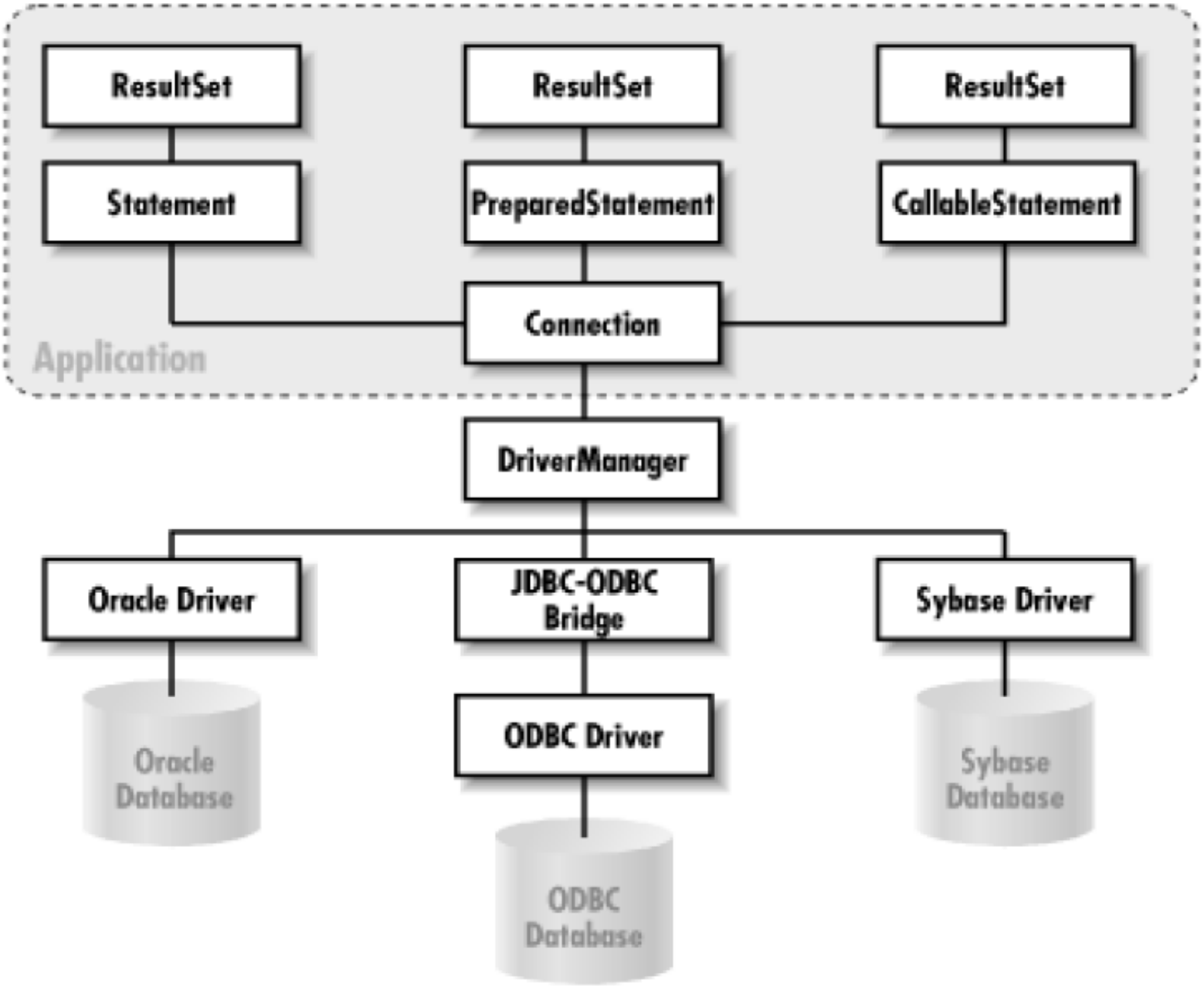
JDBC Architecture



JDBC Driver Types

JDBC driver specification classifies JDBC drivers into four groups. Each group is referred to as a JDBC driver type and addresses a specific need for communicating with various DBMSs. The JDBC driver types are as follows:

Type 1 JDBC-to-ODBC Driver

Microsoft was the first company to devise a way to create a DBMS-independent database program when they created the Open Database Connection (ODBC). ODBC is the basis from which Sun Microsystems, Inc. created JDBC. Both ODBC and JDBC have similar driver specifications and an API. The JDBC-to-ODBC driver, also called the JDBC/ODBC Bridge, is used to translate DBMS calls between the JDBC specification and the ODBC specification. The JDBC-to-ODBC driver receives messages from a J2EE component that conforms to the JDBC specification as discussed previously in this chapter. Those messages are translated by the JDBC-to-ODBC driver into the ODBC message format, which is then translated into the message format understood by the DBMS. However, avoid using the JDBC/ODBC Bridge in a mission-critical application because the extra translation might negatively impact performance.

Type 1 JDBC–ODBC Bridge Drivers

Type 1 drivers use a bridge technology to connect a Java client to an ODBC database system. The

JDBC–ODBC Bridge from Sun and InterSolv is the only extant example of a Type 1 driver. Type 1

drivers require some sort of non–Java software to be installed on the machine running your code, and

they are implemented using native code.

Type 2 Java/Native Code Driver

The Java/Native Code driver uses Java classes to generate platform-specific code—that is, code only understood by a specific DBMS. The manufacturer of the DBMS provides both the Java/Native Code driver and API classes so the J2EE component can generate the platform-specific code. The obvious disadvantage of using a Java/Native Code driver is the loss of some portability of code. The API classes for the Java/Native Code driver probably won't work with another manufacturer's DBMS.

Type 3 JDBC Driver

The Type 3 JDBC driver, also referred to as the Java Protocol, is the most commonly used JDBC driver. The Type 3 JDBC driver converts SQL queries into JDBC-formatted statements. The JDBC-formatted statements are translated into the format required by the DBMS.

Type 4 JDBC Driver

Type 4 JDBC driver is also known as the Type 4 database protocol. This driver is similar to the Type 3 JDBC driver except SQL queries are translated into the format required by the DBMS. SQL queries do not need to be converted to JDBC-formatted systems. This is the fastest way to communicate SQL queries to the DBMS.

A list of currently available JDBC drivers is available at http://java.sun.com/products/jdbc/jdbc.drivers.html.

When you are selecting a driver, you need to balance speed, reliability, and portability. Different applications have different needs.

Type 2 Native–API Partly Java Drivers

Type 2 drivers use a native code library to access a database, wrapping a thin layer of Java around the native library. Type 2 drivers are implemented with native code, so they may perform better than all–Java drivers, but they also add an element of risk, as a defect in the native code can crash the Java Virtual Machine.

JDBC Packages

The JDBC API is contained in two packages. The first package is called `java.sql` and contains core Java data objects of the JDBC API. These include Java data objects that provide the basics for connecting to the DBMS and interacting with data stored in the DBMS. `java.sql` is part of the J2SE.

The other package that contains the JDBC API is `javax.sql`, which extends `java.sql` and is in the J2EE. Included in the `javax.sql` package are Java data objects that interact with Java Naming and Directory Interface (JNDI) and Java data objects that manage connection pooling, among other advanced JDBC features.

A Brief Overview of the JDBC Process

Although each J2EE component is different, J2EE components use a similar process for interacting with a DBMS. This process is divided into five routines. These include:

- Loading the JDBC driver
- Connecting to the DBMS
- Creating and executing a statement
- Processing data returned by the DBMS
- Terminating the connection with the DBMS

Before you can use a driver, the driver must be registered with the JDBC DriverManager. This is typically done by loading the driver class using the Class.forName() method:

```
try {
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  Class.forName("com.oracle.jdbc.OracleDriver");
}
catch (ClassNotFoundException e) {
  /* Handle Exception */
}
```

Establishing a Connection

- DriverManager: This fully implemented class connects an application to a data source, which is specified by a database URL. When this class first attempts to establish a connection, it automatically loads any JDBC 4.0 drivers found within the class path.
- DataSource: This interface is preferred over DriverManager because it allows details about the underlying data source to be transparent to your application. A DataSource object's properties are set so that it represents a particular data source.

Using the DriverManager Class

Connecting to your DBMS with the DriverManager class involves calling the method DriverManager.getConnection.

```
public Connection getConnection() throws SQLException {
    Connection conn = null;
    Properties connectionProps = new Properties();
    connectionProps.put("user", this.userName);
    connectionProps.put("password", this.password);

    if (this.dbms.equals("mysql")) {
        conn = DriverManager.getConnection(
                "jdbc:" + this.dbms + "://" +
                this.serverName +
                ":" + this.portNumber + "/",
                connectionProps);
    } else if (this.dbms.equals("derby")) {
        conn = DriverManager.getConnection(
                "jdbc:" + this.dbms + ":" +
                this.dbName +
                ";create=true",
                connectionProps);
```

The java.sql.Connection object, which encapsulates a single connection to a particular database,

forms the basis of all JDBC data−handling code. An application can maintain multiple connections, up to the limits imposed by the database system itself. A standard small office or web server Oracle installation can support 50 or so connections, while a major corporate database could host several thousand. The

DriverManager.getConnection() method creates a connection:

```
Connection con = DriverManager.getConnection("url", "user", "password");
```

The getConnection() method has two other variants that are less frequently used. One variant takes a single String argument and tries to create a connection to that JDBC URL without a username or password.

The other version takes a JDBC URL and a java.util.Properties object that contains a set of

name/value pairs. You generally need to provide at least username=value and password=value pairs.

close() method.

This frees up any memory being used by the object, and it releases any other database resources the connection may be holding on to. (cursors, handles, and so on)

JDBC URLs

- jdbc:driver:databasename
- Oracle JDBC−Thin driver uses a URL of the form:

  jdbc:oracle:thin:@site:port:database
- JDBC−ODBC Bridge uses:

  jdbc:odbc:datasource;odbcoptions

```
public static void main(String[] args) {
  try {          // This is where we load the driver
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  }
  catch (ClassNotFoundException e) {
    System.out.println("Unable to load Driver Class");
    return; }
  try {          // All database access is within a try/catch block. Connect to database,
                 // specifying particular database, username, and password
    Connection con =
DriverManager.getConnection("jdbc:odbc:companydb", "", "");
     // Create and execute an SQL Statement
    Statement stmt = con.createStatement();
 ResultSet rs = stmt.executeQuery("SELECT FIRST_NAME FROM
EMPLOYEES");
```

```
// Display the SQL Results
    while(rs.next()) {
      System.out.println(rs.getString("FIRST_NAME"));
    }
    // Make sure our database resources are released
    rs.close();
    stmt.close();
    con.close();
  }
  catch (SQLException se) {
    // Inform user of any SQL errors
    System.out.println("SQL Exception: " + se.getMessage());
    se.printStackTrace(System.out);
  }
}
}
```

Statement

- A Statement is an interface that represents a SQL statement.
- execute Statement objects, and they generate ResultSet objects, which is a table of data representing a database result set.
- we need a Connection object to create a Statement object.

- Statement

Represents a basic SQL statement

- PreparedStatement

Represents a precompiled SQL statement, which can offer improved performance

- CallableStatement

Allows JDBC programs complete access to stored procedures within the database itself

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM CUSTOMERS");
```

- Statement also provides an **executeUpdate()** method, for running SQL statements that do not return results, such as the UPDATE and DELETE statements.
- executeUpdate() returns an integer that indicates the number of rows in the database that were altered.
- the **execute()** method of Statement.

This method returns true if there is a result associated with the statement.

```
Statement stmt = con.createStatement();
if(stmt.execute(sqlString)) {
  ResultSet rs = stmt.getResultSet();
  // display the results   }
else {
System.out.println("Rowsupdated:"+stmt.getUpdateCount());
}
```

It is important to remember that a Statement object represents a single SQL statement. A call to executeQuery(), executeUpdate(), or execute() implicitly closes any active ResultSet associated with the Statement.

```
Statement DataRequest;
ResultSet Results;
try {
        String query = "SELECT * FROM Customers";
        DataRequest = Database.createStatement();
        DataRequest = Db.createStatement();
        Results = DataRequest.executeQuery (query);
        DataRequest.close();
 }
```

Using Prepared Statements

If you want to **execute a Statement object many times,** it usually reduces execution time to use a PreparedStatement object instead.

Prepared Statements

- Allows you to precompile your SQL and run it repeatedly, adjusting specific parameters as necessary.
- PreparedStatement pstmt = con.prepareStatement("INSERT INTO EMPLOYEES (NAME, PHONE) VALUES (?, ?)");
- pstmt.clearParameters();
- pstmt.setString(1, "Jimmy Adelphi");
- pstmt.setString(2, "201 555−7823");
- pstmt.executeUpdate();

Table 2–1. SQL Data Types, Java Types, and Default getXXX( ) Methods

SQL Data Type	Java Type	GetXXX( ) Method
CHAR	String	getString()
VARCHAR	String	getString()
LONGVARCHAR	String	getString()
NUMERIC	java.math.BigDecimal	getBigDecimal()
DECIMAL	java.math.BigDecimal	getBigDecimal()
BIT	Boolean (boolean)	getBoolean()
TINYINT	Integer (byte)	getByte()
SMALLINT	Integer (short)	getShort()
INTEGER	Integer (int)	getInt()
BIGINT	Long (long)	getLong()
REAL	Float (float)	getFloat()
FLOAT	Double (double)	getDouble()
DOUBLE	Double (double)	getDouble()
BINARY	byte[]	getBytes()
VARBINARY	byte[]	getBytes()
LONGVARBINARY	byte[]	getBytes()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.Timestamp	getTimestamp()

CallableStatement

The CallableStatement interface is the JDBC object that supports stored procedures. The

Connection class has a prepareCall() method that is very similar to the prepareStatement()

method we used to create a PreparedStatement. Because each database has its own syntax for accessing

stored procedures, JDBC defines a standardized escape syntax for accessing stored procedures with

CallableStatement. The syntax for a stored procedure that does not return a result set is:

```
CallableStatment cstmt = con.prepareCall("{call sp_interest(?,?)}");
cstmt.registerOutParameter(2, Types.FLOAT);
cstmt.setInt(1, accountID);
cstmt.setFloat(2, 2343.23);
cstmt.execute();
out.println("New Balance:" + cstmt.getFloat(2));
```

Resultset

- Scrollable ResultSet

Stmt=DB.createStatement(TYPE_SCROLL_INSENSITIVE)

TYPE_SCROLL_INSENSITIVE
TYPE_SCROLL_SENSITIVE
TYPE_FORWARD_ONLY
first()
last()
previouse()
absolute()
relative()
getRow()

Updatable Resultset

CONCUR_UPDATABLE

CONCUR_READ_ONLYSTMT=Db.createStatement(ResultSet. CONCUR_UPDATABLE)

Multiple Result Sets

- It is possible to write a SQL statement that returns more than one ResultSet or update count .The Statement object supports this functionality via the getMoreResults() method.
- Calling this method implicitly closes any existing ResultSet and moves to the next set of results for the statement.
- getMoreResults() returns true if there is another ResultSet available to be retrieved by getResultSet(). However, the method returns false if the next statement is an update, even if there is another set of results waiting farther down the line. To be sure you've processed all the results for a Statement, you need to check that getMoreResults() returns false and that getUpdateCount() returns −1.

getMoreResults()

SQL statement that returns more than one ResultSet or update count. Calling this method implicitly closes any existing ResultSet and moves to the next set of results for the statement. getMoreResults() **returns true** if there is another ResultSet available to be retrieved by getResultSet(). However, the method returns false if the next statement is an update, even if there is another set of results waiting farther down the line. To be sure

you've processed all the results for a Statement, you need to check that getMoreResults() returns false and that getUpdateCount() returns −1.

```
Statement unknownSQL = con.createStatement();
unknownSQL.execute(sqlString);
while (true) {
  rs = unknownSQL.getResultSet();
  if(rs != null)
    // display the results
  else
    // process the update data
 // Advance and quit if done
  if((unknownSQL.getMoreResults() == false) &&
     (unknownSQL.getUpdateCount() == −1))
    break;
}
```