JDBC

Unit II

What is JDBC?

- JDBC is Java application programming interface that allows the Java programmers to access database management system from Java code.
- JDBC is an API specification developed by Sun Microsystems that defines a uniform interface for accessing various relational databases.

JDBC
Definition

- JDBC: Java Database Connectivity
  - It provides a standard library for Java programs to connect to a database and send it commands using SQL
  - It generalizes common database access functions into a set of common classes and methods
  - Abstracts vendor specific details into a code library making the connectivity to multiple databases transparent to user
- JDBC API Standardizes:
  - Way to establish connection to database
  - Approach to initiating queries
  - Method to create stored procedures
  - Data structure of the query result

JDBC

API

- Two main packages java.sql and javax.sql
  - **Java.sql** contains all core classes required for accessing database (Part of Java 2 SDK, Standard Edition)
  - **Javax.sql** contains optional features in the JDBC 2.0 API (part of Java 2 SDK, Enterprise Edition)
- Javax.sql adds functionality for enterprise applications
  - DataSources
  - JNDI
  - Connection Pooling
  - Rowsets
  - Distributed Transactions

The java.sql Package

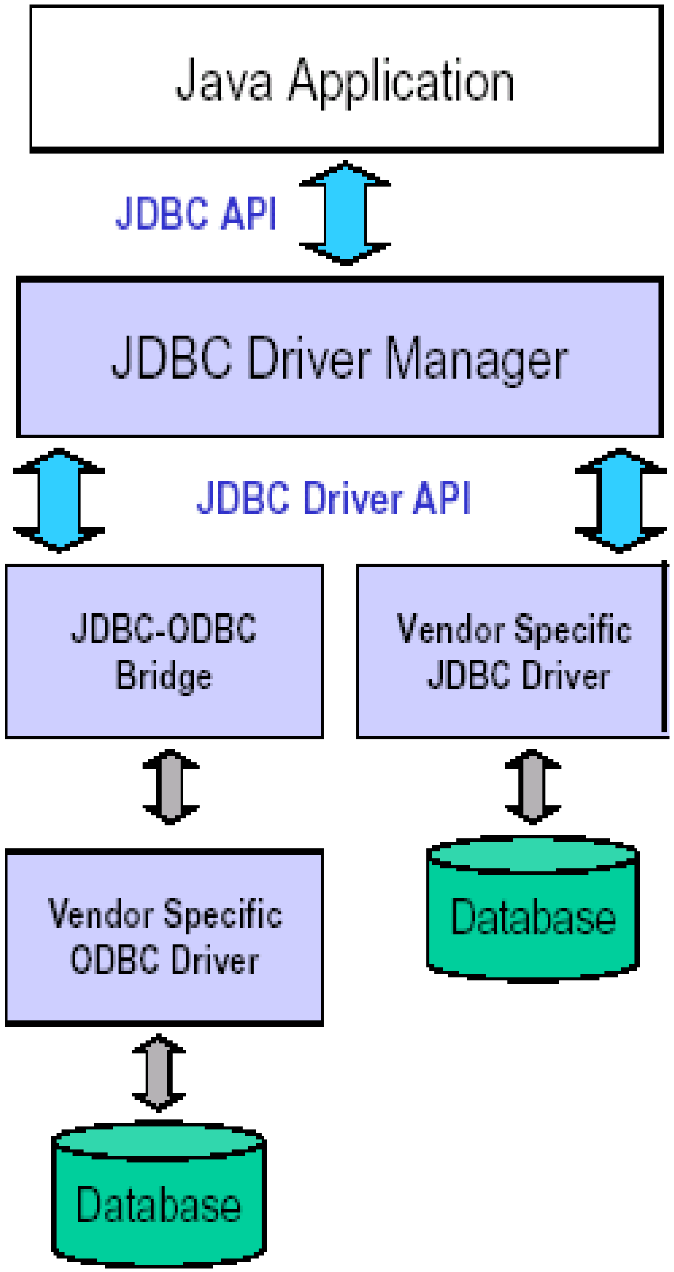
- Contains the entire JDBC API that sends SQL statements to relational databases and retrieves the results of executing those SQL statements.
- The Driver interface represents a specific JDBC implementation for a particular database system.
- Connection represents a connection to a database.
- The Statement, PreparedStatement, and CallableStatement interfaces support the execution of various kinds of SQL statements.
- ResultSet is a set of results returned by the database in response to a SQL query.
- The ResultSetMetaData interface provides metadata about a result set
- DatabaseMetaData provides metadata about the database as a whole.

The **javax.sql** package

- Contains the JDBC 2.0 Standard Extension API.
- The classes and interfaces in this package provide new functionality,
  - connection pooling, that do not fall under the scope of the original JDBC API and can therefore be safely packaged separately.
- The DataSource interface serves as a factory for Connection objects;
- DataSource objects can be registered with a JNDI (*Java Naming and Directory Interface*)server, making it possible to get the name of a database from a name service.
- PooledConnection supports connection pooling, which allows an application to handle multiple database connections in a fairly transparent manner.
- RowSet extends the ResultSet interface to a JavaBeans component that can be manipulated at design time and used with non-SQL data sources

JDBC
Architecture

- JDBC Consists of two parts:
  - JDBC API, a purely Java-based API
  - JDBC Driver Manager, which communicates with vendor-specific drivers that perform the real communication with the database
- Translation to the vendor format occurs on the client
  - No changes needed to the server
  - Driver (translator) needed on client



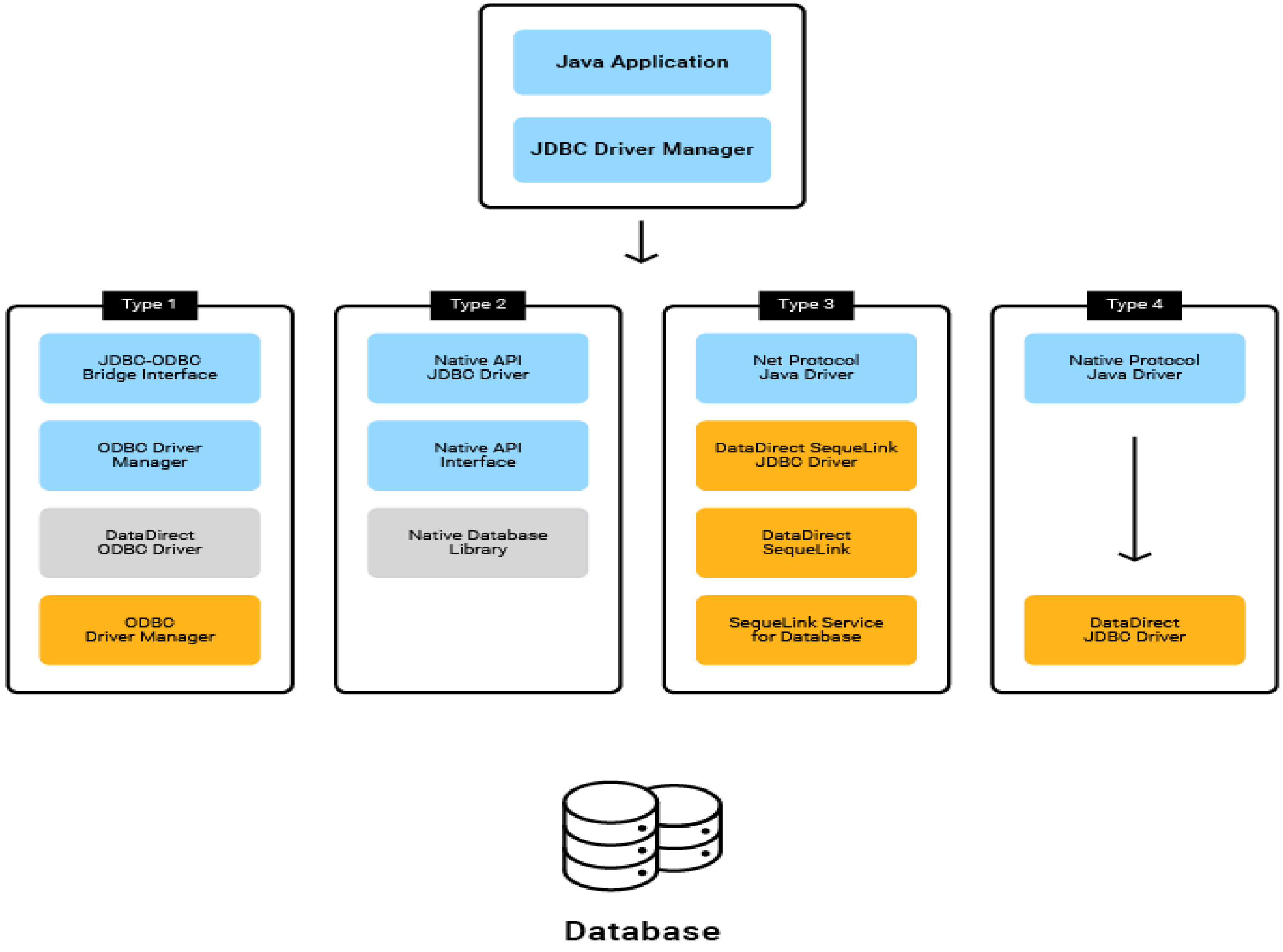
JDBC Architecture

- The primary function of the JDBC API is to provide a means for the developer to issue SQL statements and process the results in a consistent, database-independent manner.
- The JDBC driver manager ensures that the correct driver is used to access each data source.
- A JDBC driver translates standard JDBC calls into a network or database protocol or into a database library API call that facilitates communication with the database.

Driver types

- There are four types of drivers:
  - **JDBC Type 1 Driver** -- JDBC/ODBC Bridge drivers
    - ODBC (Open DataBase Connectivity) is a standard software API designed to be independent of specific programming languages
    - Sun provides a JDBC/ODBC implementation
  - **JDBC Type 2 Driver** -- use platform-specific APIs for data access
  - **JDBC Type 3 Driver** -- 100% Java, use a net protocol to access a remote listener and map calls into vendor-specific calls
  - **JDBC Type 4 Driver** -- 100% Java
    - Most efficient of all driver types

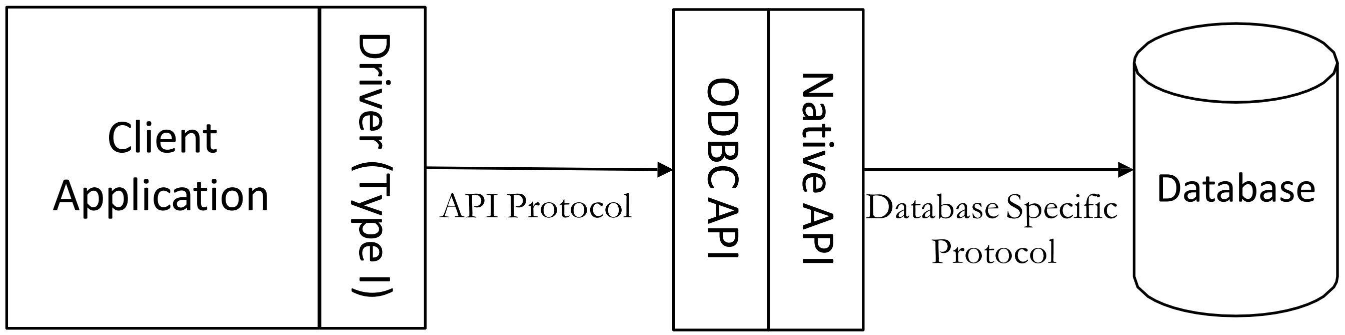
Driver types



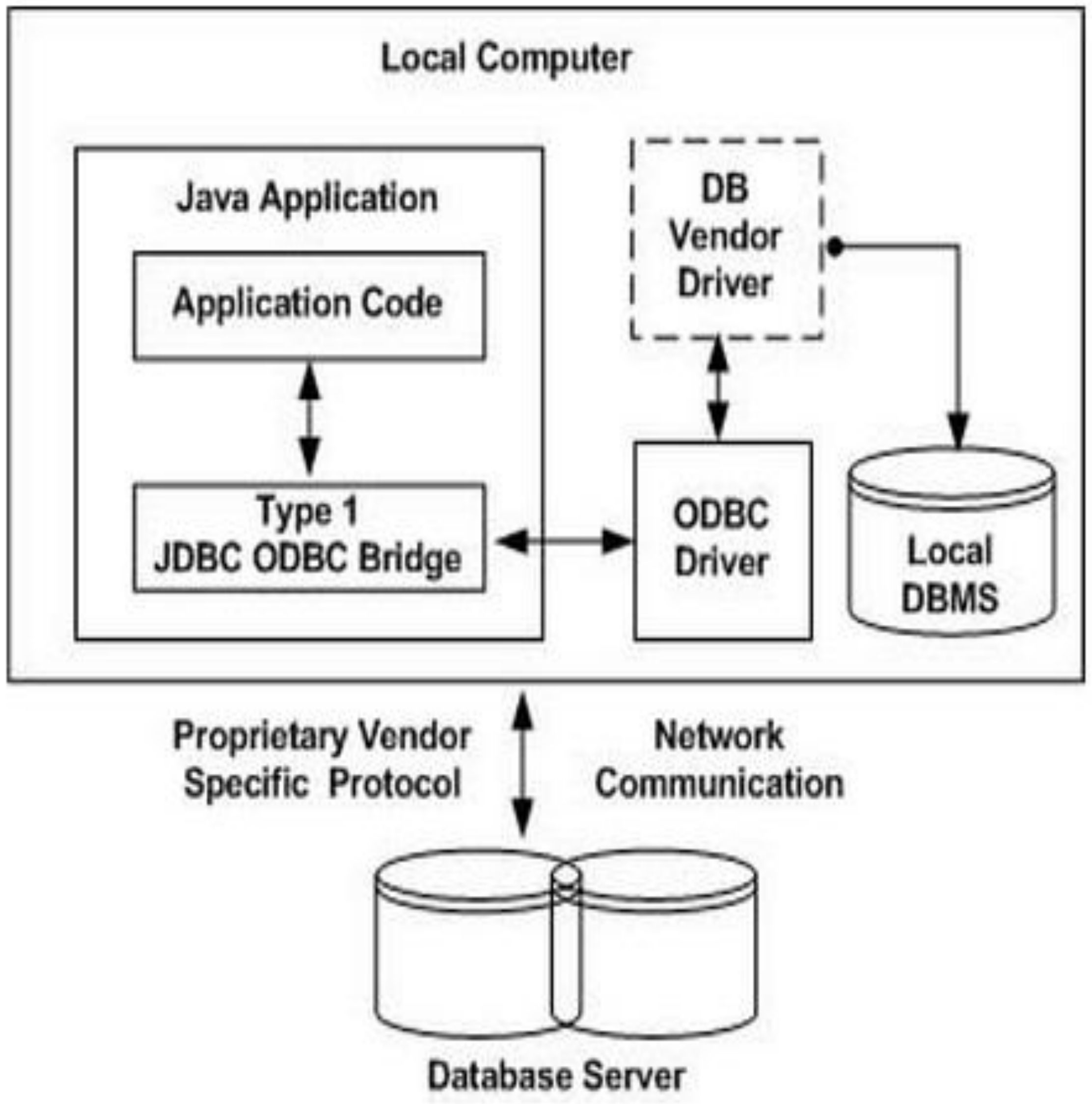
JDBC

Drivers (Type I)

- Type I driver provides mapping between JDBC and access API of a database
  - The access API calls the native API of the database to establish communication
- A common Type I driver defines a JDBC to ODBC bridge
  - ODBC is the database connectivity for databases
  - JDBC driver translates JDBC calls to corresponding ODBC calls
  - Thus if ODBC driver exists for a database this bridge can be used to communicate with the database from a Java application
- Inefficient and narrow solution
  - Inefficient, because it goes through multiple layers
  - Narrow, since functionality of JDBC code limited to whatever ODBC supports



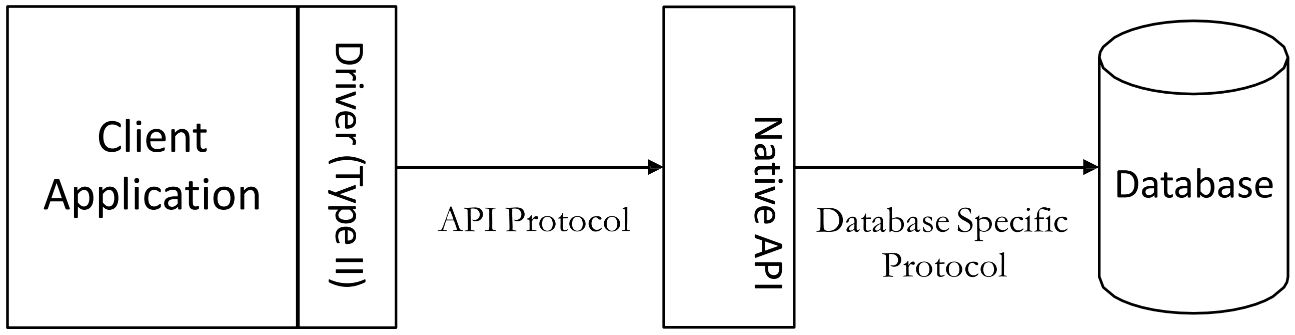
JDBC Drivers (Type I)



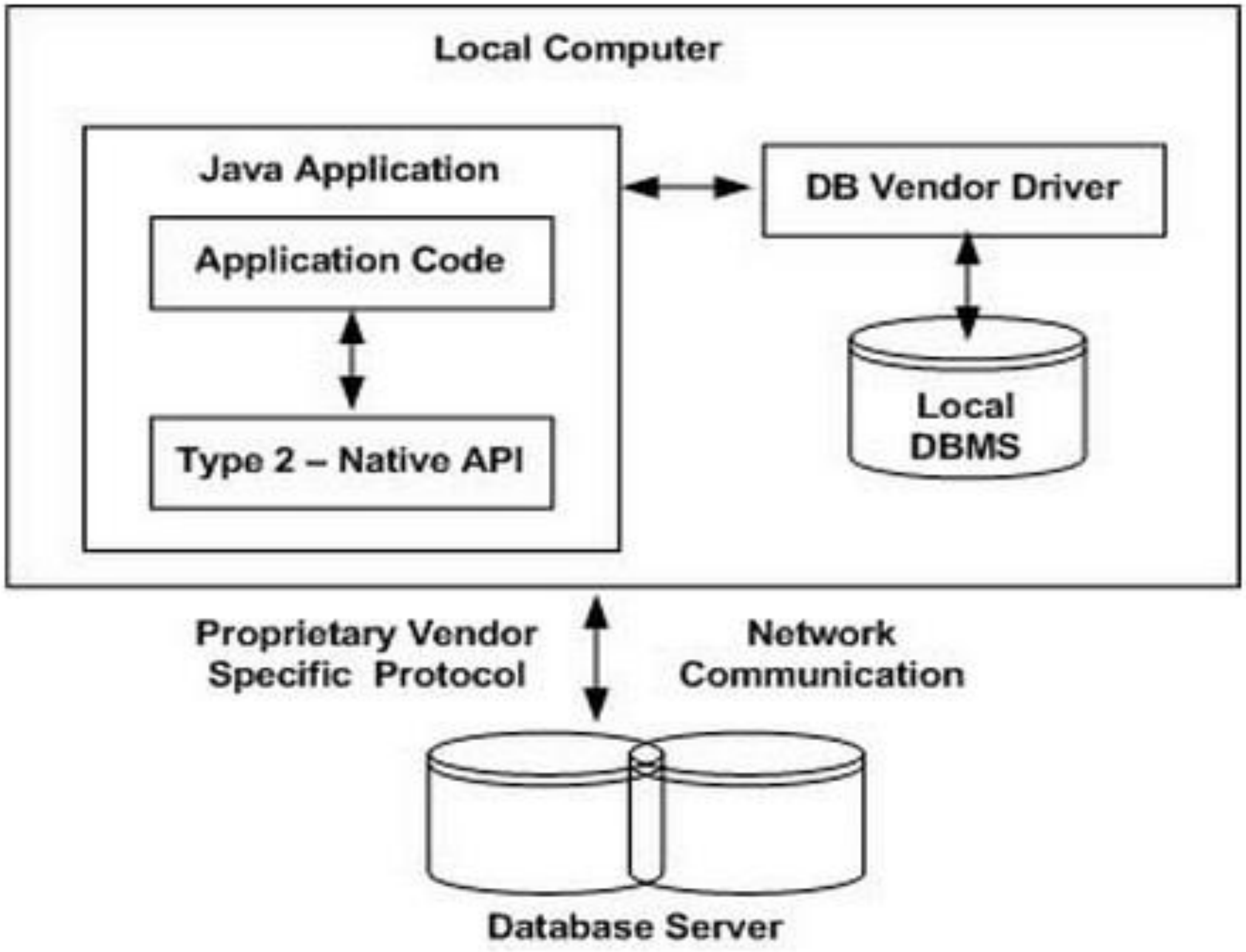
JDBC

Drivers (Type II)

- Type II driver communicates directly with native API
  - Type II makes calls directly to the native API calls
  - More efficient since there is one less layer to contend with (i.e. no ODBC)
  - It is dependent on the existence of a native API for a database



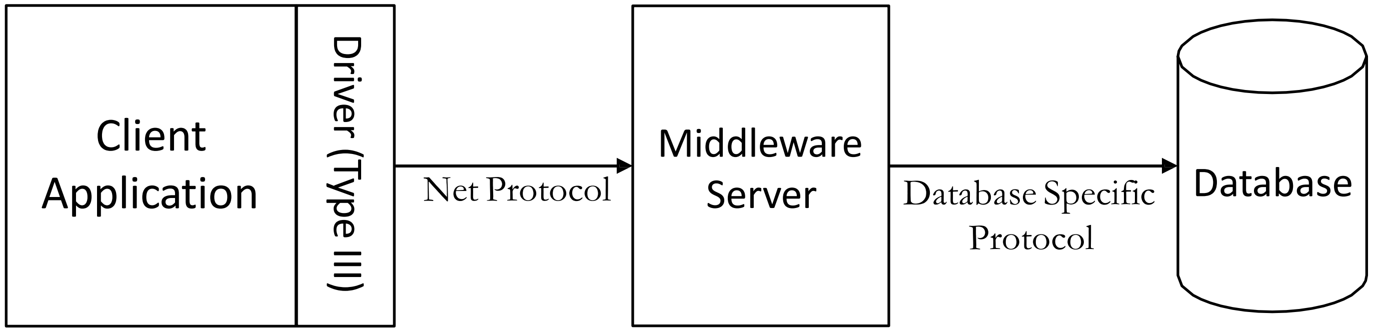
JDBC Drivers (Type II)



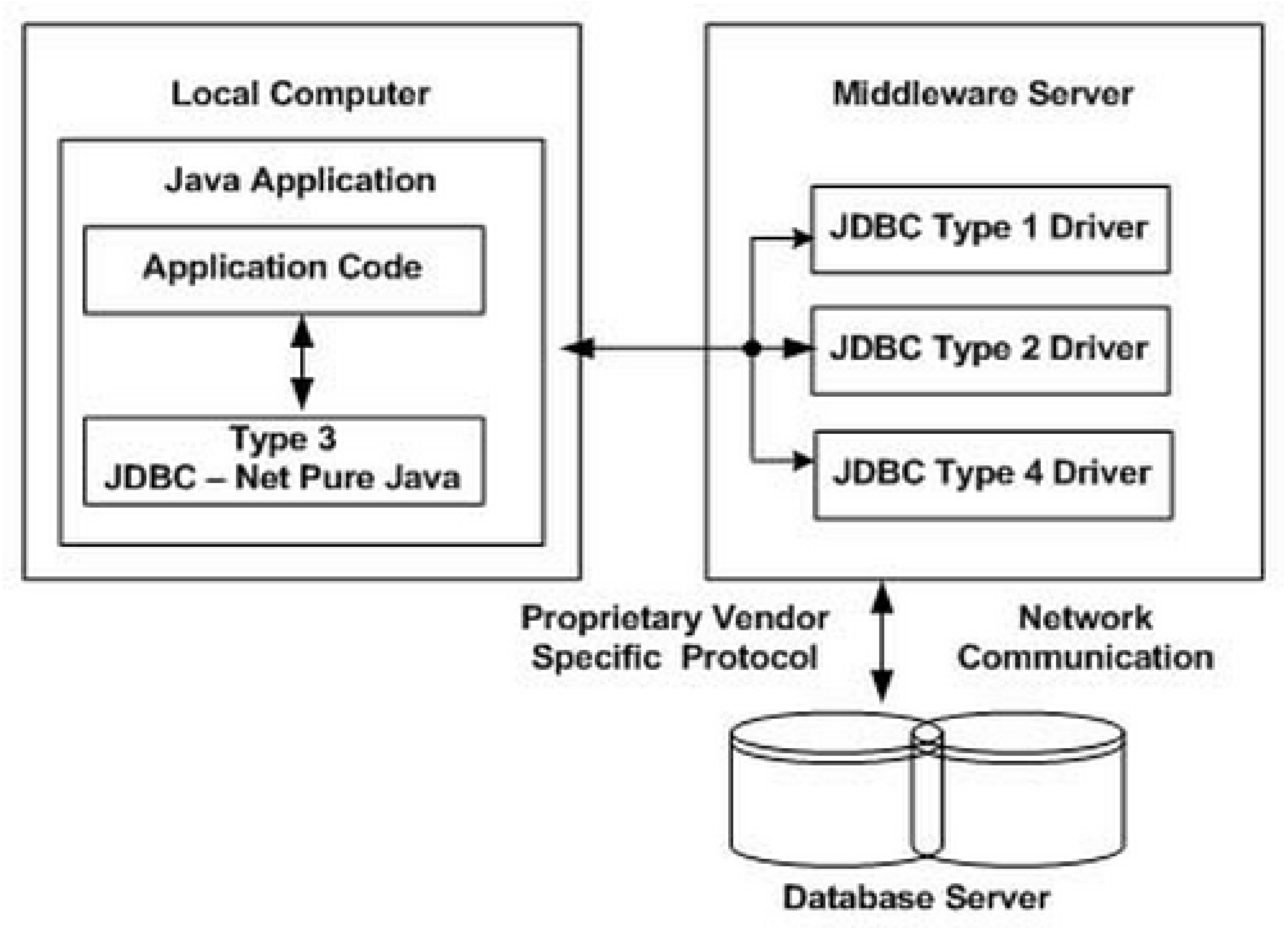
JDBC

Drivers (Type III)

- Type III driver make calls to a middleware component running on another server
  - This communication uses a database independent net protocol
  - Middleware server then makes calls to the database using database-specific protocol
  - The program sends JDBC call through the JDBC driver to the middle tier
  - Middle-tier may use Type I or II JDBC driver to communicate with the database.



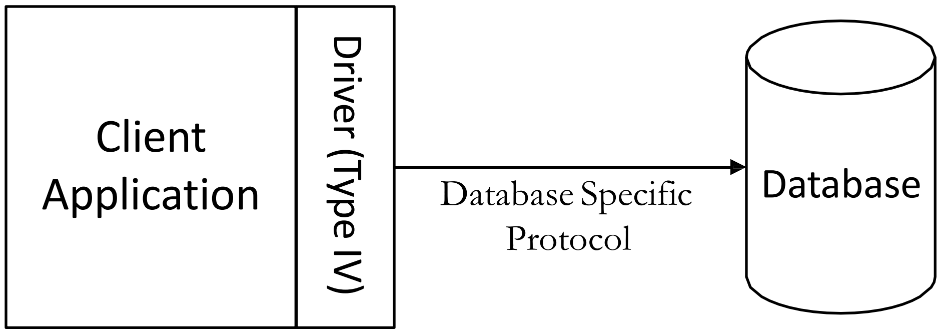
JDBC Drivers (Type III)



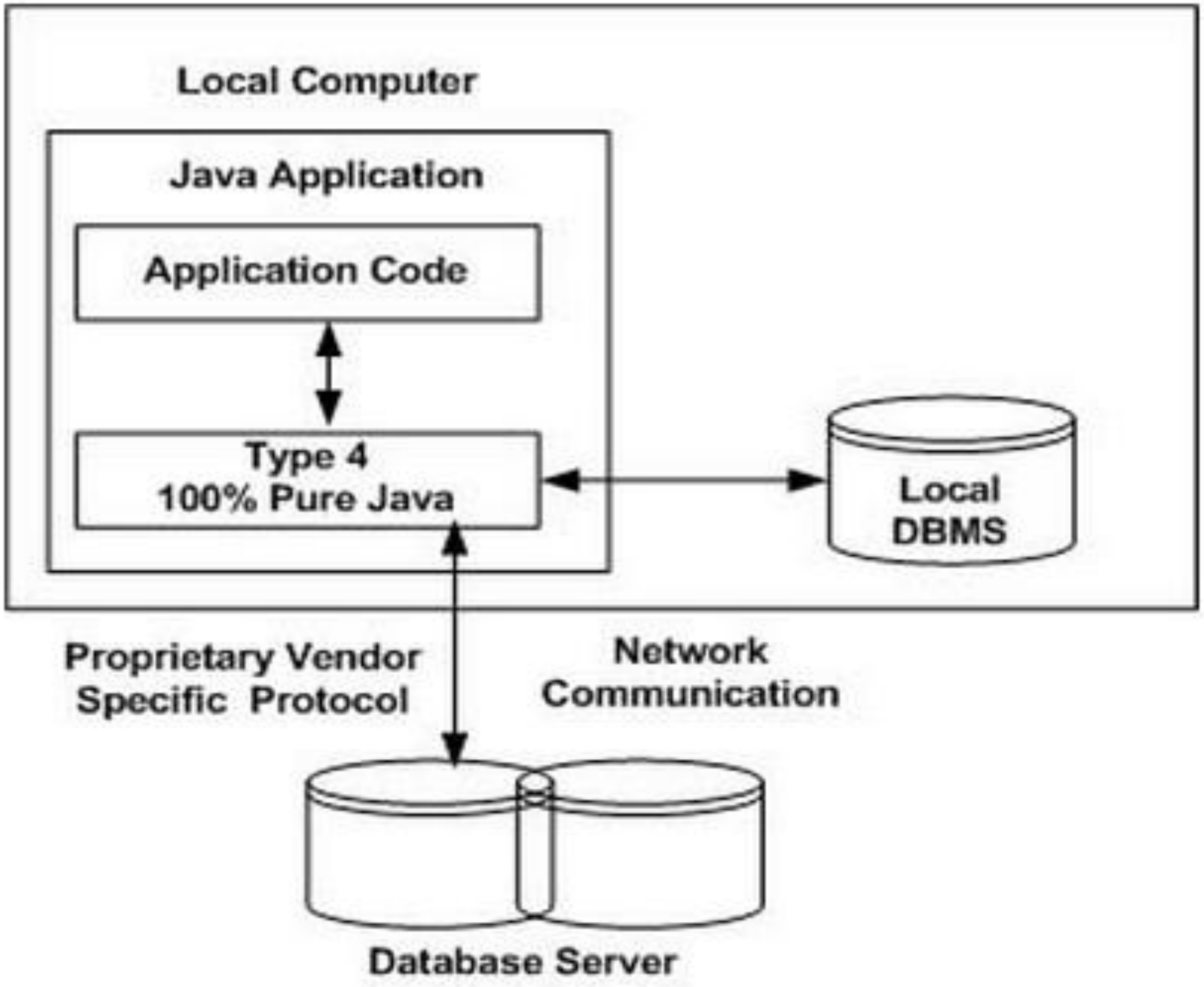
JDBC

Drivers (Type IV)

- Type IV driver is an all-Java driver that is also called a thin driver
  - It issues requests directly to the database using its native protocol
  - It can be used directly on platform with a JVM
  - Most efficient since requests only go through one layer
  - Simplest to deploy since no additional libraries or middle-ware



JDBC Drivers (Type IV)



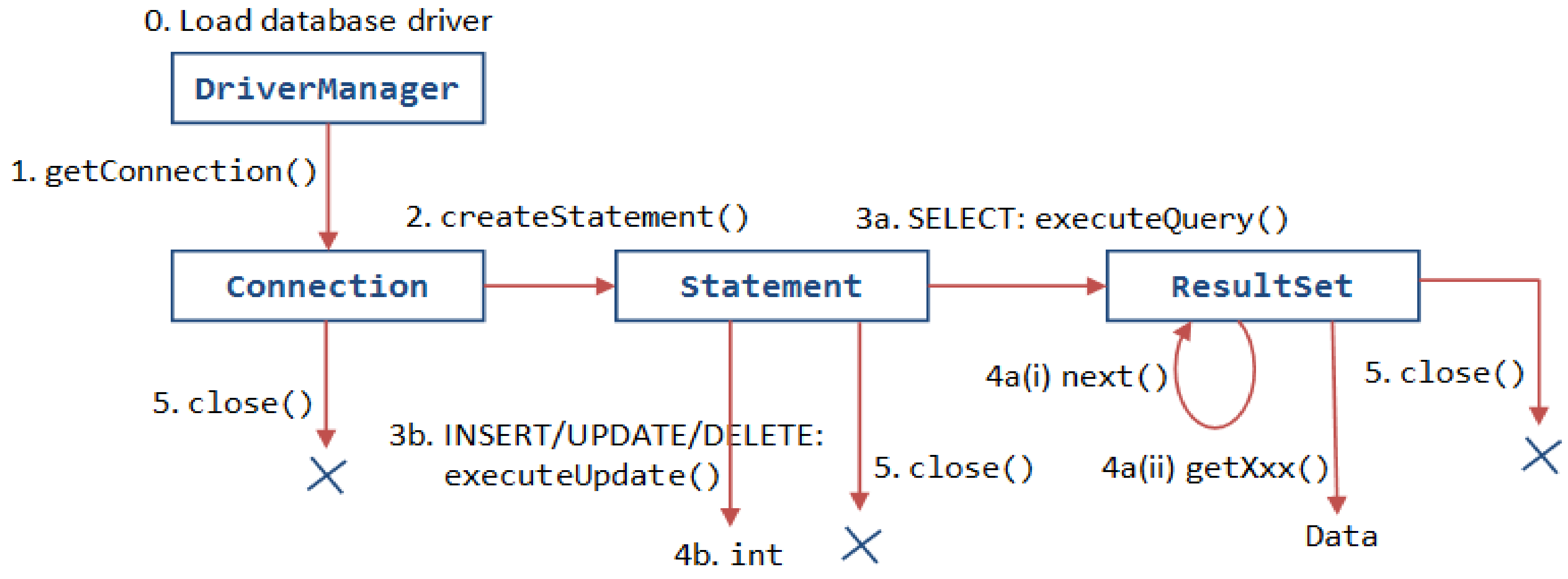
Which Driver should be Used?

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, 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

Conceptual Components

- **Driver Manager:** Loads database drivers and manages connections between the application and the driver
- **Driver:** Translates API calls into operations for specific database
- **Connection:** Session between application and data source
- **Statement:** SQL statement to perform query or update
- **Metadata:** Information about returned data, database, & driver
- **Result Set:** Logical set of columns and rows of data returned by executing a statement



Basic Steps to use database in Java

1. Establish a **connection**
2. Create JDBC **Statements**
3. Execute **SQL** Statements
4. GET **ResultSet**
5. **Close** connections

1. Establish a connection

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

  Method : Class.forName()

  Code:

```
try { Class.forName("com.mysql.jdbc.Driver");
}
catch(Exception e) {
        System.out.println(e);
}
```

  - Dynamically loads a driver class, for mysql database

- **Make the connection**

  ■The Connection interface represents a connection to the database. An instance of the Connection interface is obtained from the getConnection method of the DriverManager class.

  ■Method : DriverManager.getConnection()

java.sql [Imp: Class & Interface]

- The DriverManager class
- The Connection interface
- The Statement interface
- The ResultSet interface
- The PreparedStatement interface

DriverManager

- DriverManager class : The DriverManager class provides static methods for managing JDBC drivers. Each JDBC driver you want to use must be registered with the DriverManager.

- Following table lists down the popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format	URL Example
MySQL	com.mysql.jdbc.Driver	**jdbc:mysql://**hostname/ databaseName	jdbc:mysql://localhost/ mca
ORACLE	oracle.jdbc.driver.Oracle Driver	**jdbc:oracle:thin:@**hostname:p ort Number:databaseName	jdbc:oracle:thin:@mcas erver.com:1521:ormca
DB2	COM.ibm.db2.jdbc.net.D B2Driver	**jdbc:db2:**hostname:port Number/databaseName	
Sybase	com.sybase.jdbc.SybDriv er	**jdbc:sybase:Tds:**hostname: port Number/databaseName	

Step 1: Example

Code:

```
try {
	Class.forName(“com.mysql.jdbc.Driver”);
	Connection con = DriverManager.getConnection(
	"jdbc:mysql://localhost/mydb", "root", "admin");

} catch(Exception e) {
	System.out.println(e);
}
```

2. Create JDBC statement(s)

- Use for general-purpose access to your database.
- Useful when you are using static SQL statements at runtime.
- **Method: connection.createStatement();**

Code:

```
try {
      Class.forName(“com.mysql.jdbc.Driver”);
      Connection con = DriverManager.getConnection(
                              "jdbc:mysql://localhost/mydb", "root", "admin");
      Statement stat=con.createStatement();
} catch(Exception e) {
      System.out.println(e);
}
```

Statement – Important Methods

- executeUpdate() : Execute Create, Insert, Update, and Delete SQL statement.
- executeQuery() : Executes an SQL statement that returns a single ResultSet object.
- execute() : Execute Procedure and Functions.

3. Executing SQL Statements

Code:

```
try {
    Class.forName(“com.mysql.jdbc.Driver”);
    Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost/mydb", "root", "admin");
    Statement stat=con.createStatement();

    int rel=stat.executeUpdate("insert into user
    values('"+username+"','"+password+"','"+firstname+"','"+
    lastname+"')");
}
```

4. ResultSet Interface

- The ResultSet interface represents a table-like database result set. A ResultSet object maintains a cursor pointing to its current row of data. Initially, the cursor is positioned before the first row.
- Important Method : next();

Code:

```
Class.forName(“com.mysql.jdbc.Driver”);
Connection con = DriverManager.getConnection(
"jdbc:mysql://localhost/mydb", "root", "admin");
Statement stat=con.createStatement();
ResultSet rs = stat.executeQuery(“select * from user”);
while (rs.next()) {
   username= rs.getString(“username”);
   name= rs.getString(“firstname”);
}
rs.close();
```

5. Close connection

Code:

```
Class.forName(“com.mysql.jdbc.Driver”);
Connection con = DriverManager.getConnection(
"jdbc:mysql://localhost/mydb", "root", "admin");
Statement stat=con.createStatement();
int rel=stat.executeUpdate("insert into user values('"+username+"','"+password+"','"+firstname+"','"+lastname+"')");
if(rel==1) { }
stat.close();
con.close();
```

JDBC Classes

- DriverManager
  - Manages JDBC Drivers
  - Used to Obtain a connection to a Database
- Types
  - Defines constants which identify SQL types
- Date
  - Used to Map between java.util.Date and the SQL DATE type
- Time
  - Used to Map between java.util.Date and the SQL TIME type
- TimeStamp
  - Used to Map between java.util.Date and the SQL TIMESTAMP type

JDBC Interfaces

- ResultSet
  - Represents the result of an SQL statement
  - Provides methods for navigating through the resulting data
- PreparedStatement
  - Similar to a stored procedure
  - An SQL statement (which can contain parameters) is compiled and stored in the database
- CallableStatement
  - Used for executing stored procedures
- DatabaseMetaData
  - Provides access to a database's system catalogue
- ResultSetMetaData
  - Provides information about the data contained within a ResultSet

Using a ResultSet

- The ResultSet interface defines many navigation methods

  public boolean first()
  public boolean last()
  public boolean next()
  public boolean previous()

- The ResultSet interface also defines data access methods

  public int getInt(int columnNumber)   -- Note: Columns are numbered
  public int getInt(String columnName) -- from 1 (not 0)
  public long getLong(int columnNumber)
  public long getLong(String columnName)
  public String getString(int columnNumber)
  public String getString(String columnName)

- There are MANY more methods. Check the API documentation for a complete list

SQL Types/Java Types Mapping

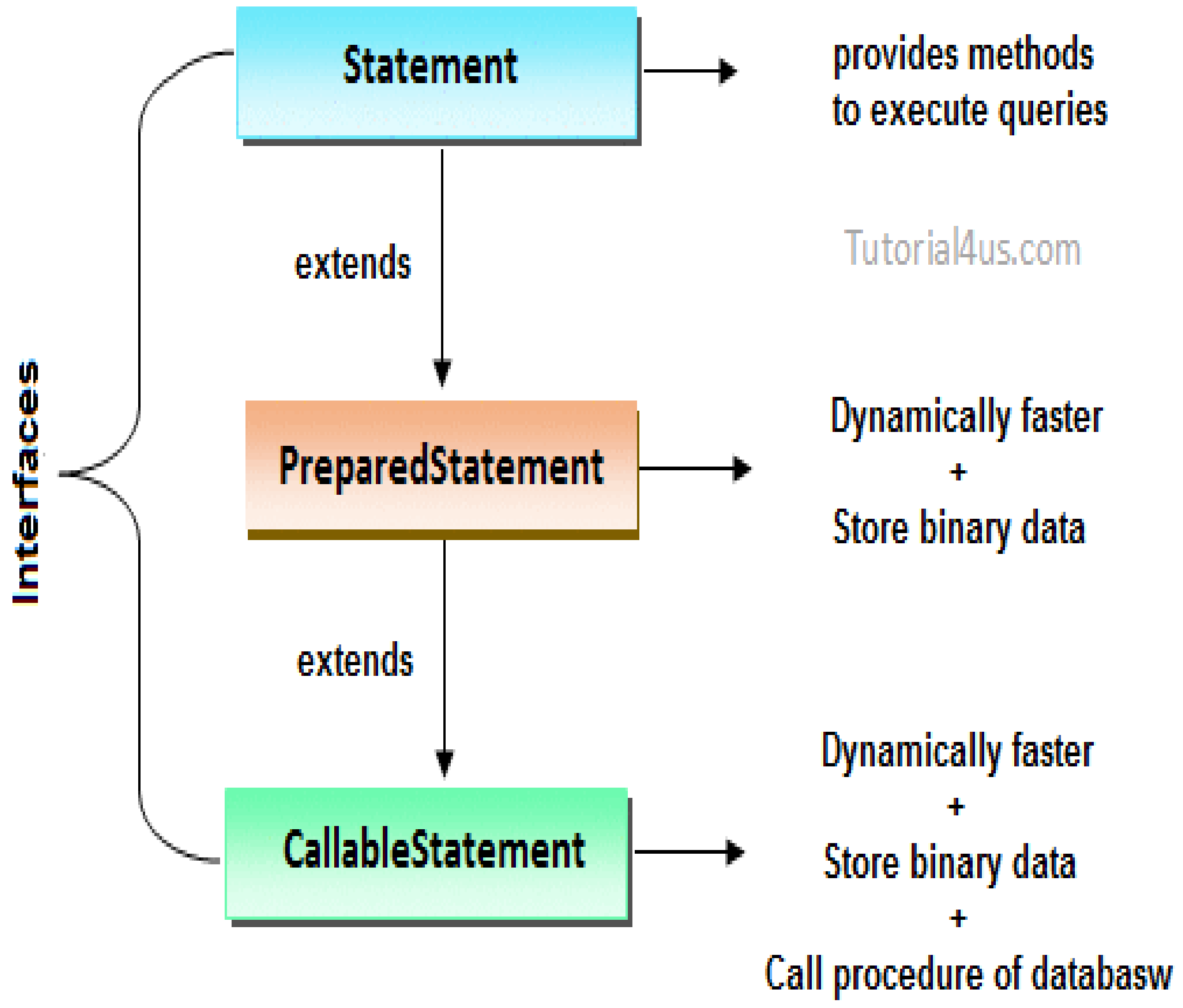
SQL Type	*Java Type*
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.Math.BigDecimal
DECIMAL	java.Math.BigDecimal
BIT	boolean
TINYINT	int
SMALLINT	int
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP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

Statement

Types

- Statements in JDBC abstract the SQL statements
- Primary interface to the tables in the database
- Used to create, retrieve, update & delete data (CRUD) from a table
  - Syntax: Statement statement = connection.createStatement();
- Three types of statements each reflecting a specific SQL statements
  - Statement
  - PreparedStatement
  - CallableStatement

PreparedStatement


Statement
provides methods to execute queries
extends
Tutorial4us.com
PreparedStatement
Dynamically faster + Store binary data
extends
CallableStatement
Dynamically faster + Store binary data + Call procedure of databasw
Interfaces

Statement Object

Example:

```
Statement stmt = null; try {
stmt = conn.createStatement( );
. . .
}
catch (SQLException e) {
. . .
}
finally {
   stmt.close();
}
```

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

boolean execute *StringSQL*:

- Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false.
- Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.

int executeUpdate *StringSQL*:

- Returns the number of rows affected by the execution of the SQL statement.
- Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.

ResultSet executeQuery *StringSQL*:

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

PreparedStatement Objects

Example:

```
PreparedStatement pstmt = null; try {
String SQL = "Update Employees
SET age = ? WHERE id = ?"; pstmt
= conn.prepareStatement(SQL);
. . .
}
catch (SQLException e) {
. . .
}
finally {
  pstmt.close();
}
```

•All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker.

•The **setXXX** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter.

•Each parameter marker is referred by its ordinal position.

•The first marker represents position 1, the next position 2, and so forth.

•This method differs from that of Java array indices, which starts at 0.

•All of the **Statement object's** methods for interacting with the database also work with the PreparedStatement object.

•However, the methods are modified to use SQL statements that can input the parameters.

CallableStatement Object

Example: Stored Procedure

```
CREATE OR REPLACE PROCEDURE
getEmpName
(EMP_ID IN NUMBER, EMP_FIRST OUT
VARCHAR) AS BEGIN
SELECT first INTO EMP_FIRST
FROM Employees WHERE ID = EMP_ID;
END;

CallableStatement cstmt = null; try {
String SQL = "{call getEmpName (?,
?)}"; cstmt = conn.prepareCall (SQL);
. . .
}
catch (SQLException e) {
. . .
}
finally {
  cstmt.close();
}
```

•The String variable SQL, represents the stored procedure, with parameter placeholders.
•If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX method that corresponds to the Java data type you are binding.
•When you use OUT and INOUT parameters you must additionally use CallableStatement method, registerOutParameter.
•The registerOutParameter method binds the JDBC data type, to the data type that the stored procedure is expected to return.
•Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX method. This method casts the retrieved value of SQL type to a Java data type.

Comparison between Statement, Prepared Statement and Callable Statement

Statement	**Prepared Statement**	**Callable Statement**
Super interface for Prepared and Callable Statement	extends Statement (sub-interface)	extends PreparedStatement (sub-interface)
Used for executing simple SQL statements like CRUD (create, retrieve, update and delete	Used for executing dynamic and pre-compiled SQL statements	Used for executing stored procedures
The Statement interface cannot accept parameters.	The PreparedStatement interface accepts input parameters at runtime.	The CallableStatement interface can also accept runtime input parameters.
stmt = conn.createStatement();	PreparedStatement ps=con.prepareStatement ("insert into studentDiet values(?,?,?)");	CallableStatement cs=conn.prepareCall("{call getbranch(?,?)}");
java.sql.Statement is slower as compared to Prepared Statement in java JDBC.	PreparedStatement is faster because it is used for executing precompiled SQL statement in java JDBC.	None
java.sql.Statement is suitable for executing DDL commands - CREATE, drop, alter and truncate in java JDBC.	java.sql.PreparedStatement is suitable for executing DML commands - SELECT, INSERT, UPDATE and DELETE in java JDBC.	java.sql.CallableStatement is suitable for executing stored procedure.

Querying the Database

Executing Queries
Methods

- Two primary methods in statement interface used for executing Queries
  - executeQuery Used to retrieve data from a database
  - executeUpdate: Used for creating, updating & deleting data
- executeQuery used to retrieve data from database
  - Primarily uses Select commands
- executeUpdate used for creating, updating & deleting data
  - SQL should contain Update, Insert or Delete commands
- Uset setQueryTimeout to specify a maximum delay to wait for results

Executing Queries

Data Definition Language (DDL)

- Data definition language queries use executeUpdate
- Syntax: int executeUpdate(String sqlString) throws SQLException
  - It returns an integer which is the number of rows updated
  - sqlString should be a valid String else an exception is thrown
- Example 1: Create a new table

```
Statement statement = connection.createStatement();
String sqlString =
"Create Table Catalog"
+ "(Title Varchar(256) Primary Key Not Null,"+
+ "LeadActor Varchar(256) Not Null, LeadActress Varchar(256) Not Null,"
+ "Type Varchar(20) Not Null, ReleaseDate Date Not NULL )";
Statement.executeUpdate(sqlString);
```

  - executeUpdate returns a zero since no row is updated

Executing Queries

DDL (Example)

- Example 2: Update table

  Statement statement =

  connection.createStatement(); String sqlString =

  “Insert into Catalog”

  + “(Title, LeadActor, LeadActress, Type, ReleaseDate)”

  + “Values("Gone With The Wind",“Clark Gable", "Vivien Liegh“,+ “Romantic“,02/18/2003“)

  Statement.executeUpdate(sqlString);

  – executeUpdate returns a 1 since one row is added

Executing Queries
Data Manipulation Language (DML)

- Data definition language queries use executeQuery
- Syntax

  ResultSet executeQuery(String sqlString) throws SQLException

  – It returns a ResultSet object which contains the results of the Query
- Example 1: Query a table

```
Statement statement = connection.createStatement();
String sqlString = “Select Catalog.Title, Catalog.LeadActor,
                    Catalog.LeadActress,” + “Catalog.Type,
                    Catalog.ReleaseDate From Catalog”;
ResultSet rs = statement.executeQuery(sqlString);
```

ResultSet

Definition

- ResultSet contains the results of the database query that are returned
- Allows the program to scroll through each row and read all columns of data
- ResultSet provides various access methods that take a column index or column name and returns the data
  - All methods may not be applicable to all resultsets depending on the method of creation of the statement.
- When the executeQuery method returns the ResultSet the cursor is placed before the first row of the data
  - Cursor refers to the set of rows returned by a query and is positioned on the row that is being accessed
  - To move the cursor to the first row of data next() method is invoked on the resultset
  - If the next row has a data the next() results true else it returns false and the cursor moves beyond the end of the data
- First column has index 1, not 0

ResultSet

- ResultSet contains the results of the database query that are returned
- Allows the program to scroll through each row and read all the columns of the data
- ResultSet provides various access methods that take a column index or column name and returns the data
  - All methods may not be applicable to all resultsets depending on the method of creation of the statement.
- When the executeQuery method returns the ResultSet the cursor is placed before the first row of the data
  - Cursor is a database term that refers to the set of rows returned by a query
  - The cursor is positioned on the row that is being accessed
  - First column has index 1, not 0
- Depending on the data numerous functions exist
  - getShort(), getInt(), getLong()
  - getFloat(), getDouble()
  - getClob(), getBlob(),
  - getDate(), getTime(), getArray(), getString()

ResultSet

- Examples:
  - Using column Index:

    Syntax:public String getString(int columnIndex) throws SQLException

    e.g. ResultSet rs = statement.executeQuery(sqlString);

    String data = rs.getString(1)
  - Using Column name

    public String getString(String columnName) throws SQLException

    e.g. ResultSet rs = statement.executeQuery(sqlString);

    String data = rs.getString(Name)
- The ResultSet can contain multiple records.
  - To view successive records next() function is used on the ResultSet
  - Example: while(rs.next()) {
  - System.out.println(rs.getString); }

Scrollable ResultSet

- ResultSet obtained from the statement created using the no argument constructor is:
  - Type forward only (non-scrollable)
  - Not updateable
- To create a scrollable ResultSet the following statement constructor is required
  - Statement createStatement(int resultSetType, int resultSetConcurrency)
- ResultSetType determines whether it is scrollable. It can have the following values:
  - ResultSet.TYPE_FORWARD_ONLY
  - ResultSet.TYPE_SCROLL_INSENSITIVE (Unaffected by changes to underlying database)
  - ResultSet.TYPE_SCROLL_SENSITIVE (Reflects changes to underlying database)
- ResultSetConcurrency determines whether data is updateable. Its possible values are
  - CONCUR_READ_ONLY
  - CONCUR_UPDATEABLE
- On a scrollable ResultSet the following commands can be used
  - boolean next(), boolean previous(), boolean first(), boolean last()
  - void afterLast(), void beforeFirst()
  - boolean isFirst(), boolean isLast(), boolean isBeforeFirst(), boolean isAfterLast()
- Not all database drivers may support these functionalities

RowSet

- ResultSets limitation is that it needs to stay connected to the data source
  - It is not serializable and can not transport across the network
- RowSet is an interface which removes the limitation
  - It can be connected to a dataset like the ResultSet
  - It can also cache the query results and detach from the database
- RowSet is a collection of rows
- RowSet implements a custom reader for accessing any tabular data
  - Spreadsheets, Relational Tables, Files
- RowSet object can be serialized and hence sent across the network
- RowSet object can update rows while diconnected fro the data source
  - It can connect to the data source and update the data
- Three separate implementations of RowSet
  - CachedRowSet
  - JdbcRowSet
  - WebRowSet

RowSet

- RowSet is derived from the BaseRowSet
  - Has SetXXX(…) methods to supply necessary information for making connection and executing a query
- Once a RowSet gets populated by execution of a query or from some other data source its data can be manipulated or more data added
- Three separate implementations of RowSet exist
  - CachedRowSet: Disconnected from data source, scrollable & serilaizable
  - JdbcRowSet: Maintains connection to data source
  - WebRowSet: Extension of CachedRowSet that can produce representation of its contents in XML

MetaData

- Meta Data means data about data
- Two kinds of meta data in JDBC
  - Database Metadata: To look up information about the database (here)
  - ResultSet Metadata: To get the structure of data that is returned (later)
- Example
  - connection.getMetaData().getDatabaseProductName()
  - connection.getMetaData().getDatabaseProductVersion()
- Sample Code:

```
private void showInfo(String driver,String url,String user,String password,
 String table,PrintWriter out) {
        Class.forName(driver);
        Conntection con = DriverManager.getConnection(url, username, password);
        DatabaseMetaData dbMetaData = connection.getMetaData();
        String productName = dbMetaData.getDatabaseProductName();
        System.out.println("Database: " + productName);
        String productVersion = dbMetaData.getDatabaseProductVersion();
        System.out.println("Version: " + productVersion);
}
```