you need to create an updatable result set.
- `Statement stat = conn.createStatement( TYPE_SCROLL_INSENSITIVE,CONCUR_UPDATABLE);`
- Then, the result sets returned by a call to executeQuery are updatable.
- NOTE: Not all queries return updatable result sets. If your query is a join that involves multiple tables, the result may not be updatable.
- For example, suppose you want to raise the prices of some books, but you don't have a simple criterion for issuing an UPDATE command. Then, you can iterate through all books and update prices, based on arbitrary conditions.

Updatable Result Sets (JDBC 2)

```
String query = "SELECT * FROM Books";
ResultSet rs = stat.executeQuery(query);
while (rs.next()) {
    if (. . .){
       double increase = . . .
       double price = rs.getDouble("Price");
        rs.updateDouble("Price", price + increase);
        rs.updateRow();
    }
 }
```

- There are **update*Xxx*** methods for all data types that correspond to SQL types, such as **updateDouble**, **updateString**, and so on. As with the **getXxx** methods, you specify the name or the number of the column. Then, you specify the new value for the field.

Updatable Result Sets (JDBC 2)

- The **update*Xxx*** method only changes the row values, not the database. When you are done with the field updates in a row, you must call the **updateRow** method. That method sends all updates in the current row to the database.
- If you move the cursor to another row **without calling updateRow**, all **updates are discarded** from the row set and they are never communicated to the database. You can also call the **cancelRowUpdates** method to cancel the updates to the current row.
- If you want to add a new row to the database, you first use the **moveToInsertRow** method to move the cursor to a special position, called the **insertRow**. You build up a new row in the insert row position by issuing **updateXxx** instructions. Finally, when you are done, call the **insertRow** method to deliver the new row to the database. When you are done inserting, call **moveToCurrentRow** to move the cursor back to the position before the call to **moveToInsertRow**.

Updatable Result Sets (JDBC 2)

```
rs.moveToInsertRow();
rs.updateString("Title", title);
rs.updateString("ISBN", isbn);
rs.updateDouble("Price", price);
rs.insertRow();
rs.moveToCurrentRow();
```

- Finally, you can delete the row under the cursor.

```
rs.deleteRow();
```

- The **deleteRow** method immediately removes the row from both the result set and the database.
- The **updateRow**, **insertRow**, and **deleteRow** methods of the ResultSet class give you the same power as executing **UPDATE**, **INSERT**, and **DELETE** SQL commands.

javax.sql.Connection API

- Statement createStatement(int type, int concurrency)
- PreparedStatement prepareStatement(String command, int type, int concurrency)
- (JDBC 2) create a statement or prepared statement that yields result sets with the given type and concurrency.
- *Parameters:*

  command : the command to prepare

  type : one of the constants **TYPE_FORWARD_ONLY**, **TYPE_SCROLL_INSENSITIVE**, or **TYPE_SCROLL_SENSITIVE**

  concurrency: one of the constants **CONCUR_READ_ONLY** or **CONCUR_UPDATABLE**

java.sql.ResultSet API

- **int getType()**
  Returns the type of this result set, one of TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, or TYPE_SCROLL_SENSITIVE.
- **int getConcurrency()**
  Returns the concurrency setting of this result set, one of CONCUR_READ_ONLY or CONCUR_UPDATABLE.
- **boolean previous()**
  Moves the cursor to the preceding row. Returns true if the cursor is positioned on a row.
- **int getRow()**
  Gets the number of the current row. Rows are numbered starting with 1.
- **boolean absolute(int row)**
  Moves the cursor to row row. Returns true if the cursor is positioned on a row.

java.sql.ResultSet API

- **boolean relative(int d)**
  Move the cursor by d rows. If d is negative, the cursor is moved backward. Returns true if the cursor is positioned on a row.
- **boolean first()**
- **boolean last()**
  Move the cursor to the first or last row. Return true if the cursor is positioned on a row.
- **void beforeFirst()**
- **void afterLast()**
  Move the cursor before the first or after the last row.
- **boolean isFirst()**
- **boolean isLast()**
  Test if the cursor is at the first or last row.

java.sql.ResultSet API

- **boolean isBeforeFirst()**
- **boolean isAfterLast()**
  Test if the cursor is before the first or after the last row.
- **void moveToInsertRow()**
  Moves the cursor to the *insert row.* The insert row is a special row that is used for inserting new data with the **updateXxx** and **insertRow** methods.
- **void moveToCurrentRow()**
  Moves the cursor back from the insert row to the row that it occupied when the **moveToInsertRow** method was called.
- **insertRow()**
  Inserts the contents of the insert row into the database and the result set.

java.sql.ResultSet API

- **void deleteRow()**
  Deletes the current row from the database and the result set.
- **void updateXxx(int column, Xxx data)**
- **void updateXxx(String columnName, Xxx data)**
  (**Xxx** is a type such as **int**, **double**, **String**, **Date**, etc.) Update a field in the current row of the result set.
- **void updateRow()**
  Sends the current row updates to the database.
- **void cancelRowUpdates()**
  Cancels the current row updates.

Database Metadata

- JDBC can give you additional *information* about the *structure* of a database and its tables. For example, you can get a list of the tables in a particular database or the column names and types of a table.
- To find out more about the database, you need to request an object of type DatabaseMetaData from the database connection.
- ```
  DatabaseMetaData databaseMetaData =
          connection.getMetaData();
  ```
  ```

Tables Metadata

- Sometimes, we want to know the names of all the user-defined tables, system tables, or views. Also, we may like to know some explanatory comments on the tables. All of this can be done by using the *getTables*() method of the *DatabaseMetaData* object
- ```
  ResultSet resultSet =
  databaseMetaData.getTables(null, null, null,
  new String[]{"TABLE"});
  ```
- Use ”***TABLE***” for user-defined tables.
- Use ”**SYSTEM *TABLE***” for system-defined tables.
- Use “**VIEW**” to find out all the existing views
  ```

Extract Table Info - System Table

```
databaseMetaData = connection.getMetaData();
//Print TABLE_TYPE "TABLE"
ResultSet resultSet =
    databaseMetaData.getTables(null, null, null,
                new String[]{"SYSTEM TABLE"});
System.out.println("Printing SYSTEM TABLE");
while(resultSet.next()){
    //Print
    System.out.println(
                resultSet.getString("TABLE_NAME"));
}
```

Extract Table Info - Table

```
databaseMetaData = connection.getMetaData();
//Print TABLE_TYPE "TABLE"
ResultSet resultSet =
    databaseMetaData.getTables(null, null, null,
                               new String[]{"TABLE"});
System.out.println("Printing TABLE");
while(resultSet.next()){
    //Print
    System.out.println(
                resultSet.getString("TABLE_NAME"));
}
```

Extract Column Info

```
ResultSet columns = databaseMetaData.getColumns(null,
                                    null, tableName, null);
while(columns.next()){
    String columnName = columns.getString("COLUMN_NAME");
    String datatype = columns.getString("DATA_TYPE");
    String columnsize = columns.getString("COLUMN_SIZE");
    String decimaldigits = columns.getString("DECIMAL_DIGITS");
    String isNullable = columns.getString("IS_NULLABLE");
    String is_autoIncrment =
                    columns.getString("IS_AUTOINCREMENT");
    //Printing results
    System.out.println();
}
```

ResultSet Metadata

- The ResultSetMetaData provides information about the obtained ResultSet object like, the number of columns, names of the columns, datatypes of the columns, name of the table etc…

```
ResultSet rs = stat.executeQuery(
                    "SELECT * FROM " +     tableName);
ResultSetMetaData metaData = rs.getMetaData();
int columnCount = metaData.getColumnCount();
for (int i = 1; i <= columnCount; i++) {
    String columnLabel = metaData.getColumnLabel(i);
    String columnName = metaData.getColumnName(i);
    int columnWidth = meta.getColumnDisplaySize(i);
 }
```

java.sql.Connection

- **DatabaseMetaData getMetaData()**
  returns the metadata for the connection as a DataBaseMetaData object.

java.sql.DatabaseMetaData

- **`ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String types[])`**
  gets a description of all tables in a catalog that match the schema and table name patterns and the type criteria. The `catalog` and `schema` parameters can be `""` to retrieve those tables without a catalog or schema, or `null` to return tables regardless of catalog or schema.
- The `types` array contains the names of the table types to include. Typical types are `TABLE`, `VIEW`, `SYSTEM TABLE`, `GLOBAL TEMPORARY`, `LOCAL TEMPORARY`, `ALIAS`, and `SYNONYM`. If `types` is `null`, then tables of all types are returned.

java.sql.DatabaseMetaData

- The `types` array contains the names of the table types to include. Typical types are `TABLE`, `VIEW`, `SYSTEM TABLE`, `GLOBAL TEMPORARY`, `LOCAL TEMPORARY`, `ALIAS`, and `SYNONYM`. If `types` is `null`, then tables of all types are returned.
- The result set has five columns, all of which are of type String:

1	`TABLE_CAT`	Table catalog (may be `null`)
2	`TABLE_SCHEM`	Table schema (may be `null`)
3	`TABLE_NAME`	Table name
4	`TABLE_TYPE`	Table type
5	`REMARKS`	Comment on the table

java.sql.DatabaseMetaData

- `int getJDBCMajorVersion()`
- `int getJDBCMinorVersion()`

  (JDBC 3) Return the major and minor JDBC version numbers of the driver that established the database connection. For example, a JDBC 3.0 driver has major version number 3 and minor version number 0.

- `int getMaxStatements()`

  Returns the maximum number of concurrently open statements per database connection, or 0 if the number is unlimited or unknown.

java.sql.ResultSet

- **`ResultSetMetaData getMetaData()`**
  gives you the metadata associated with the current `ResultSet` columns.

java.sql.ResultSetMetaData

- **`int getColumnCount()`**
  returns the number of columns in the current `ResultSet` object.
- **`int getColumnDisplaySize(int column)`**
  tells you the maximum width of the column specified by the index parameter.
- **`String getColumnLabel(int column)`**
  gives you the suggested title for the column.
- **`String getColumnName(int column)`**
  gives the column name associated with the column index specified.
- See more...
  https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSetMetaData.html

Transactions

- The major reason for grouping commands into transactions is ***database integrity*.**
- If you group updates to a transaction, then the transaction either succeeds in its entirety and it can be *committed*, or it fails somewhere in the middle. In that case, you can carry out a *rollback* and the database automatically undoes the effect of all updates that occurred since the last committed transaction.
- By default, a database connection is in *autocommit* mode, and each SQL command is committed to the database as soon as it is executed. Once a command is committed, you cannot roll it back.

Transactions

- Transactions enable you to control if, and when, changes are applied to the database. It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.
- To enable manual - transaction support instead of the auto-commit mode that the JDBC driver uses by default, use the Connection object's **setAutoCommit()** method. If you pass a boolean false to **setAutoCommit( ),** you turn off auto-commit. You can pass a boolean true to turn it back on again.
- For example, code the following to turn off auto – commit: **conn.setAutoCommit(false);**

Transactions

Commit & Rollback

- Once you are done with your changes and you want to commit the changes then call **commit()** method on connection object as follows:

```
conn.commit( );
```

- Otherwise, to roll back updates to the database made using the Connection named conn, use the following code:

```
conn.rollback( );
```

Transactions

```
try{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();
   String SQL = "INSERT INTO Employees  " +
                "VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);
   //Submit a malformed SQL statement that breaks
   String SQL = "INSERTED IN Employees  " +
                "VALUES (107, 22, 'Sita', 'Singh')";
   stmt.executeUpdate(SQL);
   // If there is no error.
   conn.commit();
}catch(SQLException se){
   // If there is any error.
   conn.rollback();
}
```

Using Savepoints

- The new **JDBC 3.0** Savepoint interface gives you the additional transactional control. Most modern DBMS, support **savepoints** within their environments such as Oracle's PL/SQL.
- When you **set a savepoint you define a logical rollback point within a transaction**. If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.

Using Savepoints

- The Connection object has two new methods that help you manage savepoints:
  - **setSavepoint(String savepointName):** Defines a new savepoint. It also returns a Savepoint object.
  - **releaseSavepoint(Savepoint savepointName):** Deletes a savepoint. Notice that it requires a Savepoint object as a parameter. This object is usually a savepoint generated by the setSavepoint() method.
- There is one **rollback (String savepointName)** method, which rolls back work to the specified savepoint.

Using Savepoints

```
try{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();
   //set a Savepoint
   Savepoint savepoint1 = conn.setSavepoint("Savepoint1");
   String SQL = "INSERT INTO Employees " +
                "VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);
   //Submit a malformed SQL statement that breaks
   String SQL = "INSERTED IN Employees " +
                "VALUES (107, 22, 'Sita', 'Tez')";
   stmt.executeUpdate(SQL);
   // If there is no error, commit the changes.
   conn.commit();
}catch(SQLException se){
   // If there is any error.
   conn.rollback(savepoint1);
}
```

Batch Updates

- Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.
- The **addBatch()** method of *Statement, PreparedStatement,* and *CallableStatement*is used to add individual statements to the batch. The **executeBatch()** is used to start the execution of all the statements grouped together.
- The **executeBatch()** returns an array of integers, and each element of the array represents the update count for the respective update statement.
- Just as you can add statements to a batch for processing, you can remove them with the **clearBatch()** method. This method removes all the statements you added with the addBatch() method.

Batching with Statement

- Create a Statement object using either createStatement() methods.
- Set auto-commit to false using setAutoCommit().
- Add as many as SQL statements you like into batch using addBatch() method on created statement object.
- Execute all the SQL statements using executeBatch() method on created statement object.
- Finally, commit all the changes using commit() method.

Batching with Statement

```
// Create statement object
Statement stmt = conn.createStatement();
// Set auto-commit to false
conn.setAutoCommit(false);
// Create SQL statement
String SQL = "INSERT INTO Employees . . .  ";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);
// Create one more SQL statement
String SQL = "INSERT INTO Employees . . . ";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);
// Create one more SQL statement
String SQL = "UPDATE Employees SET . . . " ;
// Add above SQL statement in the batch.
stmt.addBatch(SQL);
// Create an int[] to hold returned values
int[] count = stmt.executeBatch();
//Explicitly commit statements to apply changes
conn.commit();
```

Batching with PrepareStatement

- Create SQL statements with placeholders.
- Create PrepareStatement object using either prepareStatement() methods.
- Set auto-commit to false using setAutoCommit().
- Add as many as SQL statements you like into batch using addBatch() method on created statement object.
- Execute all the SQL statements using executeBatch() method on created statement object.
- Finally, commit all the changes using commit() method

Batching with PrepareStatement

```
// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last,
                                     age) VALUES(?, ?, ?, ?)";

// Create PrepareStatement object
PreparedStatemen pstmt = conn.prepareStatement(SQL);

//Set auto-commit to false
conn.setAutoCommit(false);

// Set the variables
pstmt.setInt( 1, 400 );
pstmt.setString( 2, "xxxxx" );
 . . . . . .
// Add it to the batch
pstmt.addBatch();
```

Batching with PrepareStatement

```
// Set the variables
pstmt.setInt( 1, 401 );
pstmt.setString( 2, "zzzzz" );
. . . . . .
// Add it to the batch
pstmt.addBatch();

//add more batches
. . . . .
//Create an int[] to hold returned values
int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes
conn.commit();
```