Advanced java Programming
GTU #3160707

Dedicated Faculty, Committed Education
Darshan
Institute of Engineering & Technology

Unit-2

JDBC Programming

Prof. Jayesh D. Vagadiya
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot

✉ Jayesh.vagadiya@darshan.ac.in
☎ 9537133260

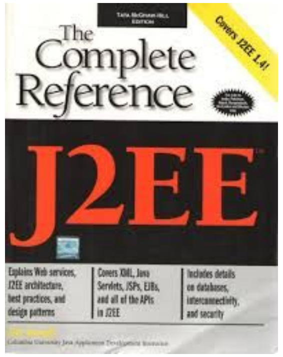
Reference Books

Sr. No.	Unit	Reference Book	Chapter
1	Java Networking	The Complete Reference, Java (Seventh Edition), Herbert Schild - Osbrone.	20
2	JDBC Programming	Complete Reference J2EE by James Keogh mcgraw publication	6,7
3	Servlet API and Overview	Professional Java Server Programming by Subrahmanyam Allamaraju, Cedric Buest Wiley Publication	7,8
4	Java Server Pages		10,11
5	Java Server Faces	Black Book " Java server programming" J2EE, 1st ed., Dream Tech Publishers, 2008. 3. Kathy walrath "	11
6	Hibernate		15
7	Java Web Frameworks: Spring MVC		21

Subject Overview

Sr. No.	Unit	% Weightage
1	Java Networking	5
2	JDBC Programming	10
3	Servlet API and Overview	25
4	Java Server Pages	25
5	Java Server Faces	10
6	Hibernate	15
7	Java Web Frameworks: Spring MVC	10

Reference Book:
Complete Reference J2EE by James Keogh mcgraw publication



Introduction

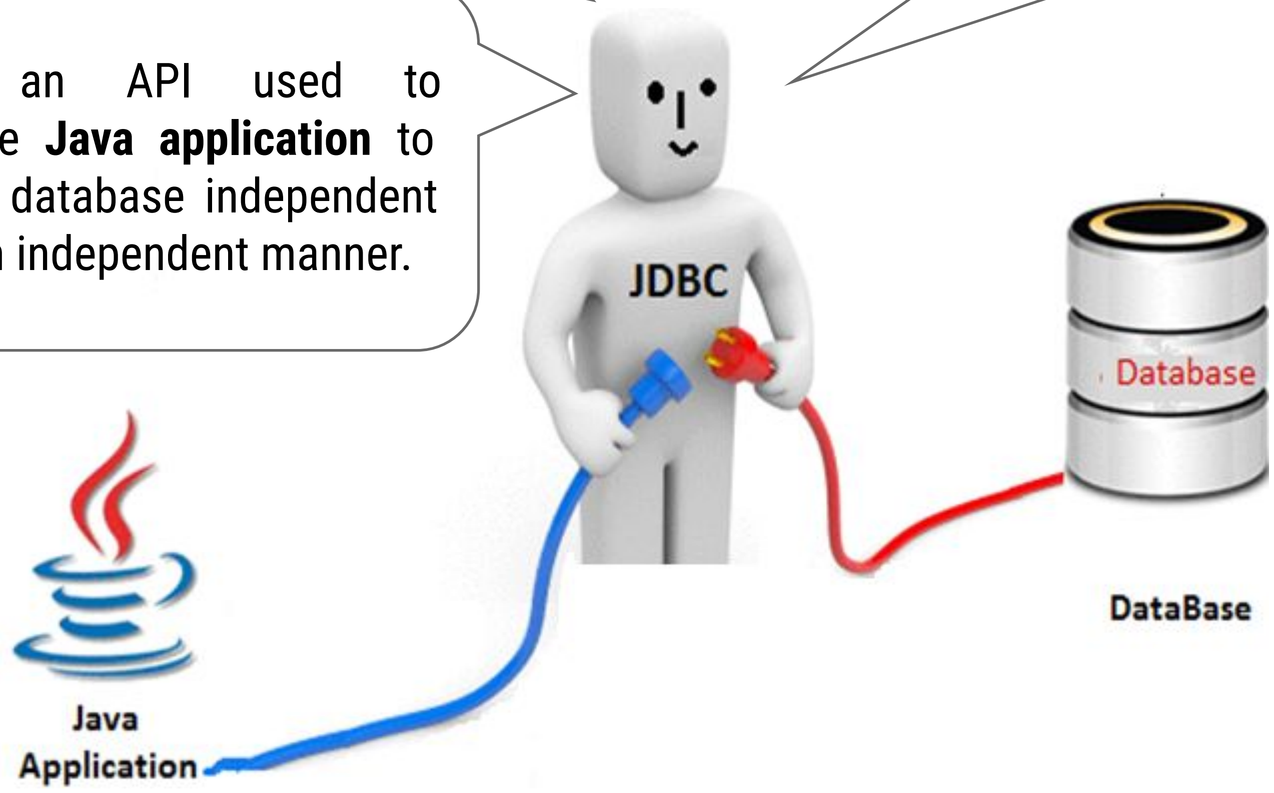
- Database
  - Collection of data
- DBMS
  - Database Management System
  - Storing and organizing data
- SQL
  - Relational database
  - Structured Query Language
- JDBC
  - Java Database Connectivity
  - JDBC driver

Introduction: JDBC

JDBC (Java Database Connectivity) is used to connect java application with database.

It provides **classes** and **interfaces** to connect or communicate Java application with database.

JDBC is an API used to communicate **Java application** to **database** in database independent and platform independent manner.



Example
Oracle
MS Access
My SQL
SQL Server
..
.

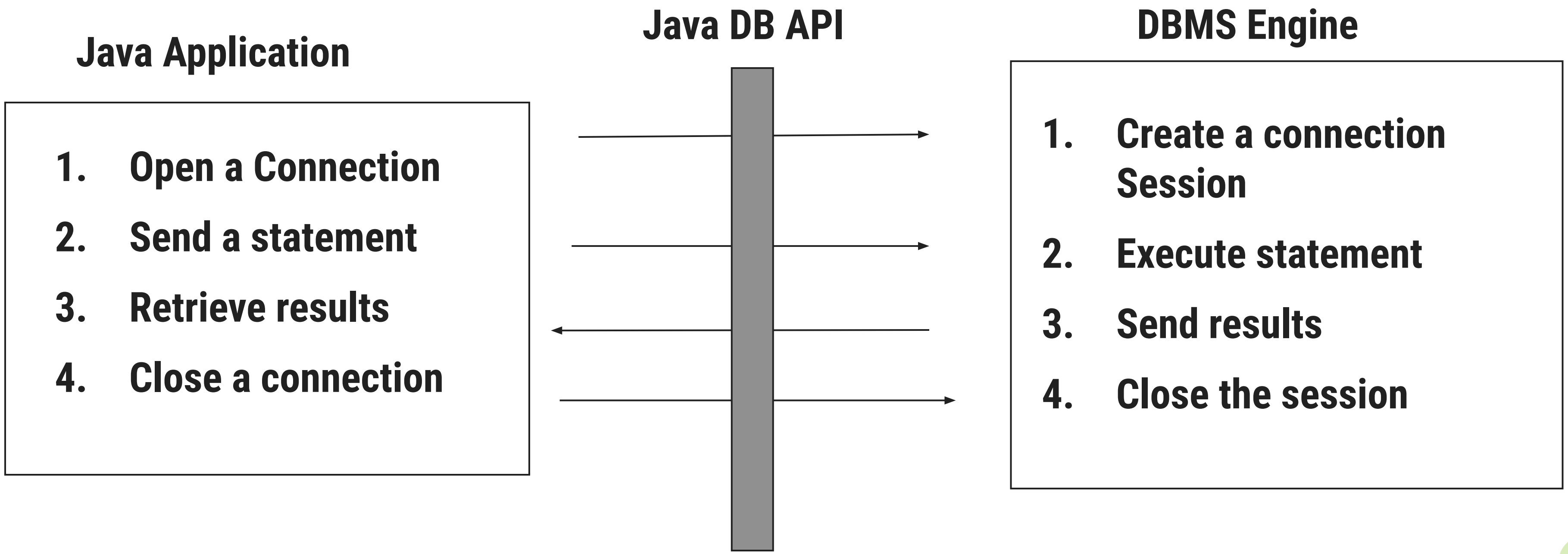
Introduction: JDBC

- JDBC (Java Database Connection) is the standard method of accessing **databases** from **Java application**.
- JDBC is a specification from **Sun Microsystem** that provides a **standard API** for java application to communicate with different database.
- JDBC is a **platform independent** interface between relational database and java applications.

- **What is API ?**
  - ***A**pplication **P**rogram **I**nterface*
  - A set of routines, protocols, and tools for building software applications.
  - JDBC is an API, which is used in java programming for interacting with database.

Introduction: JDBC API

- JDBC API allows java programs to
  - Make a connection with database
  - Creating SQL statements
  - Execute SQL statement
  - Viewing & Modifying the resulting records



Java Application

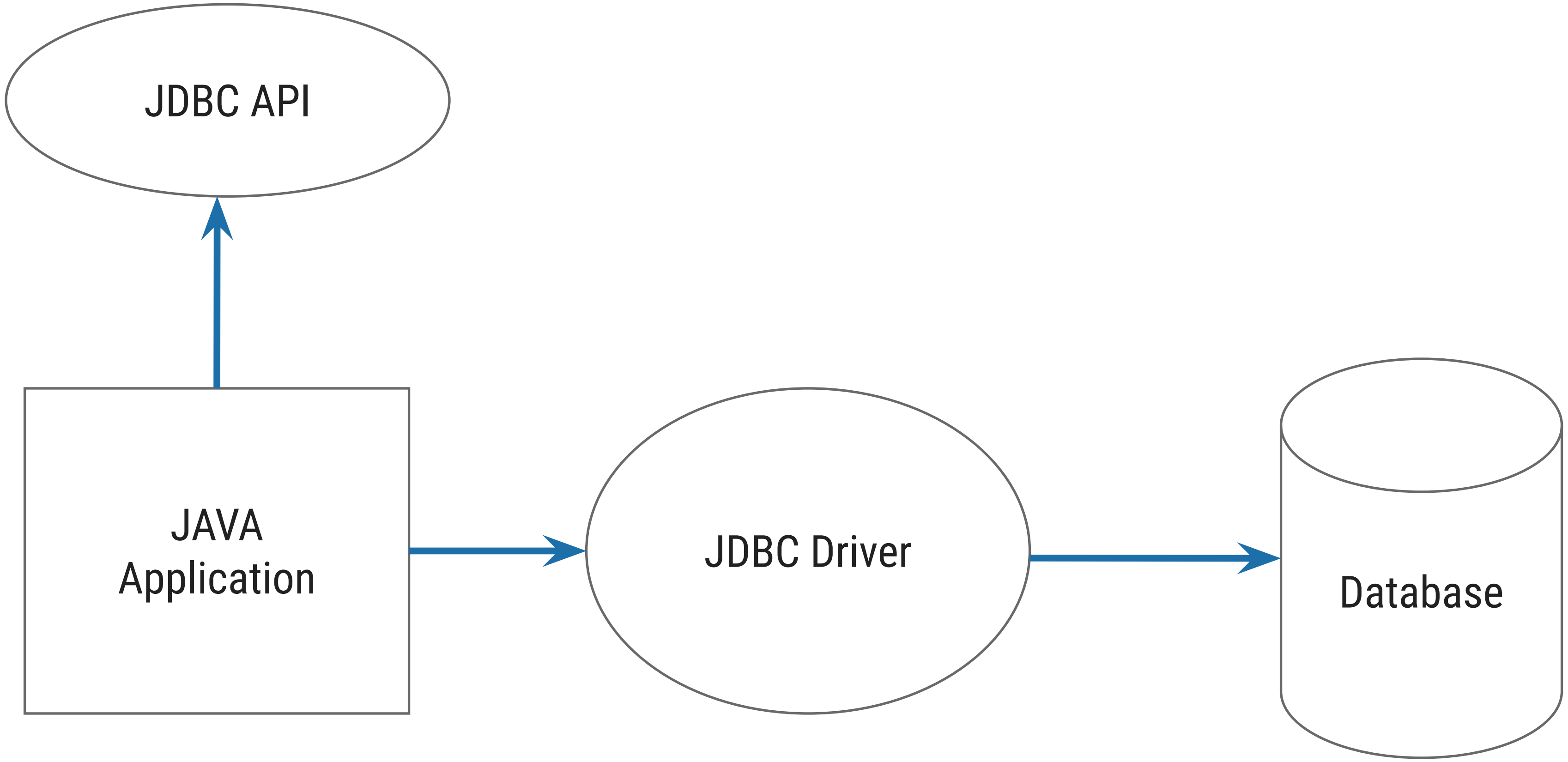
1. Open a Connection
2. Send a statement
3. Retrieve results
4. Close a connection

Java DB API

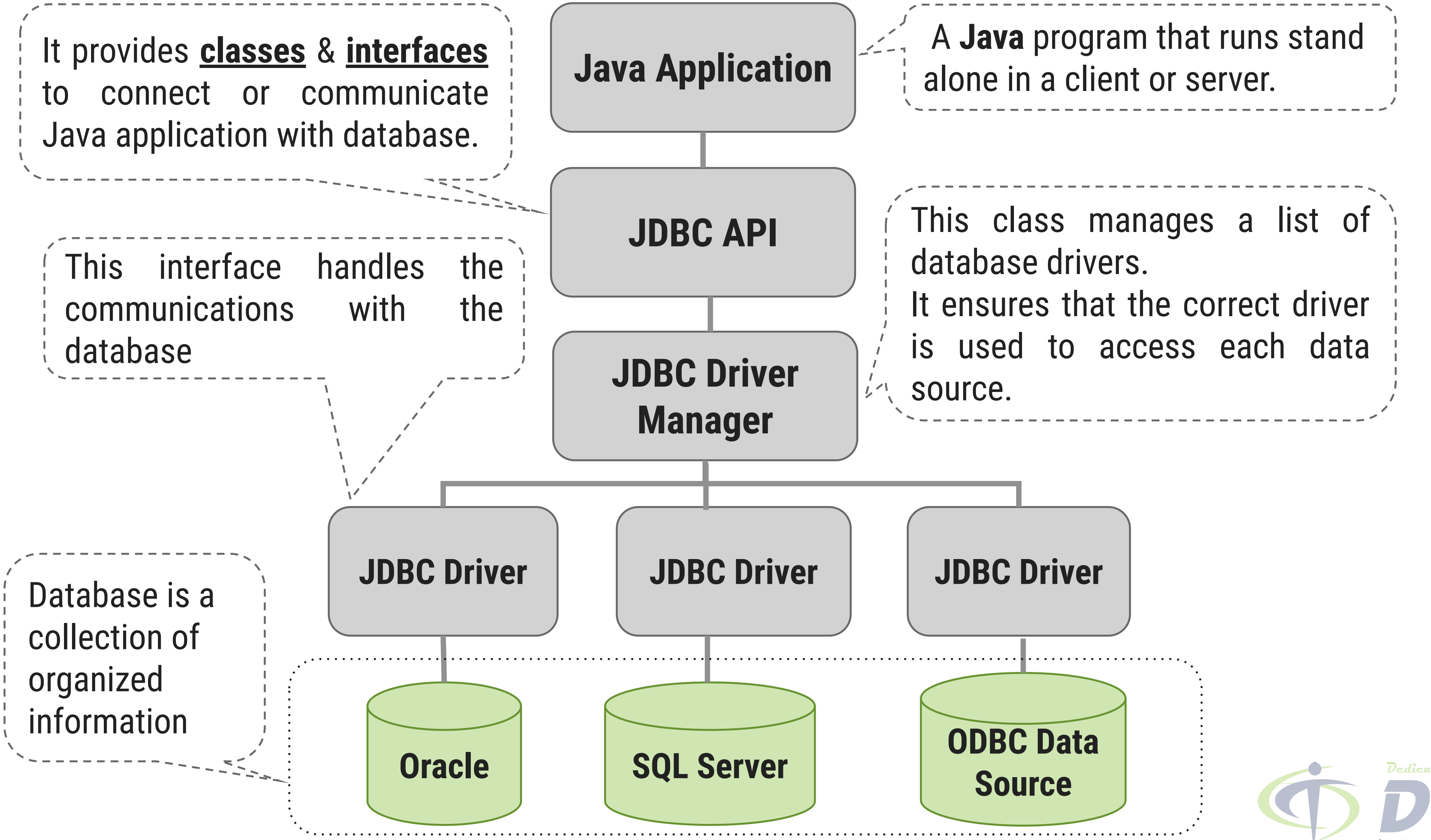
DBMS Engine

1. Create a connection Session
2. Execute statement
3. Send results
4. Close the session

The JDBC Connectivity Model



JDBC Architecture



It provides **classes** & **interfaces** to connect or communicate Java application with database.

A **Java** program that runs stand alone in a client or server.

This interface handles the communications with the database

This class manages a list of database drivers.
It ensures that the correct driver is used to access each data source.

Database is a collection of organized information

JDBC Driver

- **API:** Set of interfaces independent of the RDBMS
- **Driver:** RDBMS-specific implementation of API interfaces e.g. Oracle, DB2, MySQL, etc.

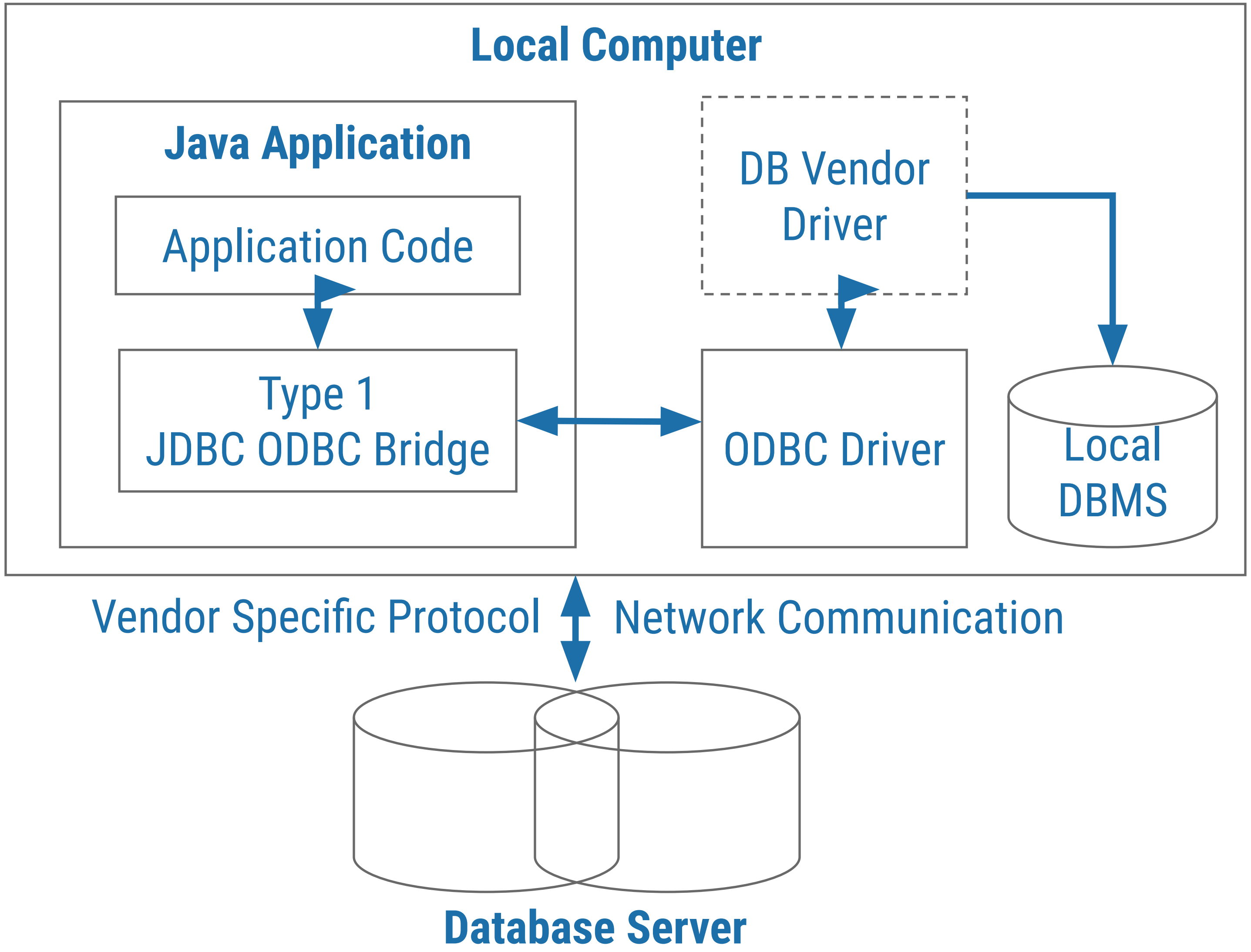
> ***Just like Java aims for "Write once, Run anywhere", JDBC strives for "Write once, Run with any database".***

JDBC Driver: Type 1 (JDBC-ODBC Driver)

- Depends on support for ODBC
- Not portable
- Translate JDBC calls into ODBC calls and use Windows ODBC built in drivers
- ODBC must be set up on every client
  - for server side servlets ODBC must be set up on web server
- driver sun.jdbc.odbc.JdbcOdbc provided by JavaSoft with JDK
- No support from JDK 1.8 (Java 8)

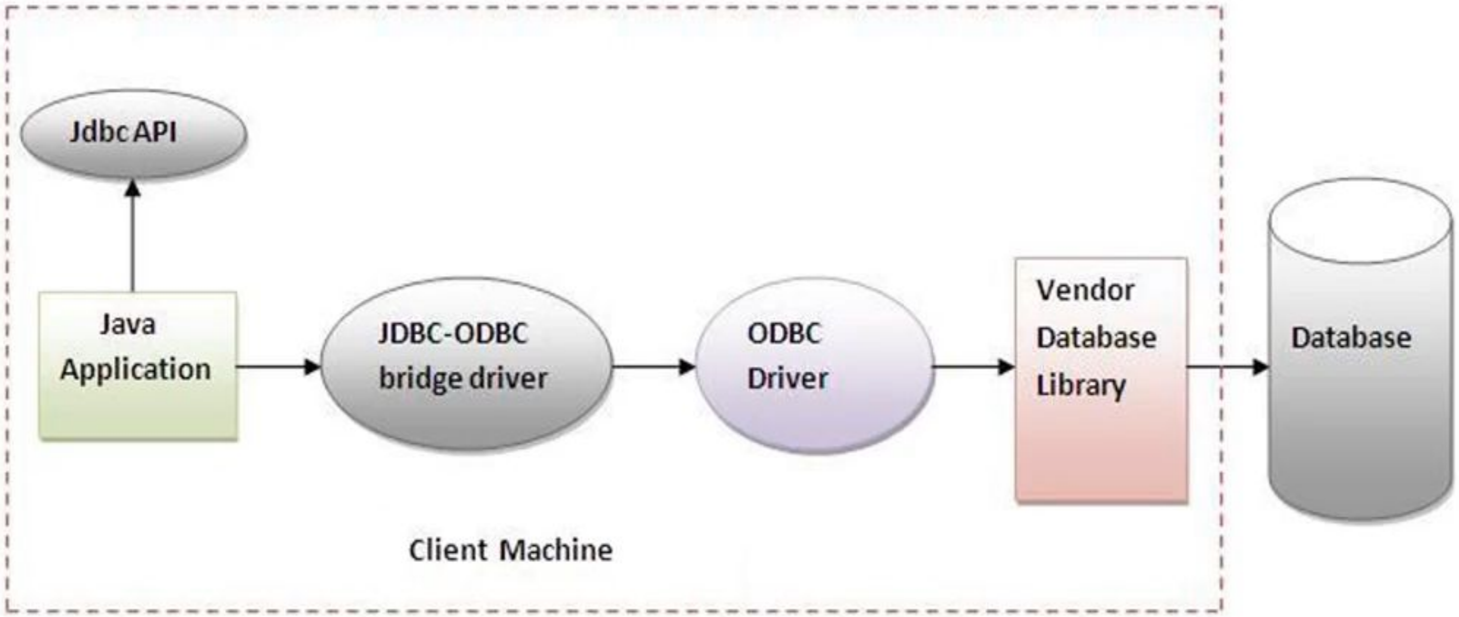
E.g. MS Access

JDBC Driver: Type 1 (JDBC-ODBC Driver)



JDBC Driver: Type 1 (JDBC-ODBC Driver)

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



JDBC Driver: Type 1 (JDBC-ODBC Driver)

Advantages :

- Allow to communicate with all database supported by ODBC driver
- It is vendor independent driver

Disadvantages:

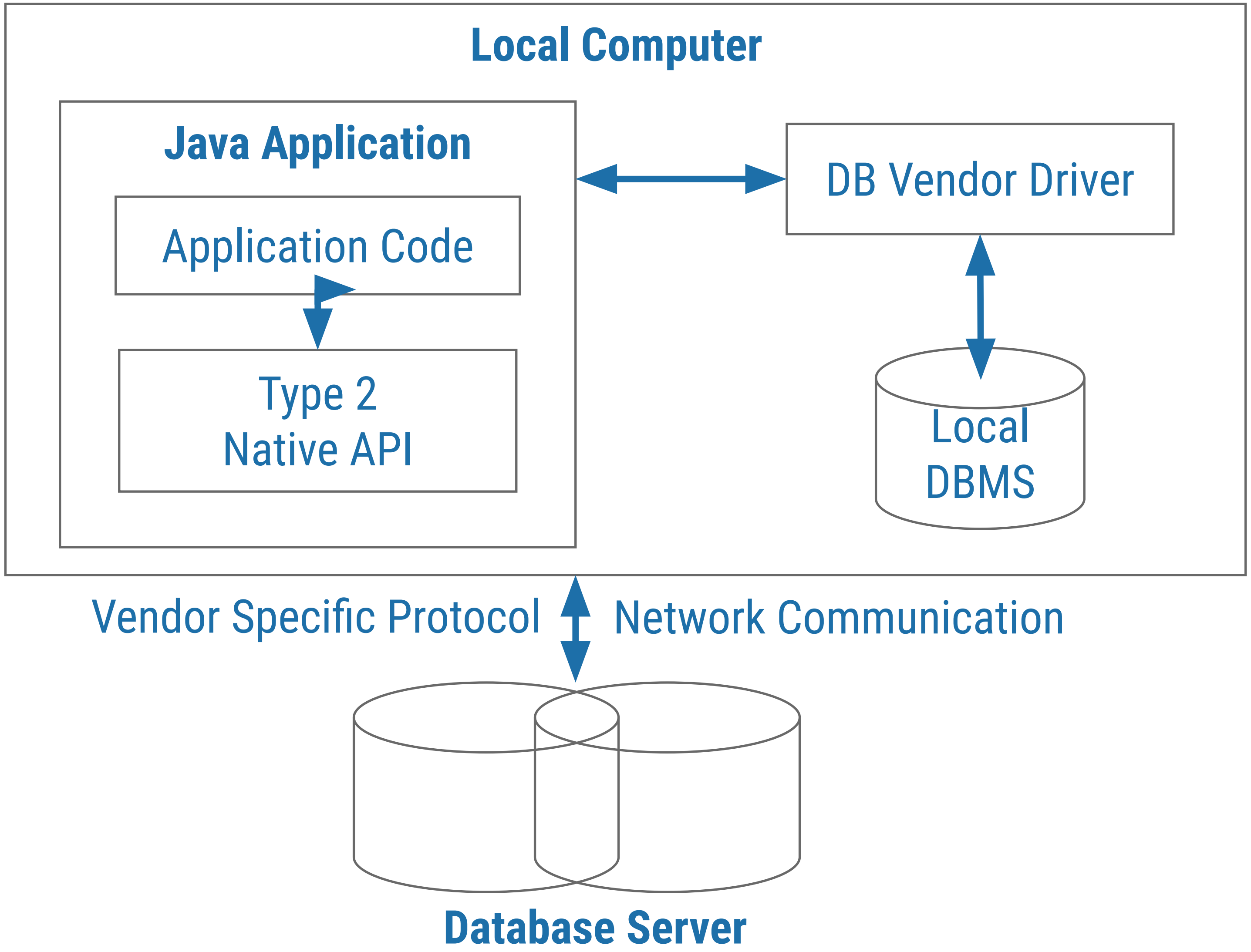
- Due to large number of translations, **execution speed** is decreased
- Dependent on the ODBC driver
- ODBC binary code or ODBC client **library to be installed** in every client machine
- Uses java native interface to make ODBC call

Because of listed disadvantage, type1 driver is not used in production environment. It can only be used, when database doesn’t have any other JDBC driver implementation.

JDBC Driver: Type 2 (Native Code Driver)

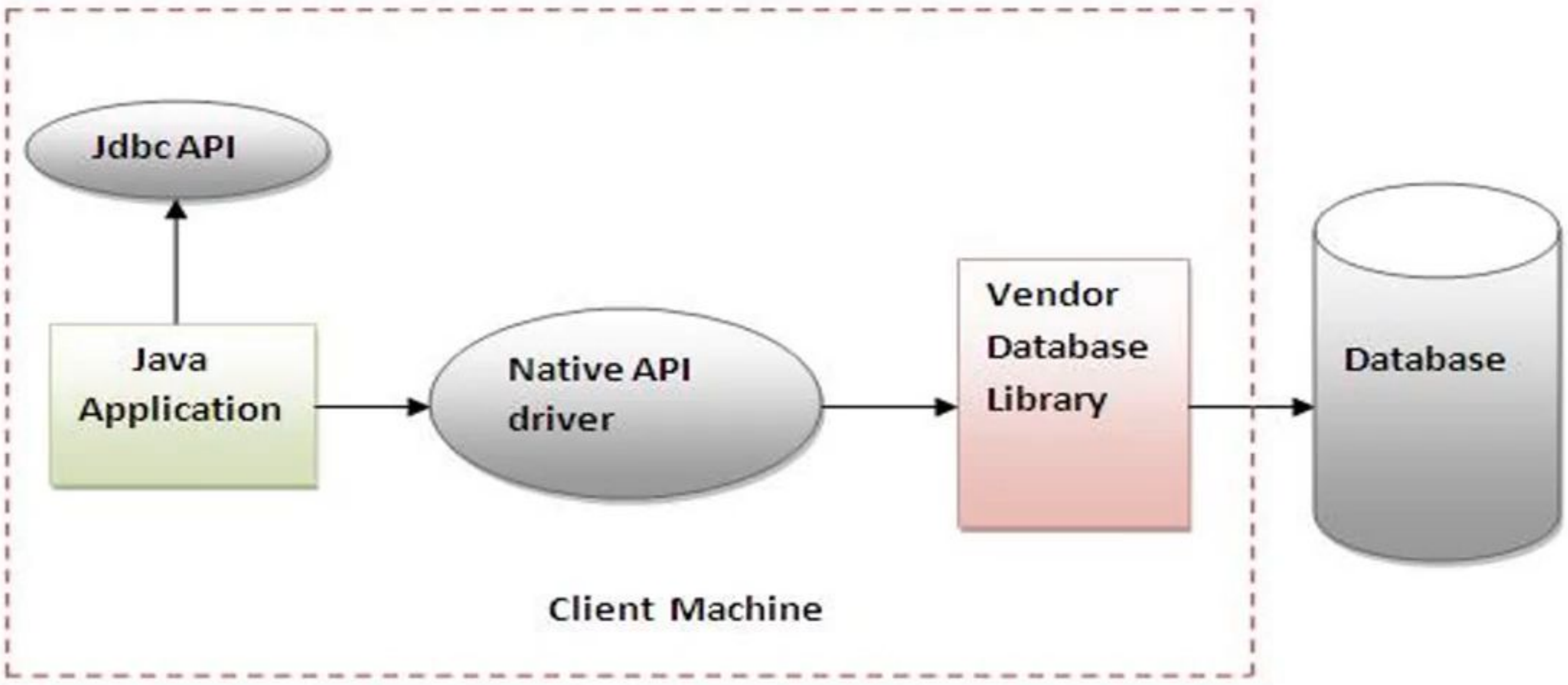
- JDBC API calls are converted into **native API calls**, which are unique to the database.
- These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge.
- Native code Driver are usually written in **C, C++.**
- The vendor-specific driver must be installed on each client machine.
- Type 2 Driver is suitable to use with server side applications.
- E.g. Oracle OCI driver, Weblogic OCI driver, Type2 for Sybase

JDBC Driver: Type 2 (Native Code Driver)



JDBC Driver: Type 2 (Native Code Driver)

Type 2 JDBC Driver: Native-API driver/Partly Java driver(Native Driver)



JDBC Driver: Type 2 (Native Code Driver)

Advantages

- As there is no implementation of JDBC-ODBC bridge, it may be considerably **faster than a Type 1 driver**.

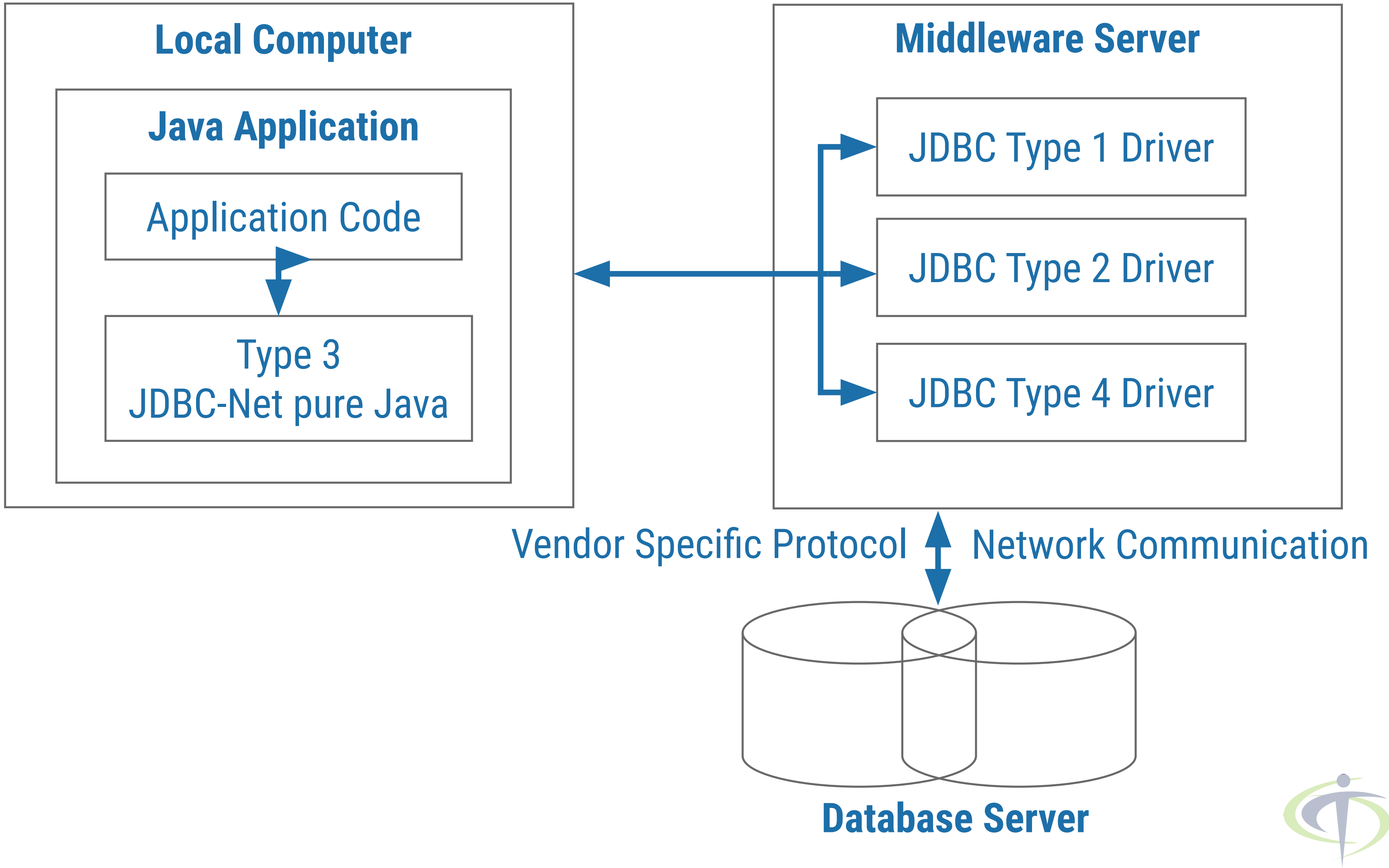
Disadvantages

- The vendor client library needs to be installed on the client machine.
- This driver is **platform dependent**.
- This driver supports all java applications except **applets**.
- It may **increase cost of application**, if it needs to run on different platform (since we may require buying the native libraries for all of the platform).

JDBC Driver: Type 3 (Java Protocol)

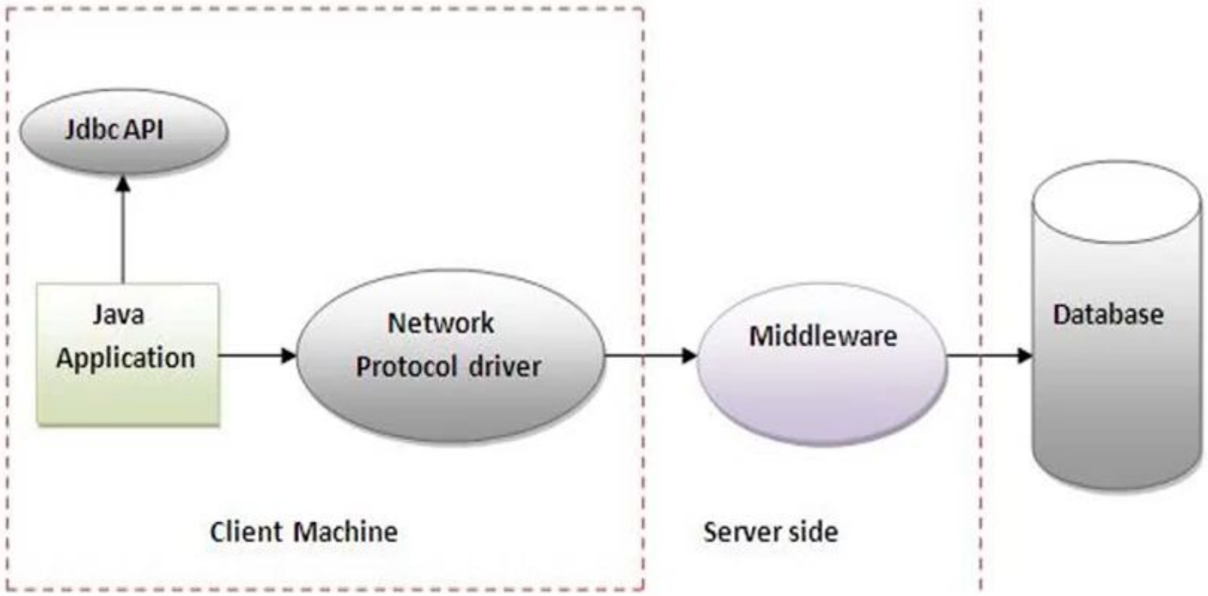
- Pure Java Driver
- Depends on Middleware server
- Can interface to multiple databases – Not vendor specific.
- Follows a three-tier communication approach.
- The JDBC clients use standard network sockets to communicate with a middleware application server.
- The socket information is then translated by the middleware application server into the call format required by the DBMS.
- This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.

JDBC Driver: Type 3 (Java Protocol)



JDBC Driver: Type 3 (Java Protocol)

Type 3 Driver : AllJava/Net-protocol driver or Network Protocol Driver(Middleware Driver)



JDBC Driver: Type 3 (Java Protocol)

Advantages

- Since the communication between client and the middleware server is database independent, there is no need for the database vendor library on the client.
- A single driver can handle any database, provided the middleware supports it.
- We can switch from one database to other without changing the client-side driver class, by just changing configurations of middleware server.
- E.g.: IDS Driver, Weblogic RMI Driver

Disadvantages

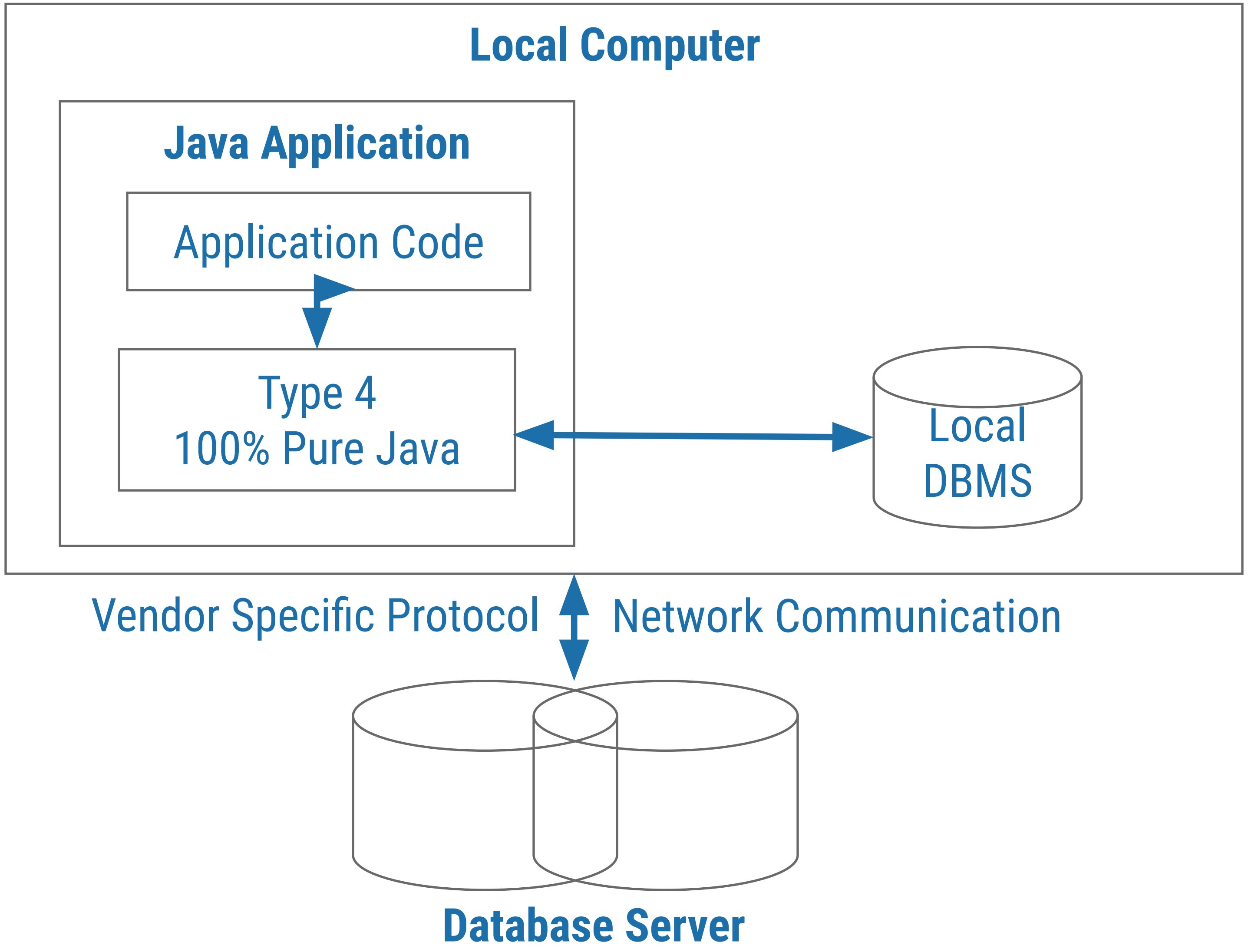
- Compared to Type 2 drivers, Type 3 drivers are slow due to increased number of network calls.
- Requires database-specific coding to be done in the middle tier.
- The middleware layer added may result in additional latency, but is typically overcome by using better middleware services.

JDBC Driver: Type 4 (Database Protocol)

- It is known as the Direct to Database Pure Java Driver
- **Need to download a new driver for each database engine**

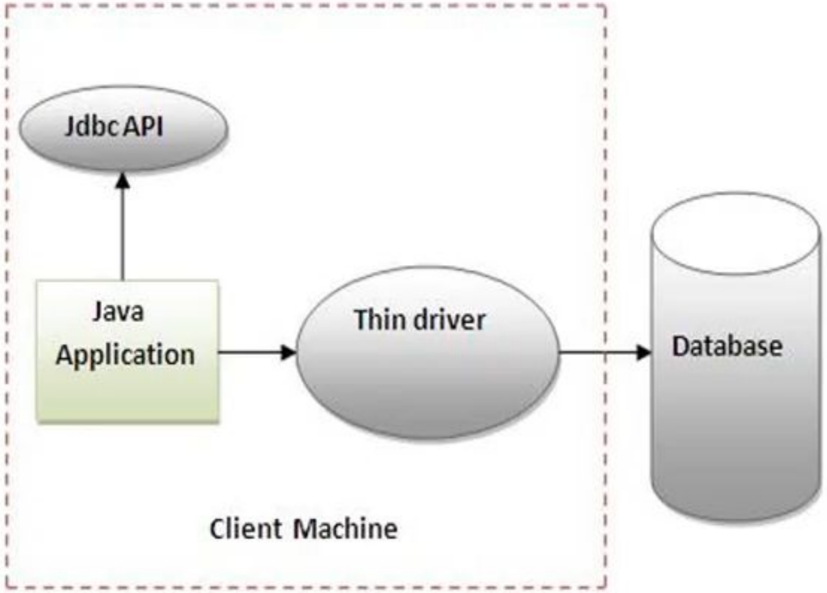
  e.g. Oracle, MySQL
- Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through socket connection.
- This kind of driver is extremely flexible, you don't need to install special software on the client or server.
- Such drivers are implemented by DBMS vendors.

JDBC Driver: Type 4 (Database Protocol)



JDBC Driver: Type 4 (Database Protocol)

Type 4 Driver : All Java/Native-protocol driver or Thin Driver (Pure Java Driver)



JDBC Driver: Type 4 (Database Protocol)

Advantages

- Completely implemented in Java to achieve platform independence.
- No native libraries are required to be installed in client machine.
- These drivers don't translate the requests into an intermediary format (such as ODBC).
- Secure to use since, it uses database server specific protocol.
- The client application connects directly to the database server.
- No translation or middleware layers are used, improving performance.
- The JVM manages all the aspects of the application-to-database connection.

Disadvantage

- This Driver uses database specific protocol and it is DBMS vendor dependent.

JDBC Driver

Thin Driver	You can connect to a database without the client installed on your machine. E.g. Type 4.
Thick Driver	Thick client would need the client installation. E.g. Type 1 and Type 2.

Comparison between JDBC Drivers

Type:	Type 1	Type 2	Type 3	Type 4
Name:	**JDBC-ODBC Bridge**	**Native Code Driver/ JNI**	**Java Protocol/ Middleware**	**Database Protocol**
Vendor Specific:	No	Yes	No	Yes
Pure Java Driver	No	No	Yes	Yes
Working	JDBC-> ODBC call ODBC -> native call	JDBC call -> native specific call	JDBC call -> middleware specific. Middleware -> native call	JDBC call ->DB specific call
Multiple DB	Yes [only ODBC supported DB]	No	Yes [DB Driver should be in middleware]	No

Which Driver should be Used?

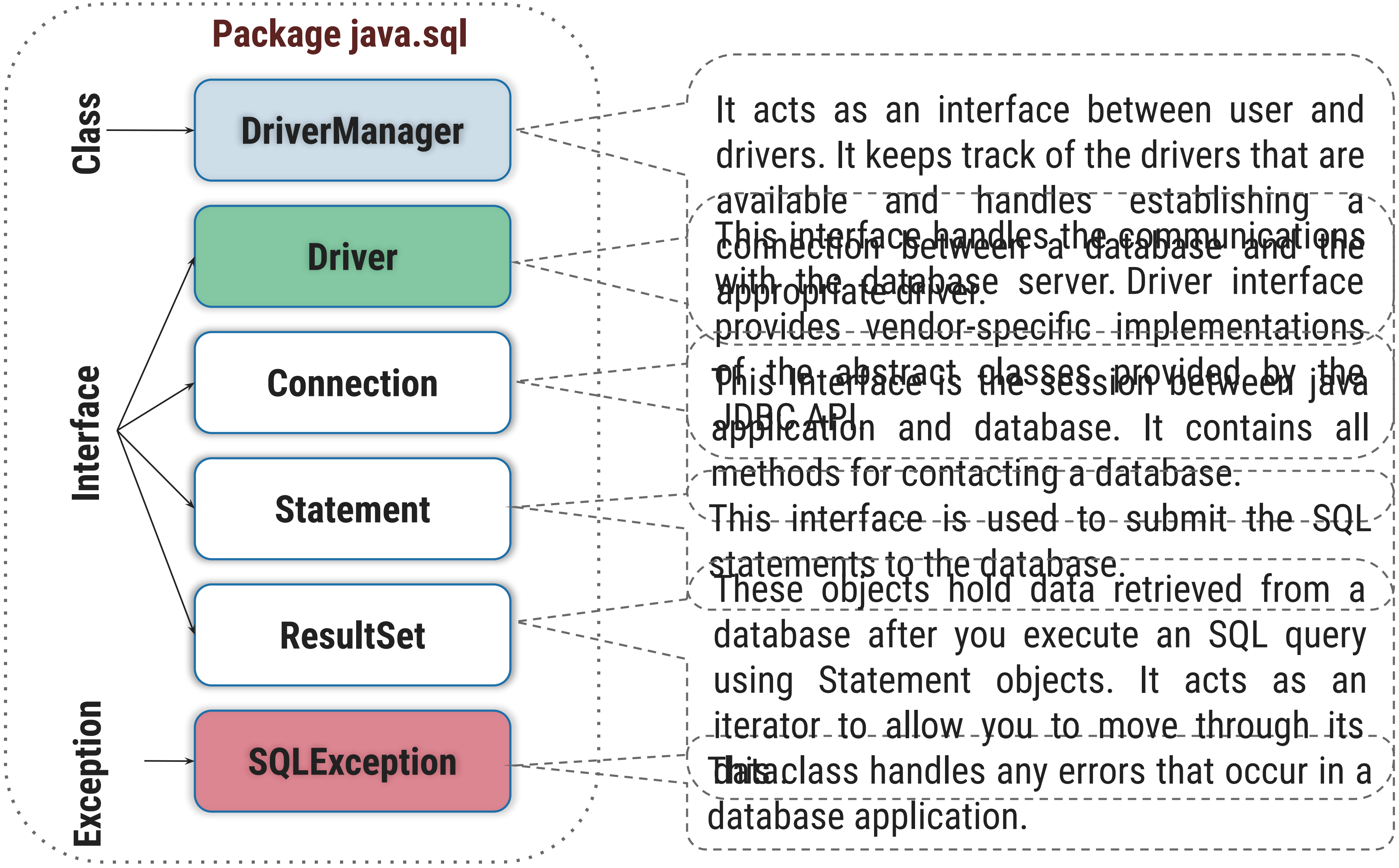
- If you are accessing one type of database such as MySql, Oracle, Sybase or IBM etc., the preferred driver type is 4.
- If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.
- The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

JDBC with different RDBMS

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname:port Number:databaseName
DB2	com.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:port Number /databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:<host>:<port>
SQLite	org.sqlite.JDBC	jdbc:sqlite:C:/sqlite/db/databaseName
SQLServer	com.microsoft.sqlserver.jdbc.SQLServerDriver	jdbc:microsoft:sqlserver://hostname:1433;DatabaseName

JDBC Components

The JDBC API provides the following interfaces and classes



Package java.sql

Class

DriverManager

It acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver.

Interface

Driver

This interface handles the communications with the database server. Driver interface provides vendor-specific implementations of the abstract classes provided by the JDBC API.

Connection

This interface is the session between java application and database. It contains all methods for contacting a database.

Statement

This interface is used to submit the SQL statements to the database.

ResultSet

These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.

Exception

SQLException

This class handles any errors that occur in a database application.

JDBC Package

- Contains core java objects of JDBC API.
- It includes java data objects, that provides basics for connecting to DBMS and interacting with data stored in DBMS.
- This package performs JDBC core operations such as Creating and Executing query.

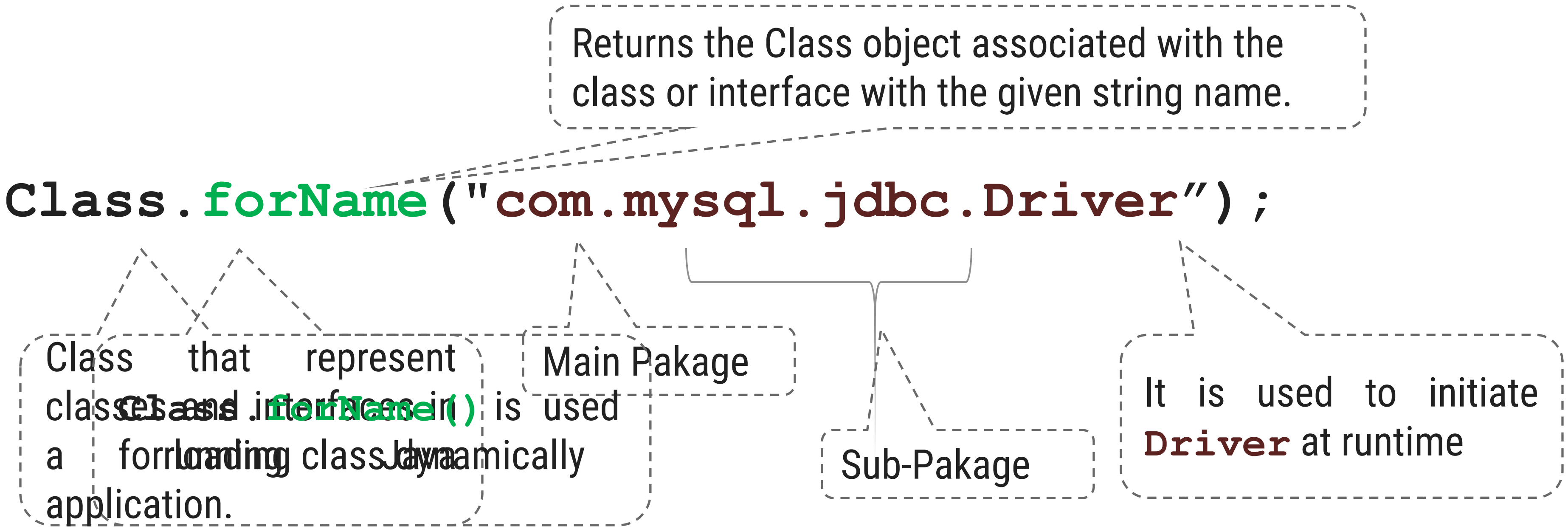
java.sql

JDBC Process

- Step 1: **Loading JDBC Driver**
- Step 2: **Connection to DBMS**
- Step 3: **Creating and executing statement**
- Step 4: **Processing data returned by the DBMS**
- Step 5: **Terminating Connection with DBMS**

Step 1: Loading JDBC Driver

- Create an instance of the driver
- Register driver in the driver manager
- Loading the driver or drivers
  - for example, you want to use driver for mysql, the following code will load it:



Returns the Class object associated with the class or interface with the given string name.

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

Class that represent classes and interfaces in a running Java application.

Class.forName() is used for loading class dynamically

Main Pakage

Sub-Pakage

It is used to initiate Driver at runtime

Step 2: Connection to DBMS

- After you've loaded the driver, you can establish a connection using the **DriverManager** class (java.sql.DriverManager).

Method: DriverManager

`public static Connection getConnection(String url) throws SQLException`	Attempts to establish a connection to the given database URL. The DriverManager attempts to select an appropriate driver from the set of registered JDBC drivers.
`public static Connection getConnection(String url, String user, String password) throws SQLException`	Attempts to establish a connection to the given database URL. **url -** a database url of the form jdbc:*subprotocol*:*subname* **user** - the database user on whose behalf the connection is being made **password** - the user's password

Interface of java.sql package

```
Connection conn= DriverManager.getConnection(URL,USER_NM,PASS);
```

Class of java.sql package

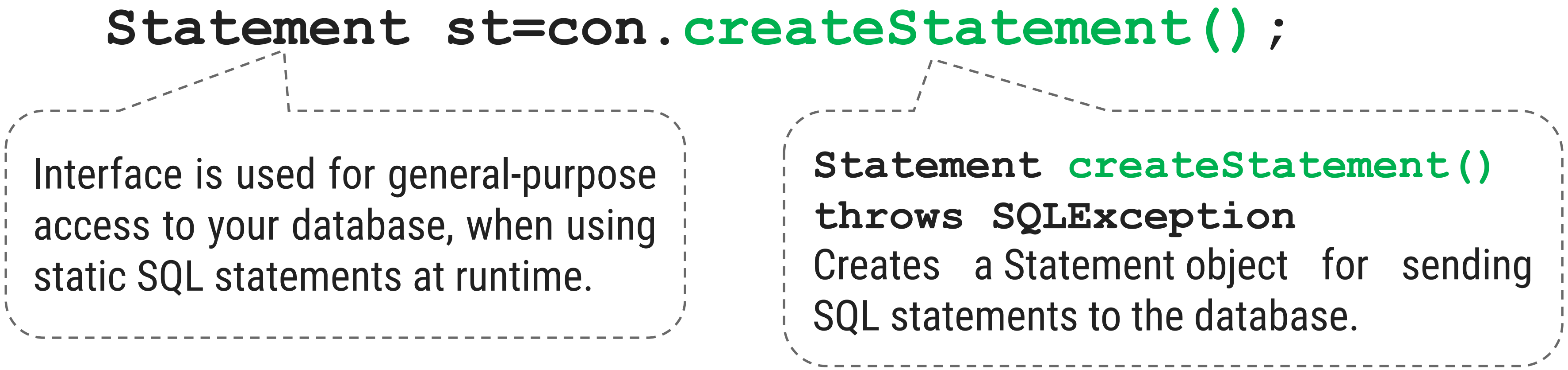
Example:

Database Name

```
Connection conn = DriverManager.getConnection
("jdbc:mysql://localhost:3306/gtu","root", "pwd");
```

Step 3: Creating statement

- Once a connection is obtained, we can interact with the database.
- The JDBC ***Statement*** interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.



```
Statement st=con.createStatement();
```

Interface is used for general-purpose access to your database, when using static SQL statements at runtime.

`Statement createStatement() throws SQLException`

Creates a Statement object for sending SQL statements to the database.

Step 3: Executing Statement

- Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.

ResultSet **executeQuery**(String sql) throws SQLException	Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.
Boolean **execute**(String sql) throws SQLException	Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false.
int **executeUpdate**(String sql) throws SQLException	Returns the number of rows affected by the execution of the SQL statement. for example, an INSERT, UPDATE, or DELETE statement.

Syntax:

```
ResultSet rs=st.executeQuery("query");
```

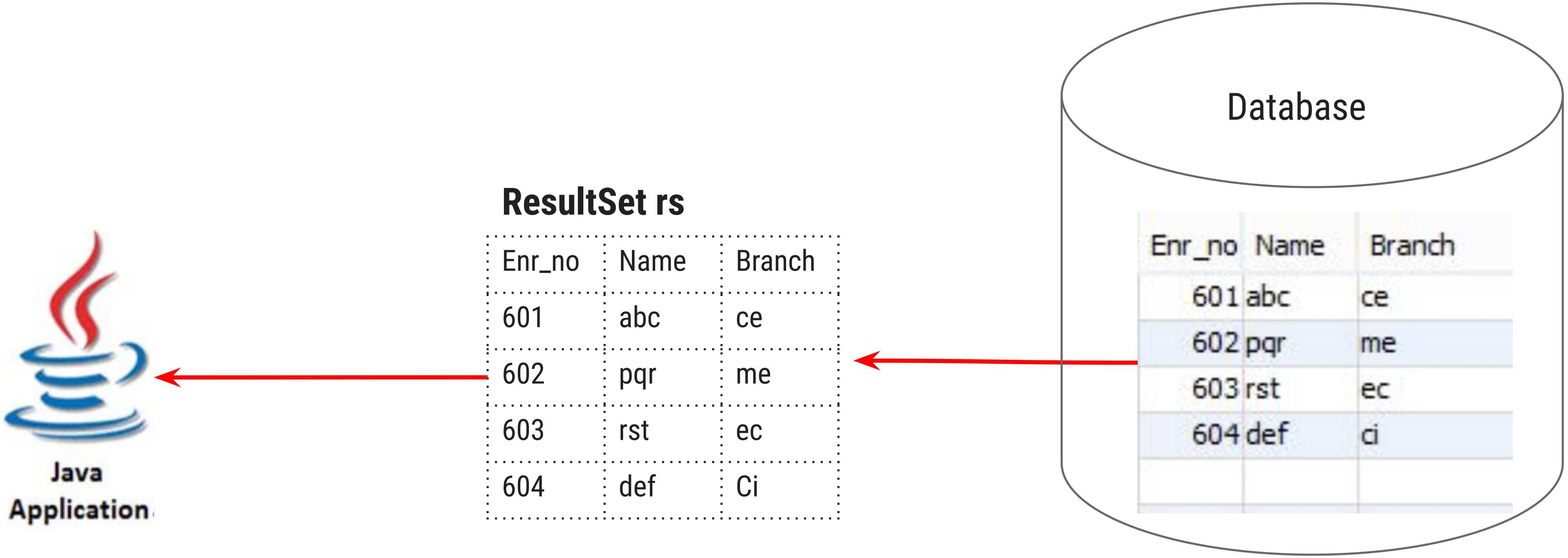
It holds data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.

Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

Step 3: Executing Statement

Example :

```
ResultSet rs = stmt.executeQuery("SELECT * from diet");
```



Database

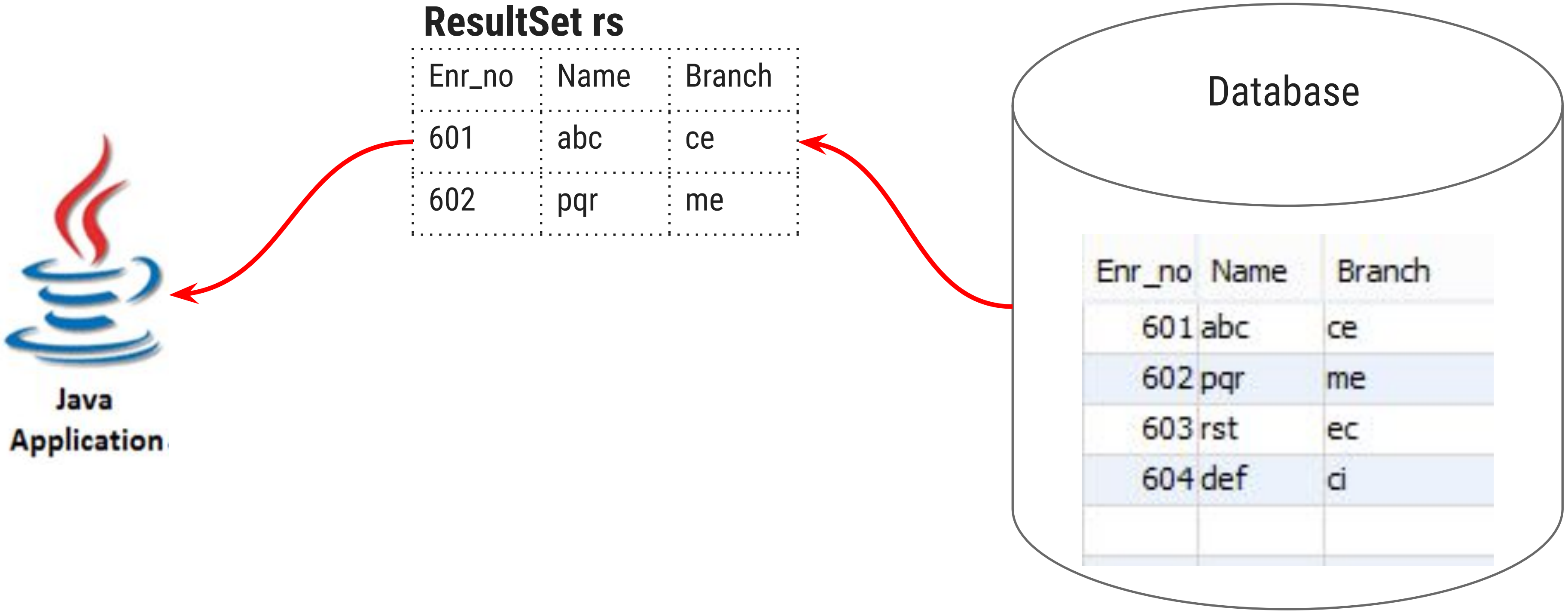
Enr_no	Name	Branch
601	abc	ce
602	pqr	me
603	rst	ec
604	def	ci

ResultSet rs

Enr_no	Name	Branch
601	abc	ce
602	pqr	me
603	rst	ec
604	def	Ci

Step 3: Executing Statement

```
ResultSet rs = stmt.executeQuery("SELECT * FROM diet WHERE
      Enr_no='601'OR Enr_no='602'");
```



ResultSet rs

Enr_no	Name	Branch
601	abc	ce
602	pqr	me

Database

Enr_no	Name	Branch
601	abc	ce
602	pqr	me
603	rst	ec
604	def	ci

Step 4:Processing data returned by the DBMS

- Method: Resultset

`boolean next()`
`Throws SQLException`
	Moves the cursor forward one row from its current position.

`String getString (int col_Index)`
`throws SQLException`
	Retrieves the value of the designated column in the current row of this ResultSet object as a String

`String getString`
`(String col_Label)`
`throws SQLException`
	Retrieves the value of the designated column in the current row of this ResultSet object as a String in the Java programming language.

`int getInt(int columnIndex) throws SQLException`	Returns the int in the current row in the specified column index.

`int getInt(String columnLabel)`
`throws SQLException`
	Retrieves the value of the designated column in the current row

Processing data returned by the DBMS

- Example

```
  while(rs.next())
 {
     System.out.println(rs.getString(1));
     System.out.println(rs.getInt(“emp_id”));
 }
```

Returns the value of specified Column number

Returns the value of specified Column name

- The connection of DBMS is terminated by using close() method.

Example

```
        rs.close();
        st.close();
        con.close();
```

Releases this ResultSet object's database and JDBC resources immediately

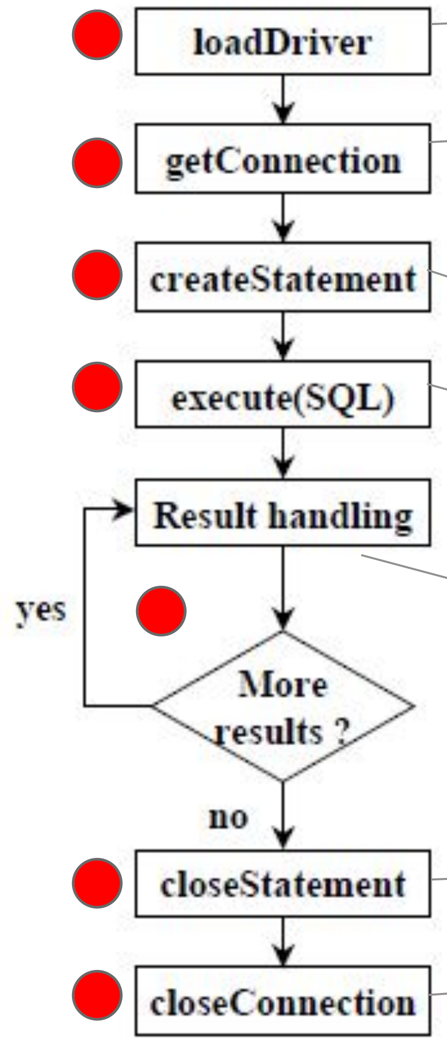
Releases this Statement object's database and JDBC resources immediately

Releases this Connection object's database and JDBC resources immediately

JDBC with different RDBMS

RDBMS	JDBC driver name	URL format
MySQL	**com.mysql.jdbc.Driver**	**jdbc:mysql://hostname/databaseName**
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname:port Number:databaseName
DB2	com.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:port Number /databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:<host>:<port>
SQLite	org.sqlite.JDBC	jdbc:sqlite:C:/sqlite/db/databaseName
SQLServer	com.microsoft.sqlserver.jdbc.SQLServerDriver	jdbc:microsoft:sqlserver: //hostname:1433;DatabaseName

JDBC Program

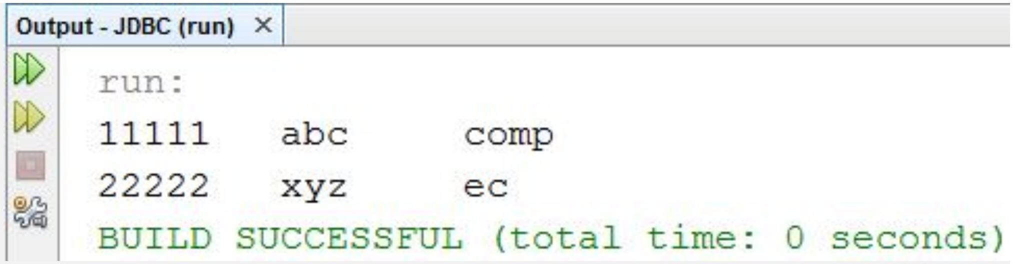


```
Class.forName("com.mysql.jdbc.Driver");
Connection conn=
DriverManager.getConnection("jdbc:mysql://localhost:3306/gtu", “root”, “pwd”);
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT *
from diet");
while(rs.next())
   System.out.print(rs.getString(1));
stmt.close();
conn.close();
```

First JDBC Program

ConnDemo.java

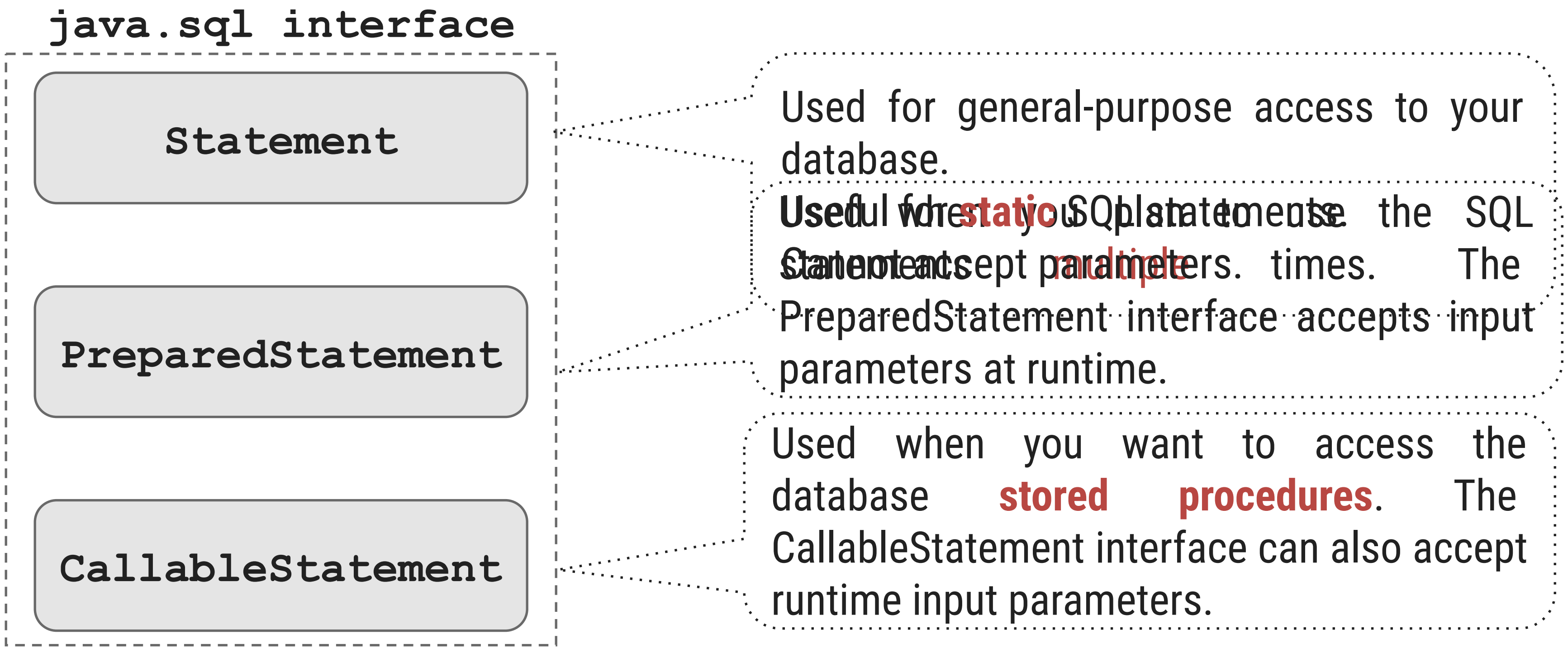
```java
import java.sql.*;
public class ConnDemo {
public static void main(String[] args) {
  try {
       Class.forName("com.mysql.jdbc.Driver");
       Connection conn= DriverManager.getConnection
                ("jdbc:mysql://localhost:3306/gtu","root",”pwd");
       Statement stmt = conn.createStatement();
       ResultSet rs = stmt.executeQuery("SELECT * from diet");
       while(rs.next()){
               System.out.print(rs.getInt(1)+"\t");
               System.out.print(rs.getString(“Name”)+"\t");
               System.out.println(rs.getString(3));
        }//while
        stmt.close();
        conn.close();
    }catch(Exception e){System.out.println(e.toString());
   }//PSVM  }//class
```



```
run:
11111   abc     comp
22222   xyz     ec
BUILD SUCCESSFUL (total time: 0 seconds)
```

Types of Statement

- The JDBC **Statement**, **PreparedStatement** **and** **CallableStatement** interface define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.



java.sql interface

Statement

PreparedStatement

CallableStatement

Used for general-purpose access to your database.

Useful for **static** SQL statements. Cannot accept parameters.

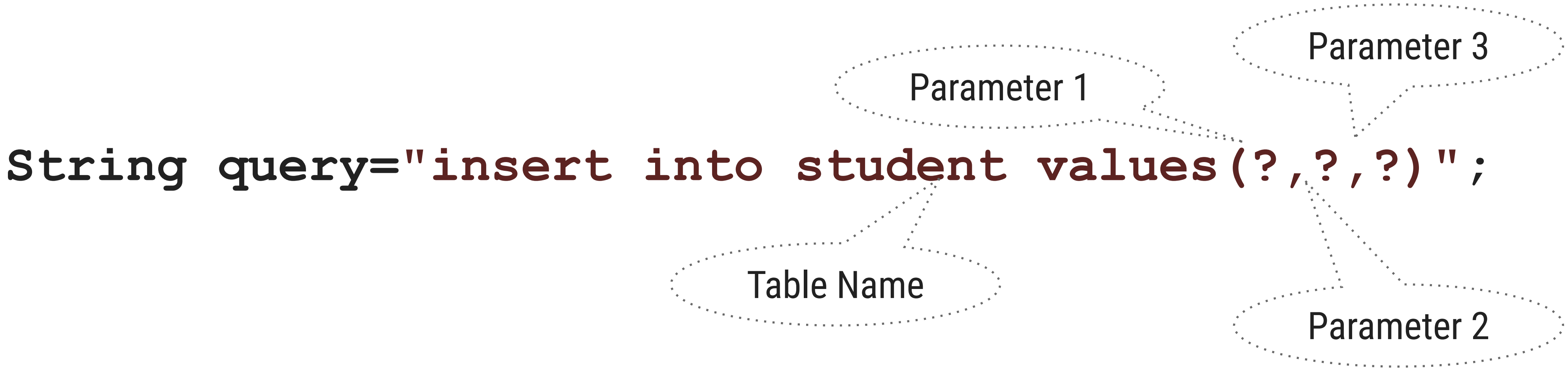
Used when you plan to use the SQL statements **multiple** times. The PreparedStatement interface accepts input parameters at runtime.

Used when you want to access the database **stored procedures**. The CallableStatement interface can also accept runtime input parameters.

Prepared Statement

- The **PreparedStatement** interface extends the Statement interface.
- It represents a **precompiled** SQL statement.
- A SQL statement is precompiled and stored in a Prepared Statement object.
- This object can then be used to efficiently execute this statement **multiple times**.

Example



```
String query="insert into student values(?,?,?)";
```

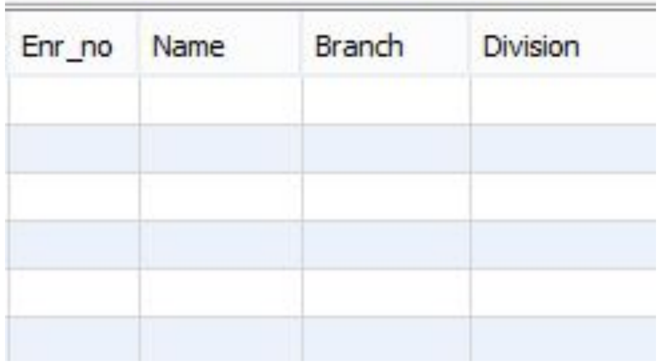
Methods of PreparedStatement interface

public void **setInt**(int paramIndex, int value)	Sets the integer value to the given parameter index.
public void **setString**(int paramIndex, String value)	Sets the String value to the given parameter index.
public void **setFloat**(int paramIndex, float value)	Sets the float value to the given parameter index.
public void **setDouble**(int paramIndex, double value)	Sets the double value to the given parameter index.
public int **executeUpdate**()	Executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSet **executeQuery**()	Executes the select query. It returns an instance of ResultSet.

Prepared Statement

- Now to create table in mysql.

```
create table gtu.DietStudent
(
    Enr_no VARCHAR(10) not null
    Name VARCHAR(20),
    Branch VARCHAR(10),
    Division VARCHAR(10),
    primary key (Enr_no)
)
```



Enr_no	Name	Branch	Division

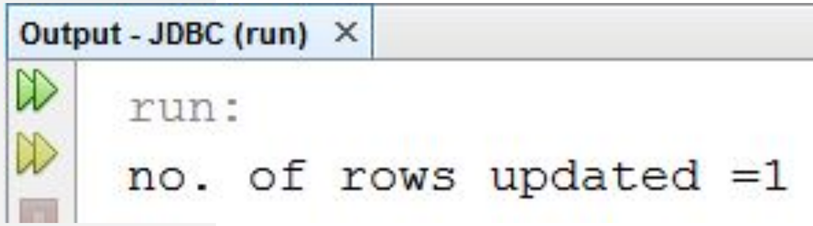
Example of PreparedStatement that inserts the record

PreparedInsert.java

```java
import java.sql.*;
public class PreparedInsert {
public static void main(String[] args) {
    try {
        Class.forName("com.mysql.jdbc.Driver");
        Connection conn= DriverManager.getConnection
            ("jdbc:mysql://localhost:3306/gtu",
            "root",“pwd");
        String query="insert into dietstudent values(?,?,?,?)";

        PreparedStatement ps=conn.prepareStatement(query);
        ps.setString(1, "14092"); //Enr_no
        ps.setString(2, "abc_comp"); //Name
        ps.setString(3, "computer"); //Branch
        ps.setString(4, "cx"); //Division
        int i=ps.executeUpdate();
        System.out.println("no. of rows updated ="+i);

        ps.close();
        conn.close();
    }catch(Exception e){System.out.println(e.toString());} }//PSVM
}//class
```



```
run:
no. of rows updated =1
```

Why to use PreparedStatement?

- The performance of the application will be faster, if you use PreparedStatement interface because query is compiled only once.
- This is because creating a PreparedStatement object by explicitly giving the SQL statement causes the statement to be precompiled within the database immediately.
- Thus, when the PreparedStatement is later executed, the DBMS does not have to recompile the SQL statement.
- Late binding and compilation is done by DBMS.
- Provides the programmatic approach to set the values.

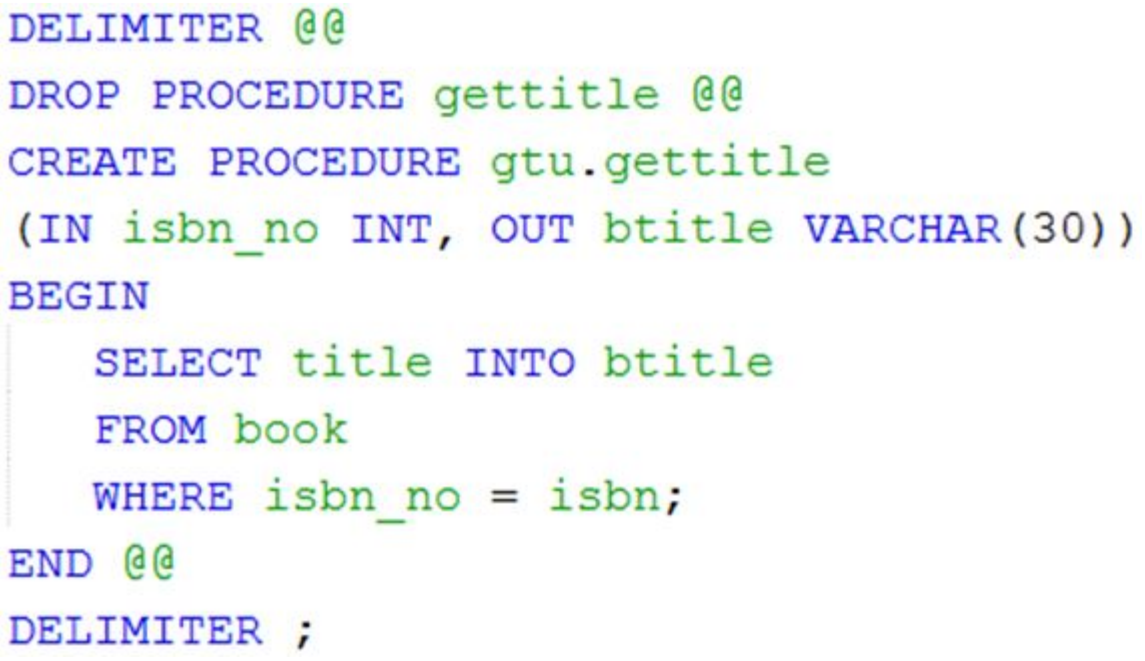
Callable Statement

- CallableStatement interface is used to call the **stored procedures**.
- We can have business logic on the database by the use of stored procedures that will make the performance better as they are **precompiled**.
- Three types of parameters exist: **IN, OUT, and INOUT.** The PreparedStatement object only uses the **IN** parameter. The CallableStatement object can use all the three.

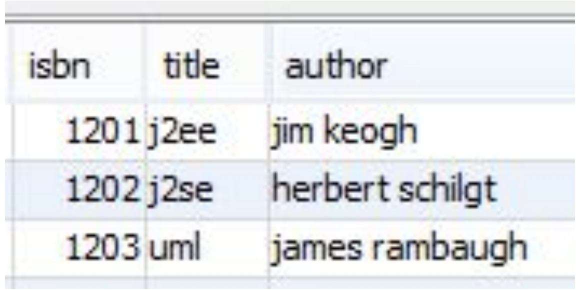
Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from the OUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

Callable Statement

- Create mysql procedure to get book title for given ISBN number.



```
DELIMITER @@
DROP PROCEDURE gettitle @@
CREATE PROCEDURE gtu.gettitle
(IN isbn_no INT, OUT btitle VARCHAR(30))
BEGIN
    SELECT title INTO btitle
    FROM book
    WHERE isbn_no = isbn;
END @@
DELIMITER ;
```



isbn	title	author
1201	j2ee	jim keogh
1202	j2se	herbert schilgt
1203	uml	james rambaugh

Example CallableStatement

CallableDemo.java

```java
import java.sql.*;
public class CallableDemo {
public static void main(String[] args) {
    try {
        Class.forName("com.mysql.jdbc.Driver");
        Connection conn= DriverManager.getConnection
           ("jdbc:mysql://localhost:3306/gtu",
       "root",“pwd");

        CallableStatement cs=conn.prepareCall("{call gettitle(?,?)}");
        cs.setInt(1,1201);
        cs.registerOutParameter(2,Types.VARCHAR);
        cs.execute();
        System.out.println(cs.getString(2));

        cs.close();
        conn.close();
    }catch(Exception e){System.out.println(e.toString());}
   }//PSVM
}//class
```

Procedure Name

Method: ResultSet

1.	**Navigational methods**	Used to move the cursor around.
2.	**Get methods**	Used to view the data in the columns of the current row being pointed by the cursor.
3.	**Update methods**	Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

ResultSet: Navigational methods

boolean **first()** throws SQLException	Moves the cursor to the first row.
boolean **last()** throws SQLException	Moves the cursor to the last row.
boolean **next()** throws SQL Exception	Moves the cursor to the next row. This method returns false if there are no more rows in the result set.
boolean **previous()** throws SQLException	Moves the cursor to the previous row. This method returns false if the previous row is off the result set.
boolean **absolute(int row)** throws SQLException	Moves the cursor to the specified row.
boolean **relative(int row)** throws SQLException	Moves the cursor the given number of rows forward or backward, from where it is currently pointing.
int **getRow()** throws SQLException	Returns the row number that the cursor is pointing to.

ResultSet: Get methods

int **getInt(String columnName)** throws SQLException	Returns the int in the current row in the column named columnName.
int **getInt(int columnIndex)** throws SQLException	Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.
String **getString(String columnLabel)** throws SQLException	Retrieves the value of the designated column in the current row of this ResultSet object as a String in the Java programming language.
String **getString(int columnIndex)** throws SQLException	Retrieves the value of the designated column in the current row of this ResultSet object as a String in the Java programming language.

ResultSet: Update methods

void **updateString(int col_Index, String s)** throws SQLException	Changes the String in the specified column to the value of s.
void **updateInt(int col_Index, int x)** throws SQLException	Updates the designated column with an int value.
void **updateFloat(int col_Index, float x)** throws SQLException	Updates the designated column with a float value.
void **updateDouble(int col_Index,double x)** throws SQLException	Updates the designated column with a double value.

Types of ResultSet

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

Concurrency of ResultSet

Concurrency	Description
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set.
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.

How to set Type and Concurrency ?

- createStatement(int RSType, int RSConcurrency);

- prepareStatement(String SQL, int RSType, int RSConcurrency);

- prepareCall(String sql, int RSType, int RSConcurrency);

ResultSetMetaData Interface

- The metadata means data about data.
- If you have to get metadata of a table like
  - total number of column
  - column name
  - column type etc.
- ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

Method: ResultSetMetaData

int **getColumnCount()** throws SQLException	it returns the total number of columns in the ResultSet object.
String **getColumnName(int index)** throws SQLException	it returns the column name of the specified column index.
String **getColumnTypeName(int index**) throws SQLException	it returns the column type name for the specified index.

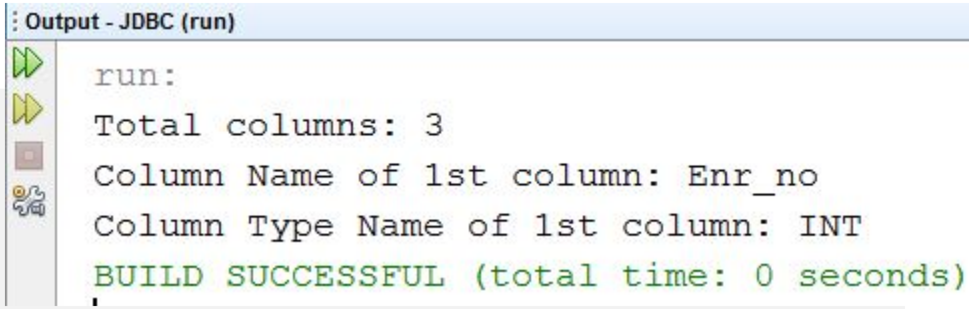
ResultSetMetaData

MetadataDemo.java

```java
import java.sql.*;
public class MetadataDemo {
public static void main(String[] args) {
    try {Class.forName("com.mysql.jdbc.Driver");
        Connection conn= DriverManager.getConnection
                   ("jdbc:mysql://localhost:3306/gtu", "root",“pwd");
        Statement stmt = conn.createStatement
(ResultSet.TYPE_FORWARD_ONLY,ResultSet.CONCUR_READ_ONLY);
        ResultSet rs = stmt.executeQuery("SELECT * from gtu");

        ResultSetMetaData rsmd=rs.getMetaData();
        System.out.println("Total columns: "+rsmd.getColumnCount());
        System.out.println("Column Name of 1st column: "+rsmd.getColumnName(1));
        System.out.println("Column Type Name of 1st column:“+rsmd.getColumnTypeName(1));

        stmt.close();
        conn.close();
    }catch(Exception e){System.out.println(e.toString());}
   }//PSVM
}//class
```



```
run:
Total columns: 3
Column Name of 1st column: Enr_no
Column Type Name of 1st column: INT
BUILD SUCCESSFUL (total time: 0 seconds)
```

DatabaseMetadata

- DatabaseMetaData interface provides methods to get meta data of a database such as
  - database product name,
  - database product version,
  - driver name,
  - name of total number of tables etc.

DabaseInfo.java

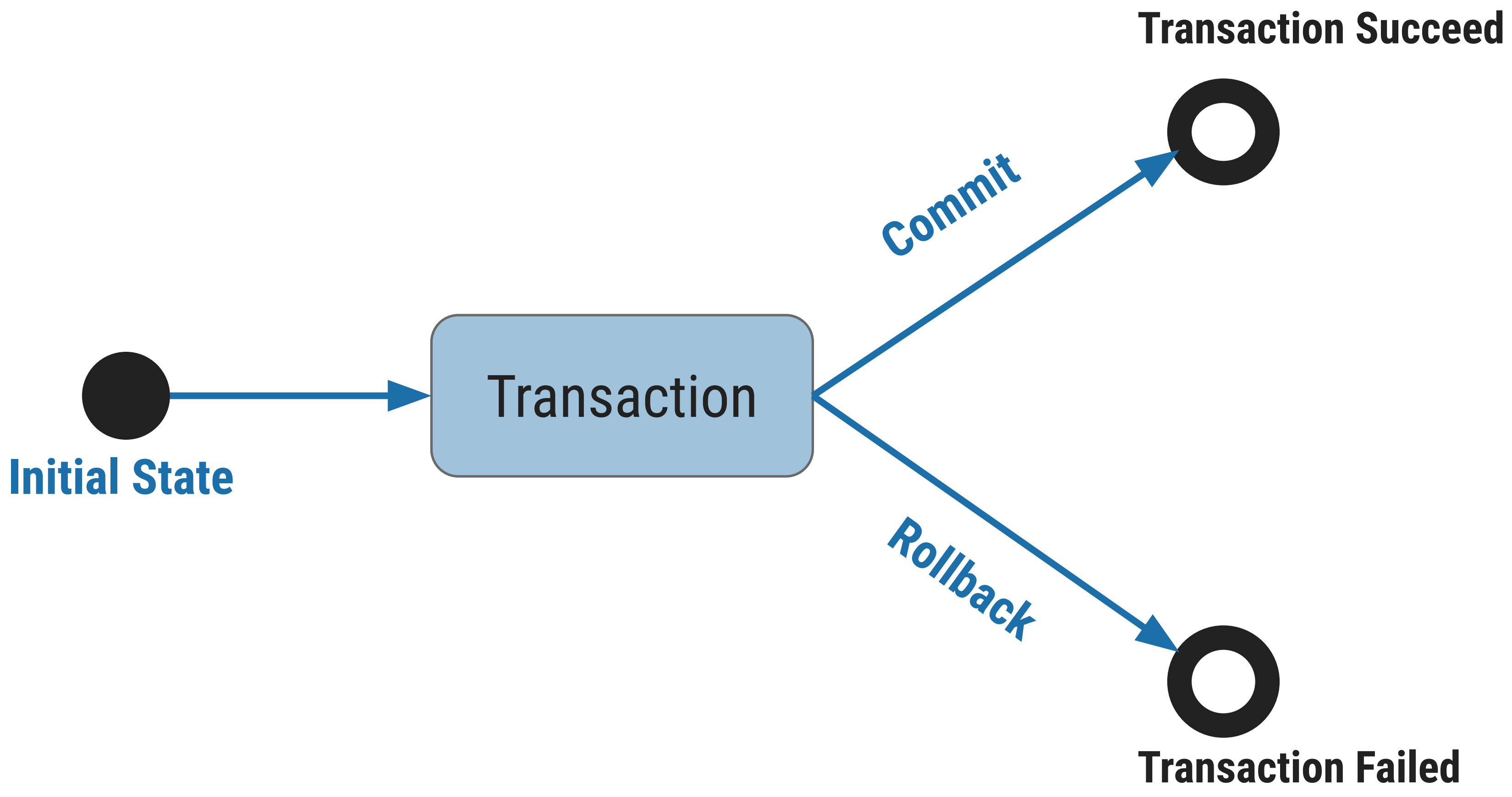
```java
Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/college",
                          "root","");
DatabaseMetaData dbmd=con.getMetaData();
System.out.println("getTransactionIsolation="+con.getTransactionIsolation());
System.out.println("getDatabaseProductName:" +dbmd.getDatabaseProductName());
System.out.println("getDatabaseProductVersion():"+dbmd.getDatabaseProductVersion());
System.out.println("getDriverName():"+dbmd.getDriverName());
System.out.println("getDriverVersion():"+dbmd.getDriverVersion());
System.out.println("getURL():"+dbmd.getURL());
System.out.println("getUserName():"+dbmd.getUserName());
```

Executing SQL updates

UpdateDemo.java

```java
import java.sql.*;
class UpdateDemo{
public static void main(String args[])
{
    try
    {
       Class.forName("com.mysql.jdbc.Driver");
           Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/GTU",
                                           "root","root");
           Statement stmt=con.createStatement();
           String query="update diet set Name='abc601' where Enr_no=601";
           int i=stmt.executeUpdate(query);
           System.out.println("total no. of rows updated="+i);
           stmt.close();
           con.close();
       }
    catch(Exception e)
    {
         System.out.println(e);
    }
}  }
```

Transaction Management



Transaction Management

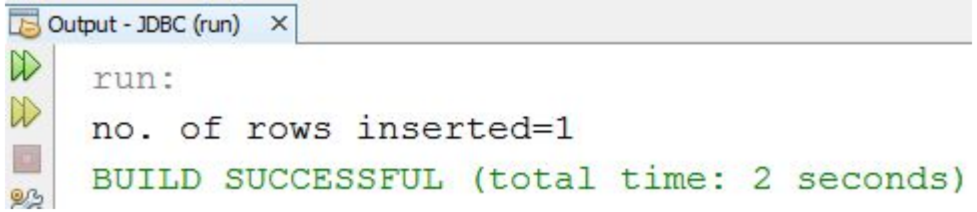
- In JDBC, **Connection interface** provides methods to manage transaction.

void **setAutoCommit**(boolean status)	It is true **by default,** means each transaction is committed bydefault.
void **commit**()	commits the transaction.
void **rollback**()	cancels the transaction.

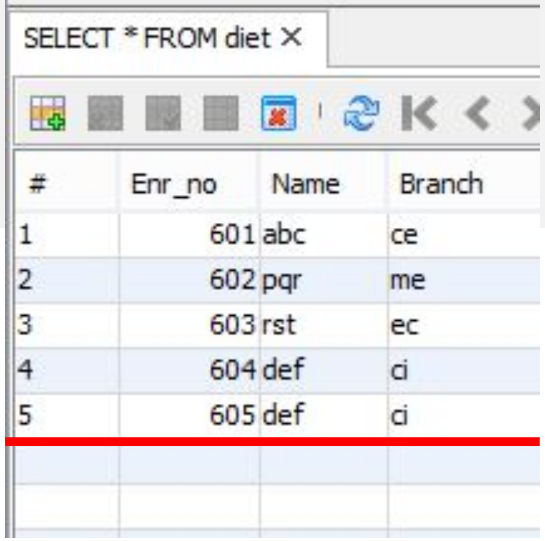
Transaction Management:commit

CommitDemo.java

```java
import java.sql.*;
class CommitDemo{
public static void main(String args[]){
try{
    Class.forName("com.mysql.jdbc.Driver");
    Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/GTU",
                                "root","root");
    con.setAutoCommit(false);//bydefault it is true
    Statement stmt=con.createStatement();
    int i=stmt.executeUpdate("insert into diet values(605,'def','ci')");
    System.out.println("no. of rows inserted="+i);
    con.commit();//commit transaction
    con.close();
 }catch(Exception e){ System.out.println(e);}
}}
```



```
run:
no. of rows inserted=1
BUILD SUCCESSFUL (total time: 2 seconds)
```

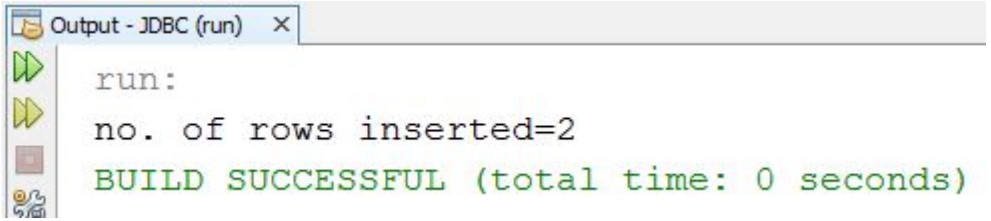


#	Enr_no	Name	Branch
1	601	abc	ce
2	602	pqr	me
3	603	rst	ec
4	604	def	ci
5	605	def	ci

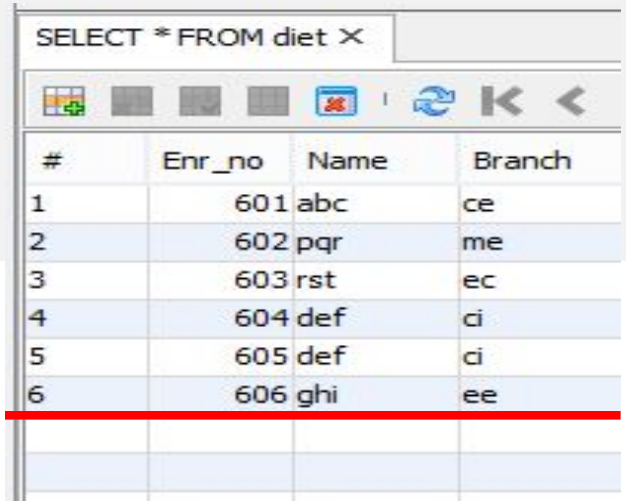
Transaction Management:rollback

RollbackDemo.java

```java
import java.sql.*;
class RollbackDemo{
public static void main(String args[]){
try{
  Class.forName("com.mysql.jdbc.Driver");
  Connection con=DriverManager.getConnection(
             "jdbc:mysql://localhost:3306/GTU","root","root");
  con.setAutoCommit(false);//bydeafault it is true
  Statement stmt=con.createStatement();
  int i=stmt.executeUpdate("insert into diet values(606,'ghi','ee')");
  con.commit(); //Commit Transaction
  i+=stmt.executeUpdate("insert into diet values(607,'mno','ch')");
  System.out.println("no. of rows inserted="+i);
  con.rollback(); //Rollback Transaction
  con.close();
 }catch(Exception e){ System.out.println(e);}
}}
```



```
run:
no. of rows inserted=2
BUILD SUCCESSFUL (total time: 0 seconds)
```



#	Enr_no	Name	Branch
1	601	abc	ce
2	602	pqr	me
3	603	rst	ec
4	604	def	ci
5	605	def	ci
6	606	ghi	ee

Batch Processing in JDBC

- Instead of executing a single query, we can execute a batch (group) of queries.
- It makes the performance fast.
- The java.sql.Statement and java.sql.PreparedStatement interfaces provide methods for batch processing.

Methods of Statement interface

void **addBatch**(String query)	It adds query into batch.
int[] **executeBatch**()	It executes the batch of queries.

Batch Processing in JDBC

Batch.java

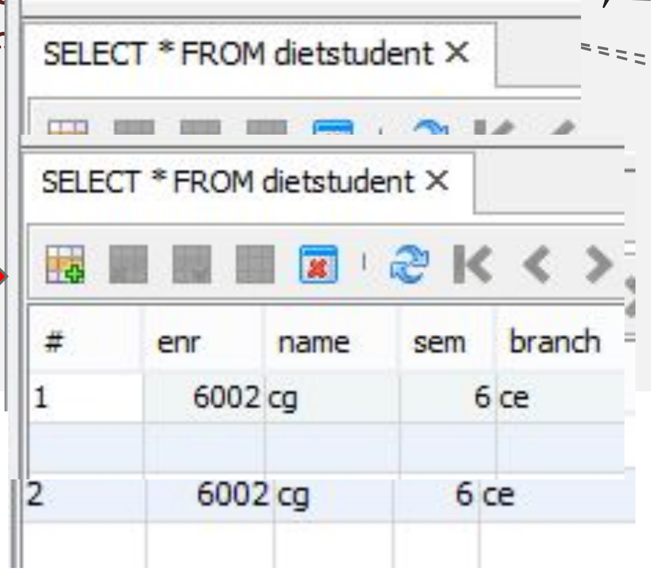
```
Class.forName("com.mysql.jdbc.Driver");
Connection con=DriverManager.getConnection(
                    "jdbc:mysql://localhost:3306/GTU","root","root");
con.setAutoCommit(false);
Statement stmt=con.createStatement();
String query1,query2,query3,query4,query5;
query1="create table DietStudent(enr INT PRIMARY KEY, name VARCHAR(20),sem INT,branch VARCHAR(10))";
query2="insert into DietStudent values(6001,'java',6,'ce')";
query3="insert into DietStudent values(6002,'php',6,'ce')";
query4="update DietStudent set name='cg' where enr=6002";
query5="delete from DietStuden
stmt.addBatch(query1);
stmt.addBatch(query2);
stmt.addBatch(query3);
stmt.addBatch(query4);
stmt.addBatch(query5);
int[] i=stmt.executeBatch();
con.commit();
```

Create table

Insert record

Update record

Delete record



SELECT * FROM dietstudent ✕

SELECT * FROM dietstudent ✕

#	enr	name	sem	branch
1	6002	cg	6	ce
2	6002	cg	6	ce

Transaction Isolation Level

- JDBC isolation level represents that, how a database maintains its interiority against the problem such as
  - dirty reads
  - non-repeatable reads
  - phantom reads

that occurs during concurrent transactions.

Transaction Isolation Level

What is Dirty read?

- Dirty read occurs when one transaction is changing the record, and the other transaction can read this record before the first transaction has been committed or rolled back.
- This is known as a dirty read scenario because there is always a possibility that the first transaction may rollback the change, resulting in the second transaction having read an invalid data.

What is Non-Repeatable Read?

- Non Repeatable Reads happen when in a same transaction same query yields to a different result.
- This occurs when one transaction repeatedly retrieves the data, while a difference transactions alters the underlying data.
- This causes the different or non-repeatable results to be read by the first transaction.

Transaction Isolation Level

- **What is Phantom read?**
  - At the time of execution of a transaction, if two queries that are identical and executed, and the no. of rows returned are different from other.
  - If you execute a query at time T1 and re-execute it at time T2, additional rows may have been added/deleted to/from the database, which may affect your results.
  - It is stated that a **phantom read** occurred.

Phantom reads vs Non-repeatable reads

Phantom Reads

T	Transaction A	Transaction B
T1	Read n=5	
T2		Read n=5
T3	Delete n	
T4		Read n

Variable Undefined

Non-Repeatable Reads

T	Transaction A	Transaction B
T1	Read n=5	
T2		Read n=5
T3	Update=8	
T4		Read n=8

Same query had retrieved two different value

Transaction Isolation Level

Int Val.	Isolation Level	Description
1	TRANSACTION_READ_UNCOMMITTED	It allows non-repeatable reads, dirty reads and phantom reads to occur
2	TRANSACTION_READ_COMMITTED	It ensures only those data can be read which is committed. Prevents dirty reads.
4	TRANSACTION_REPEATABLE_READ	It is closer to serializable, but phantom reads are also possible. Prevents dirty and non-repeatable reads.
8	TRANSACTION_SERIALIZABLE	In this level of isolation dirty reads, non-repeatable reads, and phantom reads are prevented.

One can get/set the current isolation level by using methods of Connection interface:

1. **getTransactionIsolation()**
2. **setTransactionIsolation(int isolationlevelconstant)**

Transaction Isolation Level:program

IsolationDemo.java

```java
public class IsolationDemo {

    public static void main(String[] args) throws ClassNotFoundException, SQLException
    {
        Class.forName("com.mysql.jdbc.Driver");
        Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/ce17",
                                 "root", "diet");
        System.out.println("getTransactionIsolation=" + con.getTransactionIsolation());
        con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
        System.out.println("NEW getTransactionIsolation=" +
       con.getTransactionIsolation());

    }
}
```

SQL Exception

java.sql.SQLException	It is a core JDBC exception class that provides information about database access errors and other errors. Most of the JDBC methods throw SQLException.
java.sql. BatchUpdateException	It provides the update counts for all commands that were executed successfully during the batch update.
java.sql.DataTruncation	reports a DataTruncation warning (on reads) or throws a DataTruncation exception (on writes) when JDBC unexpectedly truncates a data value.
java.sql.SQLWarning	provides information about database access warnings.