JDBC – Java DB Connectivity

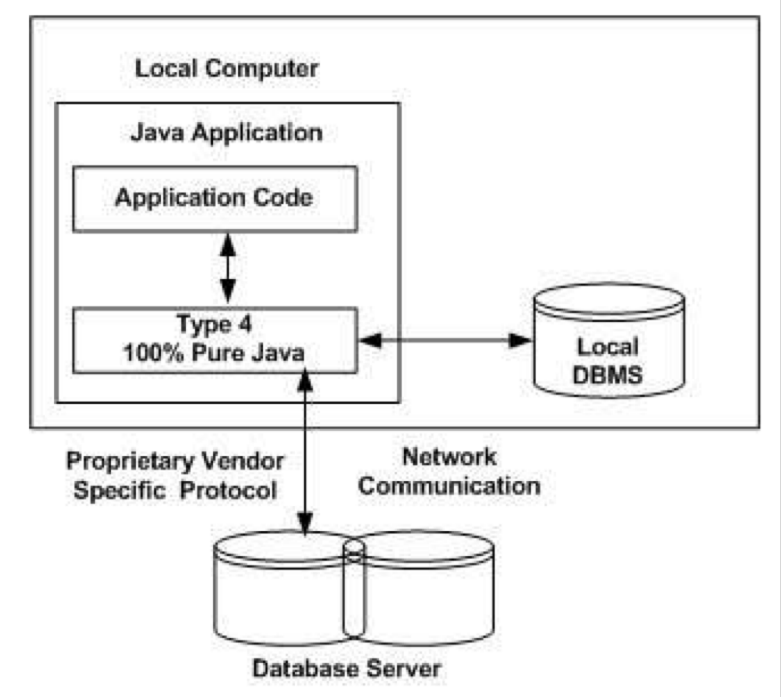
Università di Modena e Reggio Emilia

Prof. Nicola Bicocchi (nicola.bicocchi@unimore.it)

ING

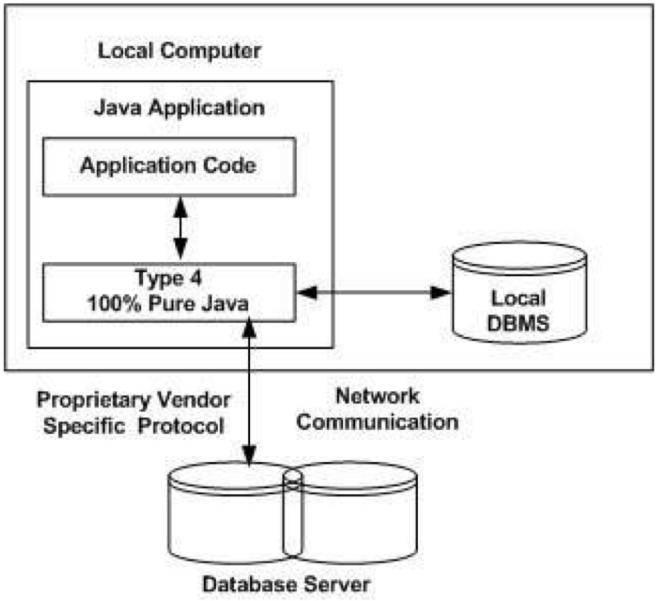
Networked DBMS

- The most of DBMS make use of the TCP protocol for communicating with applications. They accept incoming connections on a specific TCP port. This allows both local and remote connections.
  - MS SQL Server (TCP:1433)
  - PostGreSQL (TCP:5432)
  - MySQL (TCP:3306)
  - Oracle (TCP:1521)
  - SQLite (*local DB*)



Local DB

- There is also a family of libraries capable of simulating a DBMS connection while providing access to a local file using the SQL metaphor. The most promiment example of this category is SQLite.
  - Widely used on the Android platform
  - https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase



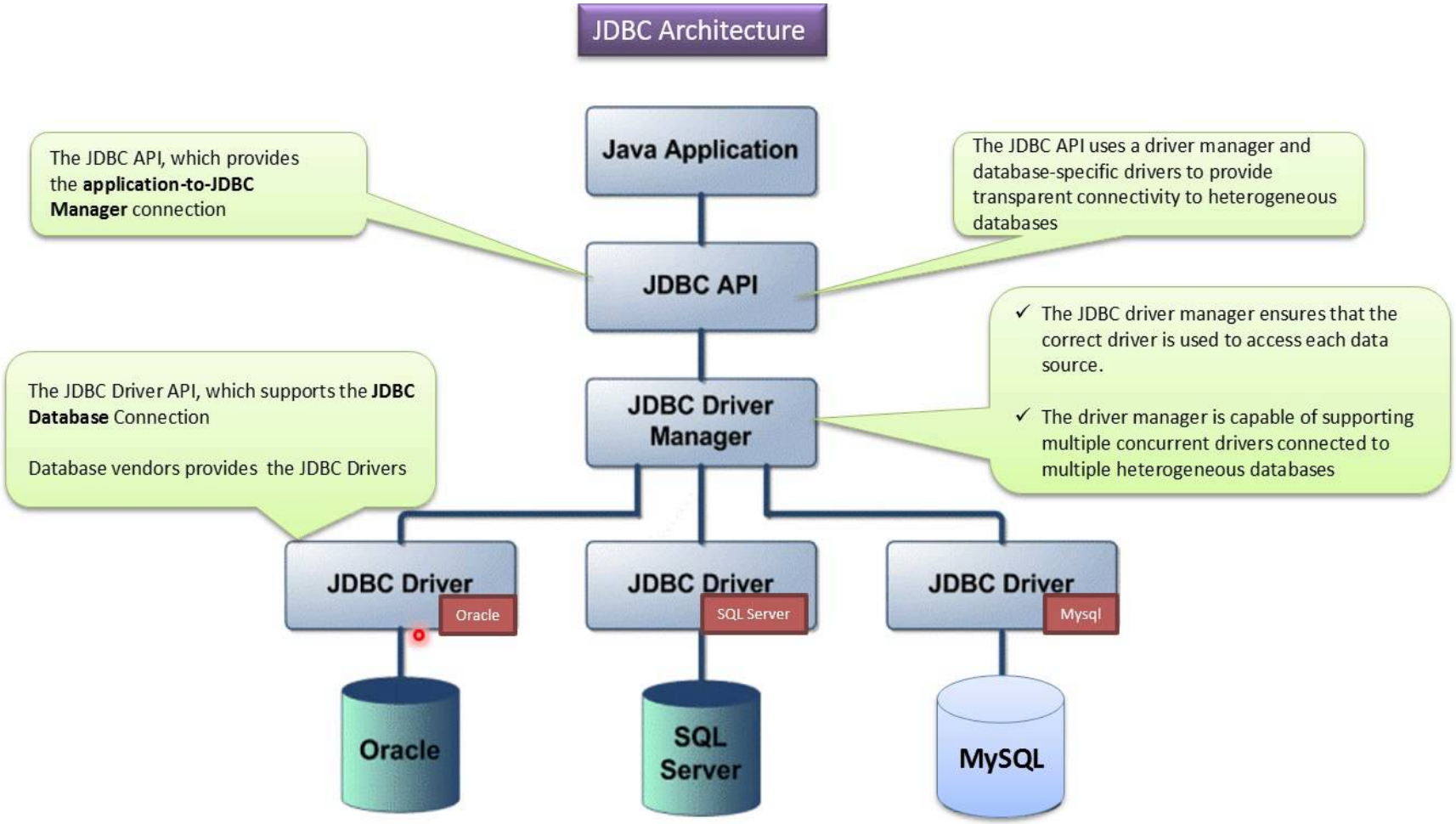
ING

What is JDBC?

- *"An API that lets you access virtually any tabular data source from the Java programming language"*
  - What's an API? *Application Programming Interface*
  - What's a tabular data source? Relational databases, spreadsheets, CSV files
- We'll focus on accessing relational databases. Nevertheless, the same principles can be applied to all data sources.

ING

General Architecture



Basic steps

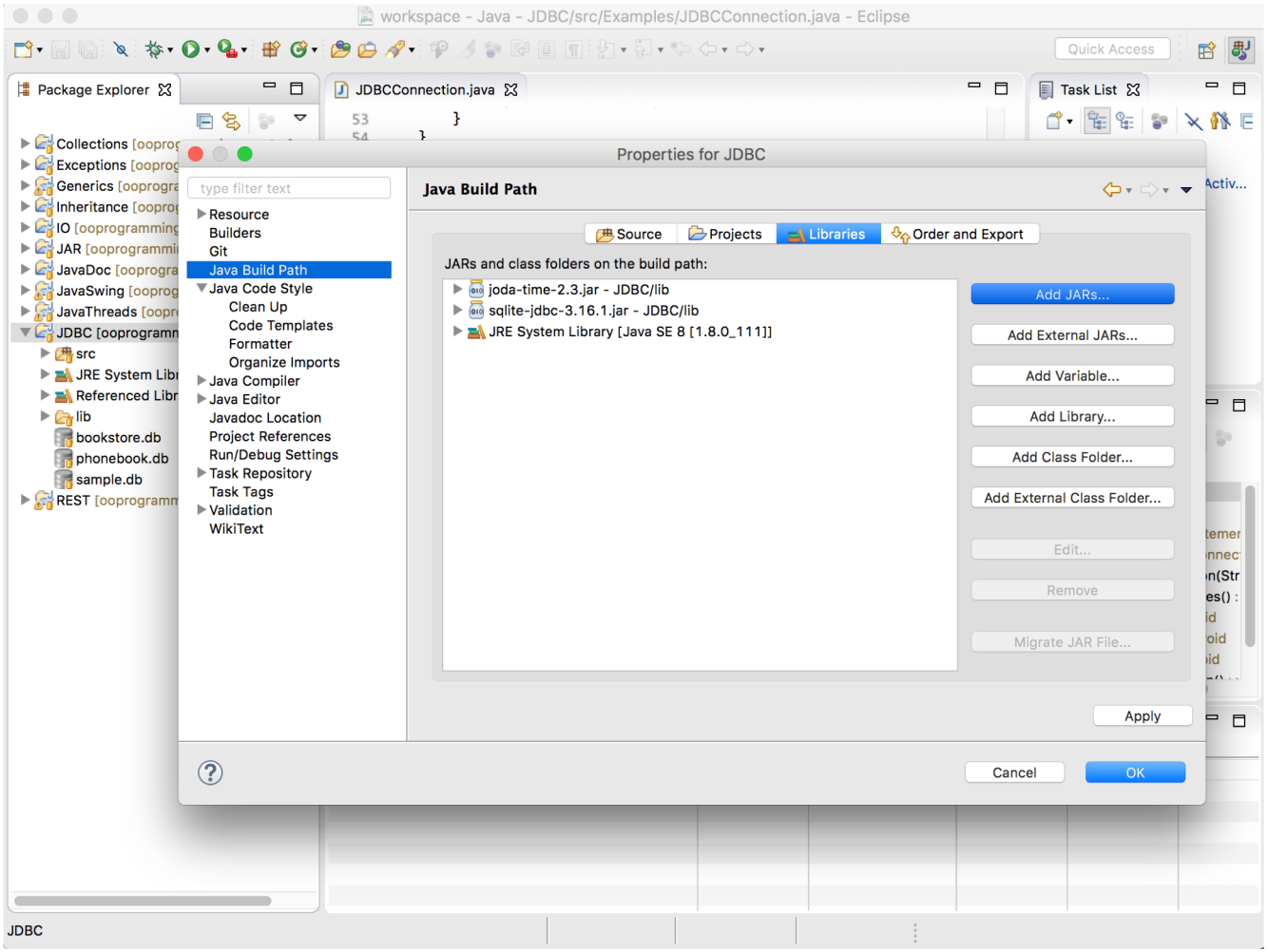
1. Load vendor specific Driver
2. Establish a Connection
3. Create a Statement
4. Execute SQL Statements
5. Get ResultSet
6. Close the Connection

Vendor specific drivers

- JDBC drivers provide the connection to the database and implement the protocol for transferring queries and results between the client and the database.
- There are 4 type of drivers. We refer to Type 4: Pure Java (see Appendix II)
- Each database needs a specific driver. They need to be downloaded separately
  - [mySQL] https://dev.mysql.com/downloads/connector/j/
  - [SQLite] https://github.com/xerial/sqlite-jdbc
- Drivers are Java binary classes (.class files) and must be included into the CLASSPATH

ING

Vendor specific drivers



1. Load vendor specific driver

```
import java.sql.*;

/* this is for MySQL*/
Class.forName("com.mysql.jdbc.Driver");

/* this is for SQLite */
Class.forName("org.sqlite.jdbc");
```

JDBC is an abstract API mostly composed of interfaces and abstract classes. Concrete implementations are mostly provided within drivers.

Class.forname() dynamically loads the driver's classes.

*throws *ClassNotFoundException!*

2. Establish a Connection (with URL)

- `DriverManager.getConnection(String url);`
- `DriverManager.getConnection(String url,`
  `String user,`
  `String password);`
- `DriverManager.getConnection(String url,`
  `Properties prop);`

2. Establish a Connection (with URL)

```
/* this is for MySQL*/
Connection c = DriverManager.getConnection(
“jdbc:mysql://localhost/dbname?user=user&password=pass”);

/* this is for SQLite */
Connection c = DriverManager.getConnection(
“jdbc:sqlite:filename.db”);
```

Establishes a connection to a database mediated by the Connection interface. The driver implements the Connection interface defined within JDBC

3. Create JDBC Statement(s)

```
Statement statement = c.createStatement() ;
```

The JDBC Statement, CallableStatement, and PreparedStatement interfaces define the methods and properties enabling developers to send SQL or PL/SQL commands and receive data from your database.

They also define methods helping bridge data type differences between Java and SQL data types used in a database.

3. Create JDBC Statement(s)

Interfaces	Recommended Use
Statement	Use the for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use the when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use the when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

4. Execute SQL Statements

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

- int executeUpdate (String SQL): Used for writing the database. Use this method to execute SQL statements such as INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE, etc. Returns the number of rows affected by the execution of the SQL statement.

- ResultSet executeQuery (String SQL): Used for reading the database. Use this method to execute SQL statements such as SELECT. Returns a ResultSet object.

4. Execute SQL Statements

```
statement.executeUpdate("CREATE TABLE person (" +
                        "_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
                        "cf VARCHAR(30) UNIQUE, " +
                        "name VARCHAR(30), " +
                        "surname VARCHAR(30))");

statement.executeUpdate("INSERT INTO person (cf, name,
surname) VALUES ('CHKPLH', 'Chuck', 'Palahniuk')");

statement.executeQuery("SELECT * FROM person");
```

The String passed to the statement.execute* methods depends on the specific SQL dialect used by the database. It is SQL, not Java!

5. Get ResultSet

- The java.sql.ResultSet interface represents the result set of a database query. Objects implementing the ResultSet interface maintain a cursor pointing to the current row in the result set.
  - Navigational methods: Used to move the cursor around the ResultSet
  - Get methods: Used to view the data in the columns of the current row being pointed by the cursor.
  - Update methods: Used to update the data in the columns of the current row. Updates are transparently written within the underlying database (if supported)

Navigational methods

Method	Description
`absolute()`	Moves the `ResultSet` to point at an absolute position. The position is a row number passed as parameter to the `absolute()` method.
`afterLast()`	Moves the `ResultSet` to point after the last row in the `ResultSet`.
`beforeFirst()`	Moves the `ResultSet` to point before the first row in the `ResultSet`.
`first()`	Moves the `ResultSet` to point at the first row in the `ResultSet`.
`last()`	Moves the `ResultSet` to point at the last row in the `ResultSet`.
`next()`	Moves the `ResultSet` to point at the next row in the `ResultSet`.
`previous()`	Moves the `ResultSet` to point at the previous row in the `ResultSet`.
`relative()`	Moves the `ResultSet` to point to a position relative to its current position. The relative position is passed as a parameter to the relative method, and can be both positive and negative.
	Moves the `ResultSet`

Method	Description
`getRow()`	Returns the row number of the current row - the row currently pointed to by the `ResultSet`.
`getType()`	Returns the `ResultSet` type.
`isAfterLast()`	Returns true if the `ResultSet` points after the last row. False if not.
`isBeforeFirst()`	Returns true if the `ResultSet` points before the first row. False if not.
`isFirst()`	Returns true if the `ResultSet` points at the first row. False if not.

Get methods

- resultSet.getXXX(), where XXX is a primitive data type. Columns can be selected via either name or id.
  - rs.getString("columnName")
  - rs.getLong("columnName")
  - rs.getInt("columnName")
  - rs.getDouble("columnName")

  - rs.getString(1)
  - rs.getLong(2)
  - rs.getInt(3)
  - rs.getDouble(4)

Get methods

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

while(rs.next()) {
   rs.getInt(“_id”);
   rs.getString(“cf”);
   rs.getString(“name”);
   rs.getString(“surname”);
}
```

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

while(rs.next()) {
   rs.getInt(1);
   rs.getString(2);
   rs.getString(3);
   rs.getString(4);
}
```

ING

Get methods

```
int idIndex = rs.findColumn("_id");
int cfIndex = rs.findColumn("cf");
int nameIndex = rs.findColumn("name");
int surnameIndex = rs.findColumn("surname");

while(rs.next()) {
    int id = rs.getInt(idIndex);
    String cf = rs.getString(cfIndex);
    String name = rs.getString(nameIndex);
    String surname = rs.getString(surnameIndex);
}
```

** If you need to access by column id but know only the column names*

6. Close Connection

```
statement.close();
connection.close();
```

However…

6. Close Connection

- Programs should recover from errors and always leave the database in a consistent state. Runtime errors must be minimized in industrial applications!
- If a statement throws an exception, it must be caught within a catch statement.
- The finally {…} clause can be used to leave the database in a consistent state.

6. Close Connection

```
connection = null;
statement = null;

try {
    . . .
} catch(SQLException e) {
    // do something
} finally {
    if (connection != null) {
        statement.close();
        connection.close();
    }
}
```

Types

- There are significant variations between the SQL types supported by different database products. For example, most of the major databases support an SQL data type for large binary values, but Oracle calls this type LONG RAW, Sybase calls it IMAGE and Informix calls it BYTE.
- JDBC programmers mostly program with existing database tables, and they need not concern themselves with the exact SQL type names that were used.
- The one major place where programmers may need to use SQL type names is in the SQL CREATE TABLE statement when they are creating a new database table. In this case programmers must take care to use SQL type names that are supported by their target database.

Mapping JDBC to Java types

Java Type	JDBC type
String	VARCHAR or LONGVARCHAR
java.math.BigDecimal	NUMERIC
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
byte[]	VARBINARY or LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP

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

ADVANCED RESULTSET

Advanced ResultSet

- ResultSet are iterator-like objects
- It is not possible to move back and forth within a default (`TYPE_FORWARD_ONLY`) ResultSet
  - Only `next()` can be called
- It is not possible to modify the data and, transparently, the database
  - Data have to be manipulated in memory and stored back with another operation (`statement.executeUpdate()`)

ING

Advanced ResultSet

createStatement

```
Statement createStatement(int resultSetType,
                          int resultSetConcurrency)
                   throws SQLException
```

Creates a Statement object that will generate ResultSet objects with the given type and concurrency. This method is the same as the createStatement method above, but it allows the default result set type and concurrency to be overridden. The holdability of the created result sets can be determined by calling getHoldability().

Parameters:

resultSetType - a result set type; one of ResultSet.TYPE_FORWARD_ONLY, ResultSet.TYPE_SCROLL_INSENSITIVE, or ResultSet.TYPE_SCROLL_SENSITIVE

resultSetConcurrency - a concurrency type; one of ResultSet.CONCUR_READ_ONLY or ResultSet.CONCUR_UPDATABLE

Returns:

a new Statement object that will generate ResultSet objects with the given type and concurrency

JDBC – Scrollable ResultSet

```
Statement s = c.createStatement(
   ResultSet.TYPE_SCROLL_INSENSITIVE|ResultSet.TYPE_SCROLL_SENSITIVE,
   ResultSet.CONCUR_READ_ONLY);

ResultSet rs = s.executeQuery("SELECT * FROM person");

rs.previous();      // go 1 record back
rs.relative(-5);    // go 5 records back
rs.relative(7);     // go 7 records forward
rs.absolute(100);   // go to 100th record
```

JDBC – Updateable ResultSet

```
Statement s = c.createStatement(
   ResultSet.TYPE_FORWARD_ONLY,
   ResultSet.CONCUR_UPDATABLE);

ResultSet rs = s.executeQuery("SELECT * FROM students
   WHERE type='listening'");

while (rs.next()) {
   int grade = rs.getInt("grade");
   rs.updateInt("grade", grade + 1);
   rs.updateRow();
}
```

DatabaseMetaData object

- A Connection object provides a DatabaseMetaData object which is able to provide schema information describing:
  - tables
  - supported SQL grammar
  - *supported capabilities of the connection*
  - stored procedures

** What is a stored procedure? A group of SQL statements forming a logical unit aimed at performing a specific task*

DatabaseMetaData object

```
// Establish Connection
Class.forName("org.sqlite.JDBC");
Connection connection = DriverManager.getConnection("jdbc:sqlite:sample.db");

// Get metadata
DatabaseMetaData md = connection.getMetaData();

// Verify ResultSet supported types
System.out.println("-- ResultSet Type --");
System.out.println("Supports TYPE_FORWARD_ONLY: ”
     + md.supportsResultSetType(ResultSet.TYPE_FORWARD_ONLY));
System.out.println("Supports TYPE_SCROLL_INSENSITIVE: ”
     + md.supportsResultSetType(ResultSet.TYPE_SCROLL_INSENSITIVE));
System.out.println("Supports TYPE_SCROLL_SENSITIVE: ”
     + md.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIVE));
```

ResultSetMetaData

- A ResultSet object provides a ResultSetMetaData object providing schema information
  - Useful for writing code running on different tables. For example, converting in JSON or XML the output of different queries.

```
public static void printRS(ResultSet rs) throws SQLException {
   ResultSetMetaData md = rs.getMetaData();
   // get number of columns
   int nCols = md.getColumnCount();
   // print column names
   for(int i=1; i < nCols; ++i)
    System.out.print(md.getColumnName(i)+",");
}
```

TRANSACTIONS

Definition

- A transaction is a set of actions to be carried out as a single, atomic action. Either all of the actions are carried out, or none of them are.

- The classic example of when transactions are necessary is the example of bank accounts. You need to transfer $100 from one account to the other. You do so by subtracting $100 from the first account, and adding $100 to the second account. If this process fails after you have subtracted the $100 from the first bank account, the $100 are never added to the second bank account. The money is lost in cyber space.

JDBC Transactions

- JDBC allows SQL statements to be grouped together into a single transaction
- Transaction control is performed by the Connection object, default mode is auto-commit, i.e., each sql statement is treated as a transaction
- We can turn off the auto-commit mode with connection.setAutoCommit(false);
- And turn it back on with connection.setAutoCommit(true);
- Once auto-commit is off, no SQL statement will be committed until an explicit is invoked connection.commit(). At this point all changes done by the SQL statements will be made permanent in the database.

JDBC Transactions

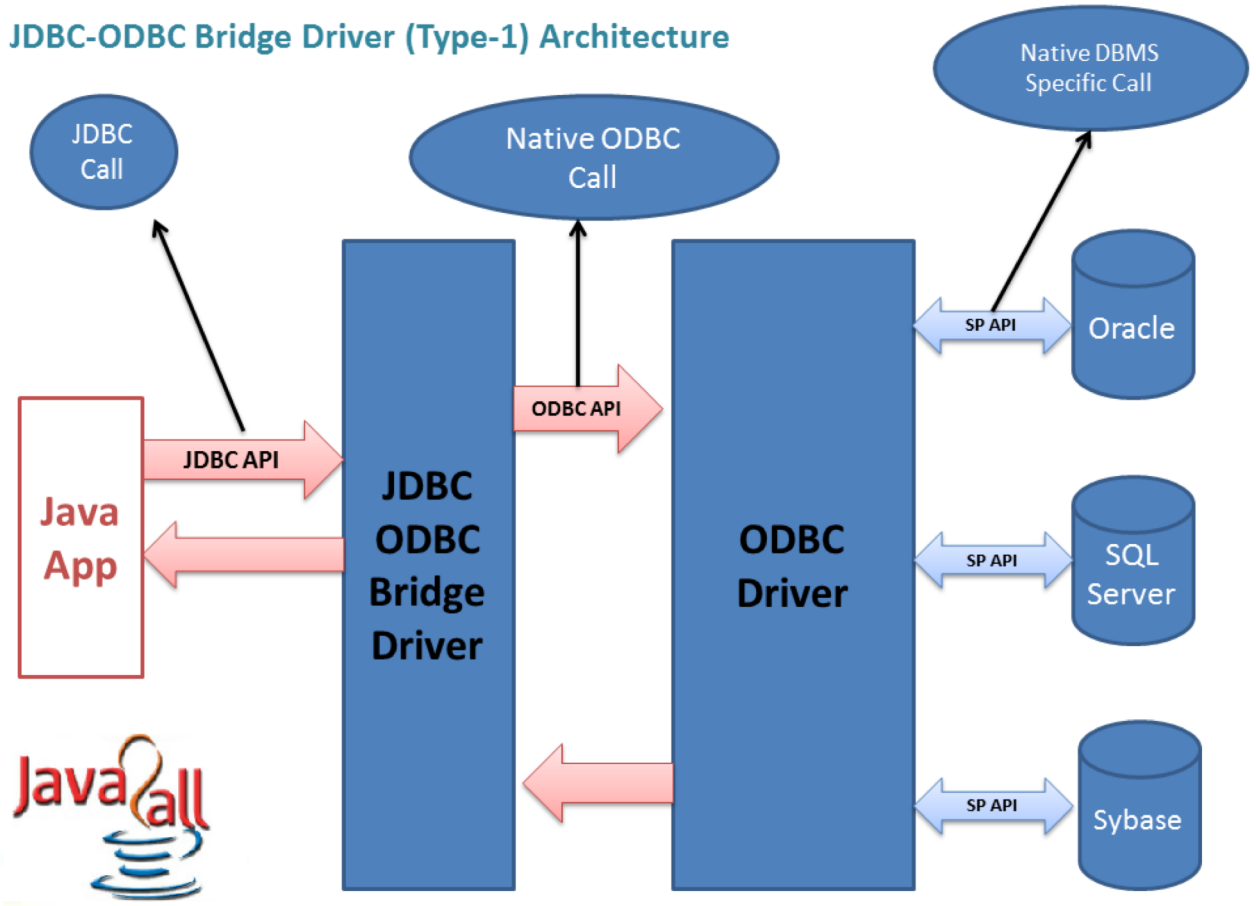
```
Connection connection = ...
try{
    connection.setAutoCommit(false);

    // create and execute statements etc.

    connection.commit();
} catch(Exception e) {
    connection.rollback();
} finally {
    if(connection != null) {
        connection.close();
    }
}
```

APPENDIX I: JDBC-ODBC

General Architecture

- What happens if I need to use a *rare* DBMS which is not supported by JDBC? (e.g., no driver released) ? Use ODBC!
- Open Database Connectivity (ODBC) is a standard application programming interface (API) for accessing database management systems (DBMS).
- Released in 1992, it allows applications to be independent from database-specific details. Same goal as JDBC, released in 1997.

JDBC-ODBC


JDBC-ODBC Bridge Driver (Type-1) Architecture
JDBC Call
Native ODBC Call
Native DBMS Specific Call
Java App
JDBC API
JDBC ODBC Bridge Driver
ODBC API
ODBC Driver
SP API
Oracle
SP API
SQL Server
SP API
Sybase

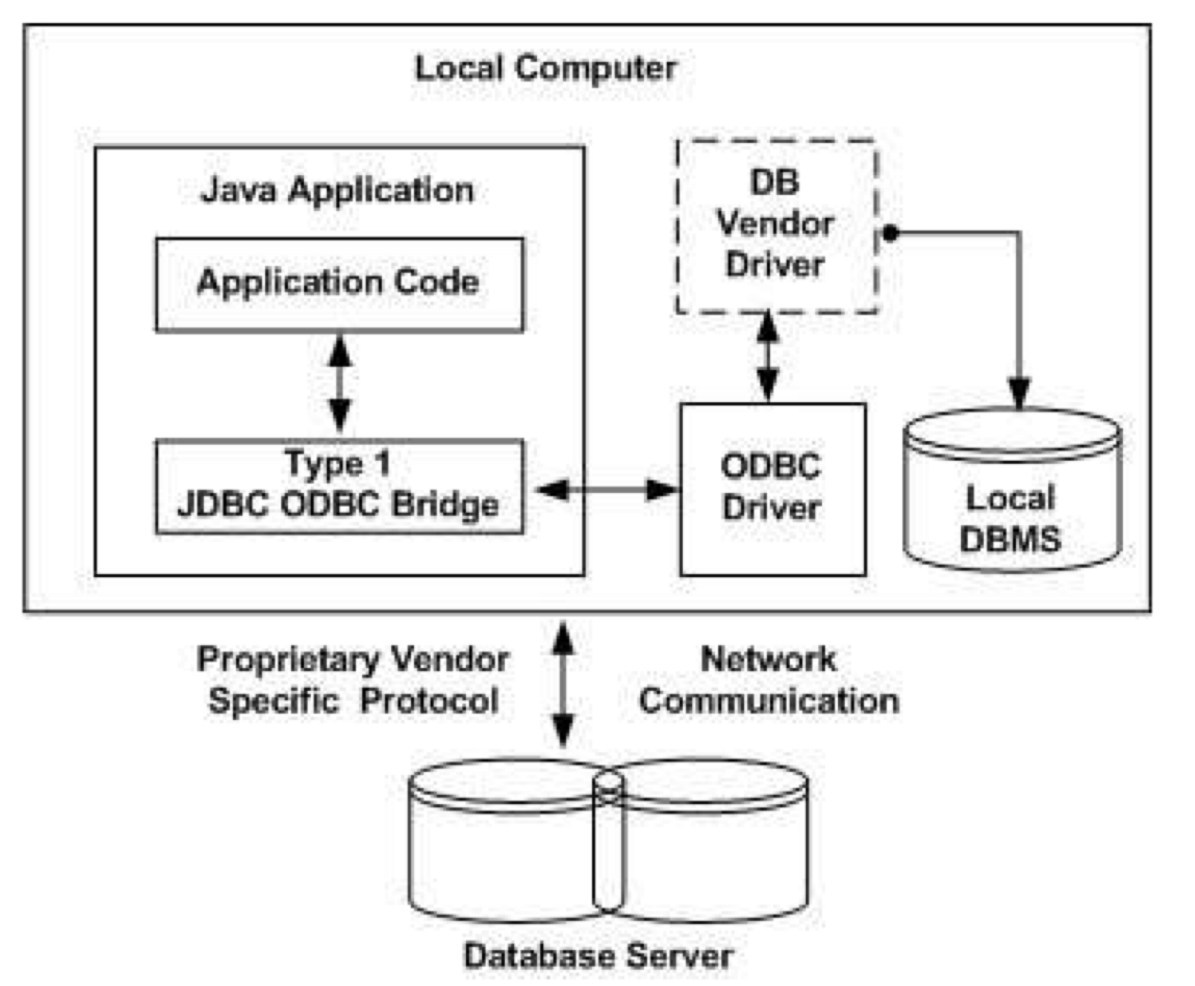
JDBC-ODBC

A JDBC-ODBC bridge consists of a JDBC driver which employs an ODBC driver to connect to a target database. This driver translates JDBC method calls into ODBC function calls. Programmers usually use such a bridge when a given database lacks a JDBC driver, but is accessible through an ODBC driver.

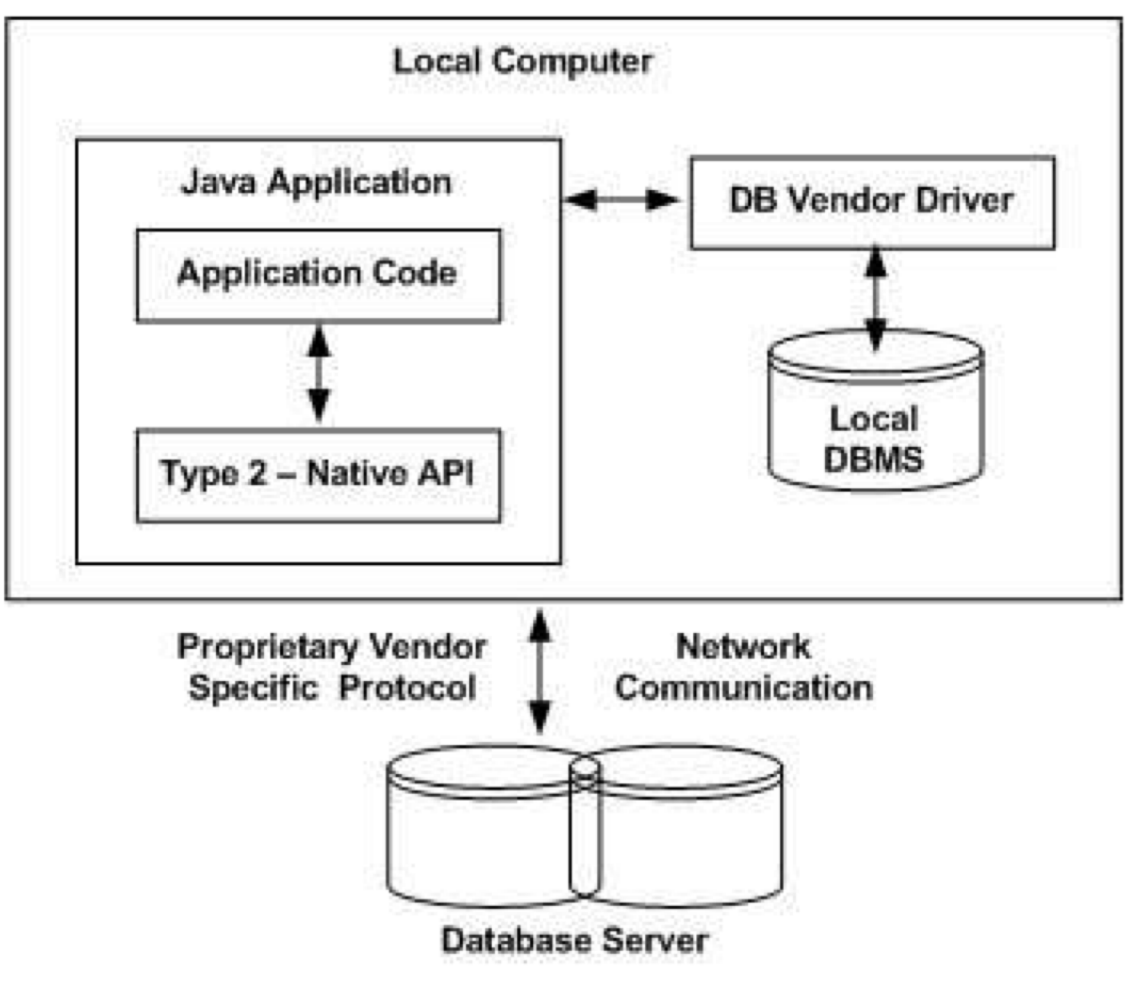
Vendors deliver JDBC-ODBC bridges which far outperform the JVM built-in (Removed from JVM since Java8).

APPENDIX II: DRIVER TYPES

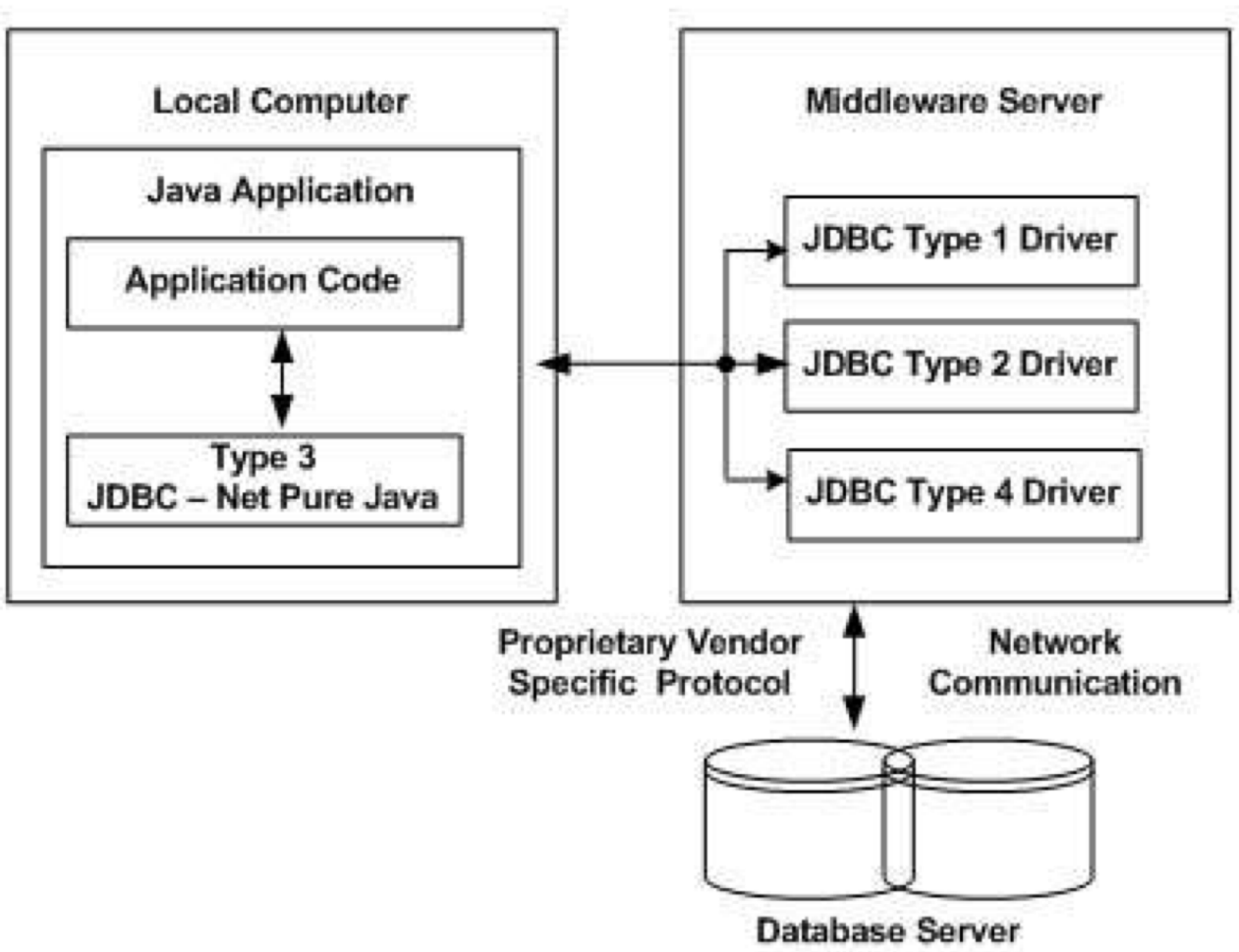
Type 1 JDBC Driver



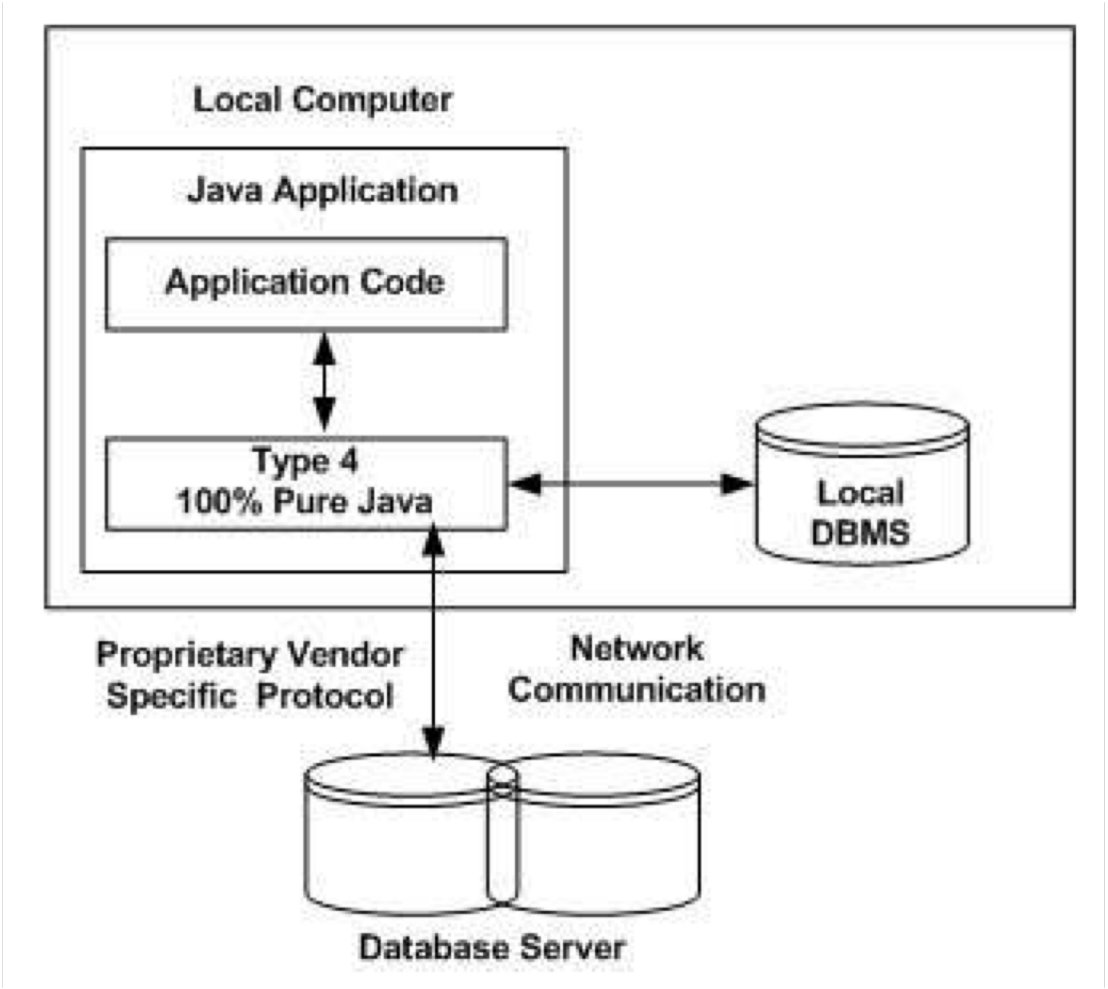
Type 2 JDBC Driver



Type 3 JDBC Driver



Type 4 JDBC Driver



References

- JDBC Data Access API – JDBC Technology Homepage
  - http://java.sun.com/products/jdbc/index.html
- JDBC Database Access – The Java Tutorial
  - http://java.sun.com/docs/books/tutorial/jdbc/index.html
- JDBC Documentation
  - http://java.sun.com/j2se/1.4.2/docs/guide/jdbc/index.html
- java.sql package
  - http://java.sun.com/j2se/1.4.2/docs/api/java/sql/package-summary.html
- JDBC Technology Guide: Getting Started
  - http://java.sun.com/j2se/1.4.2/docs/guide/jdbc/getstart/GettingStartedTOC.fm.html
- JDBC API Tutorial and Reference (book)
  - http://java.sun.com/docs/books/jdbc/