Java™ Education & Technology Services

Database Programming using Java

Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

Course Content

Java™ Education
and Technology Services

Invest In Yourself,
Develop Your Career

Course outline

- Lesson 1: Introduction to JDBC
- Lesson 2: Processing SQL Statements with JDBC
- Lesson 3: Connecting With DataSource Objects
- Lesson 4: Prepared Statement, Transaction, and Batch Update
- Lesson 5: Dealing with RowSet Objects

Lesson 1

Introduction To Java Database Connectivity (JDBC)

Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

1.1 What is the JDBC?

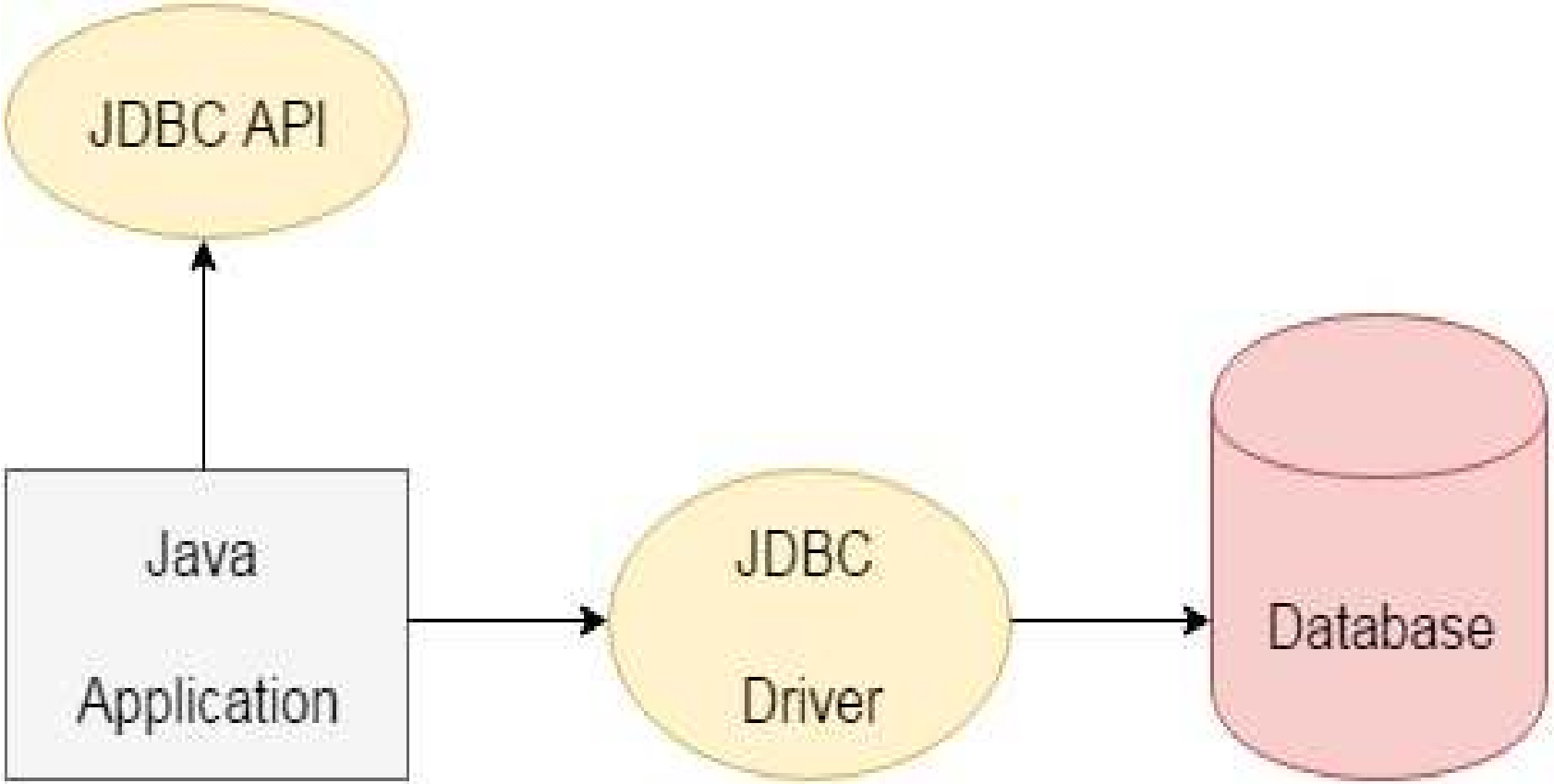
- In order to connect to a database from a Java application, you'll need an API (Application Programming Interface), which is a set of standard code that allows you to interact with another part of a system.
- One of the most common APIs is JDBC, which stands for Java Database Connectivity. JDBC is used by Java applications to connect to databases. It works with many database types such as Oracle, SQL Server, MySQL, Microsoft Access, SQLite, and PostgreSQL.
- JDBC is written in Java code and it is included in both Java SE (Standard Edition) and Java EE (Enterprise Edition).
- JDBC is a Java API for connecting to any DBMS and executing SQL statements.

(Hides DB specific details from application)

1.1 What is the JDBC?

- It consists of a set of classes and interfaces written in Java language.
- The JDBC interfaces are usually implemented by DBMS Vendors in order to provide their own vendor-specific JDBC Drivers.
- JDBC API uses JDBC drivers to connect with the database.
- We can use JDBC API to access tabular data stored into any relational database.

1.1 What is the JDBC?



1.2 Why Should We Use JDBC?

- Before JDBC, ODBC API was the database API to connect and execute query with the database. But, ODBC API was written in C,C++, Python, Core Java and these languages (except Java and some part of Python) are platform dependent. Therefore to remove dependence, JDBC was developed by database vendors which consisted of classes and interfaces written in Java.
- That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).
- We can use JDBC API to handle database using Java program and can perform following activities:
  - Connect to the database
  - Execute queries and update statements to the database
  - Retrieve the result received from the database(if available).

1.3 JDBC Product Components

- JDBC includes four components:
  - The JDBC API
  - JDBC Driver Manager
  - JDBC Test Suite
  - JDBC-ODBC Bridge

1.3.1 The JDBC API

- The JDBC™ API provides programmatic *access* to relational data from the Java™ programming language.
- Using the JDBC API, applications can *execute* SQL statements, *retrieve* results, and *propagate* changes back to an underlying data source.
- The JDBC API can also *interact* with multiple data sources in a distributed, *heterogeneous environment*.
- The JDBC API is part of the Java platform, which includes the Java™ SE and the Java™ EE. The JDBC 4.0 API is divided into two packages:
  - **`java.sql`** and
  - **`javax.sql`**.
- Both packages are included in the Java SE and Java EE platforms.

1.3.2 JDBC Driver Manager

- The JDBC **`DriverManager`** class defines objects which can connect Java applications to a JDBC driver.
- **`DriverManager`** has traditionally been the backbone of the JDBC architecture. It is quite small and simple.
- The Standard Extension packages **`javax.naming`** and **`javax.sql`** let you use a **`DataSource`** object registered with a *Java Naming and Directory Interface*™ (JNDI) naming service to establish a connection with a data source.
- You can use either connecting mechanism, but using a **`DataSource`** object is recommended whenever possible.

1.3.3 JDBC Test Suite

- The JDBC driver test suite helps you to determine that JDBC drivers will run your program. These tests are not comprehensive or exhaustive, but they do exercise many of the important features in the JDBC API.
- This test suite is intended to assist driver vendors in preparing their drivers for certification in a J2EE compatible configuration using the J2EE Compatibility Test Suite.
- For more details of installing the JDBC Test suit see the following link:

http://www.oracle.com/technetwork/java/jdbctestsuite-1-2-1-137683.html

1.3.4 JDBC-ODBC Bridge

- The Java Software bridge provides JDBC access via ODBC drivers. Note that you need to load ODBC binary code onto each client machine that uses this driver. As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

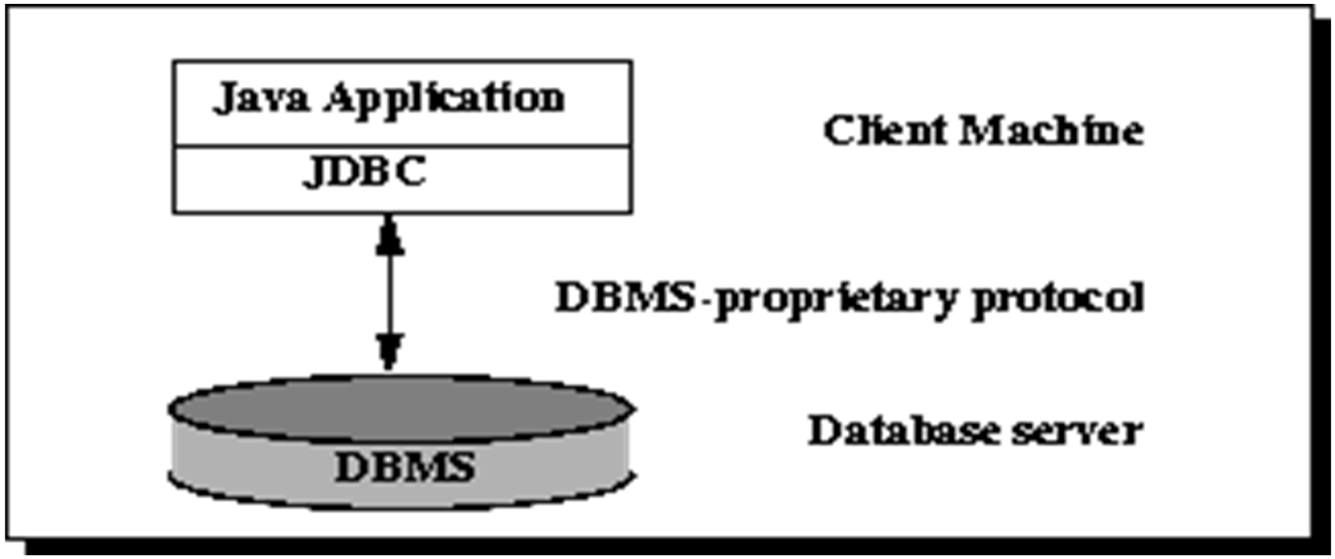
- This Trail uses the first two of these four JDBC components to connect to a database and then build a java program that uses SQL commands to communicate with a test Relational Database.
- The last two components are used in specialized environments to test web applications, or to communicate with ODBC-aware DBMSs.

1.4 JDBC Architecture

- The JDBC API supports both two-tier and three-tier processing models for database access.

1.4.1 Two-tier processing models for database access:

- In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed.

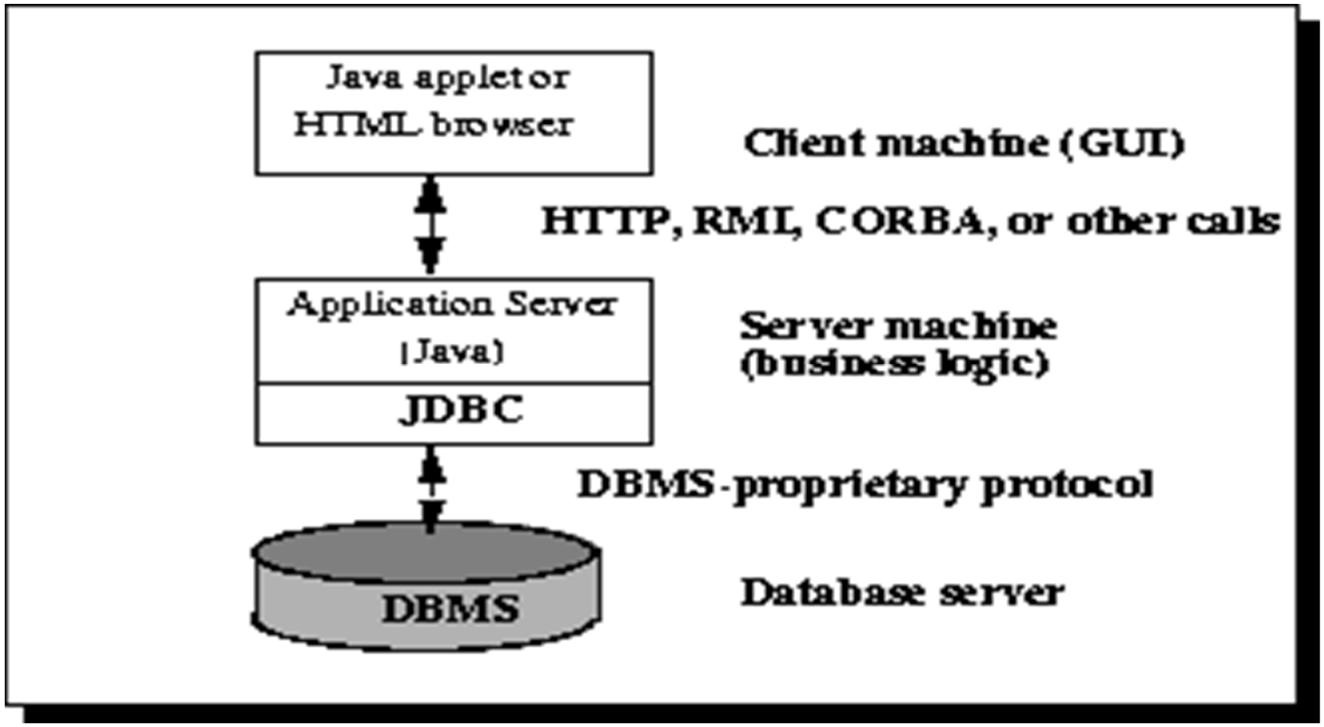


- A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user.

1.4.1 Two-tier processing models for database access:

- The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server.
- The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

1.4.2 Three-tier processing models for database access:



- In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user.

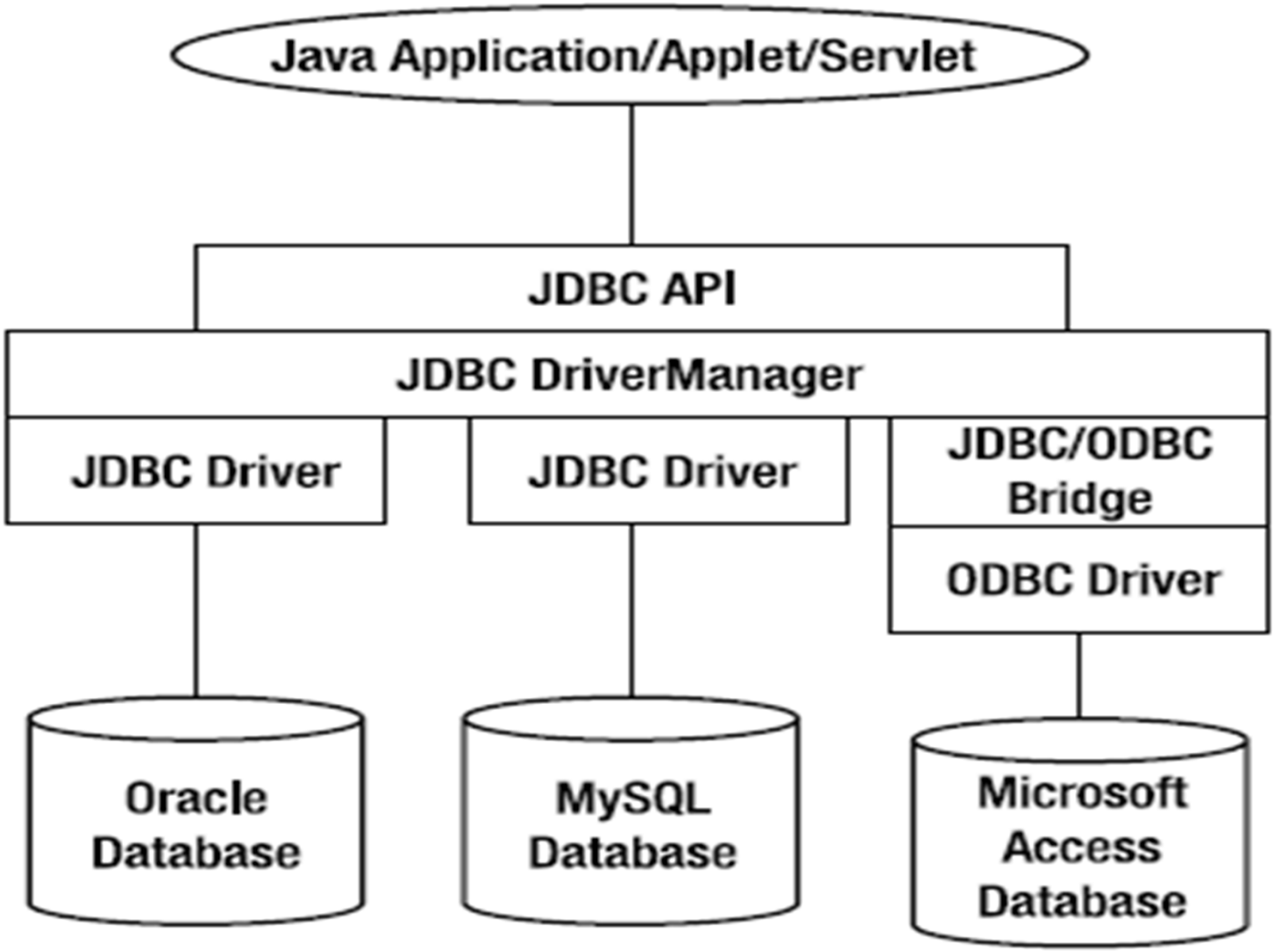
1.4.2 Three-tier processing models for database access:

- MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data.
- Another advantage is that it simplifies the deployment of applications.
- Finally, in many cases, the three-tier architecture can provide performance advantages.
- With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture.
- Some of the features that make JDBC a server technology are its support for *connection pooling*, *distributed transactions*, and *disconnected RowSets*.

1.5 JDBC Driver

- **1.5.1 What is JDBC Driver?**
  - JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server.
  - For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.
  - The **`Java.sql`** package that ships with JDK, contains various classes with their behaviors defined and their actual implementations are done in third-party drivers.
  - Third party vendors implements the **`java.sql.Driver`** interface in their database driver.

1.5.1 What is JDBC Driver?



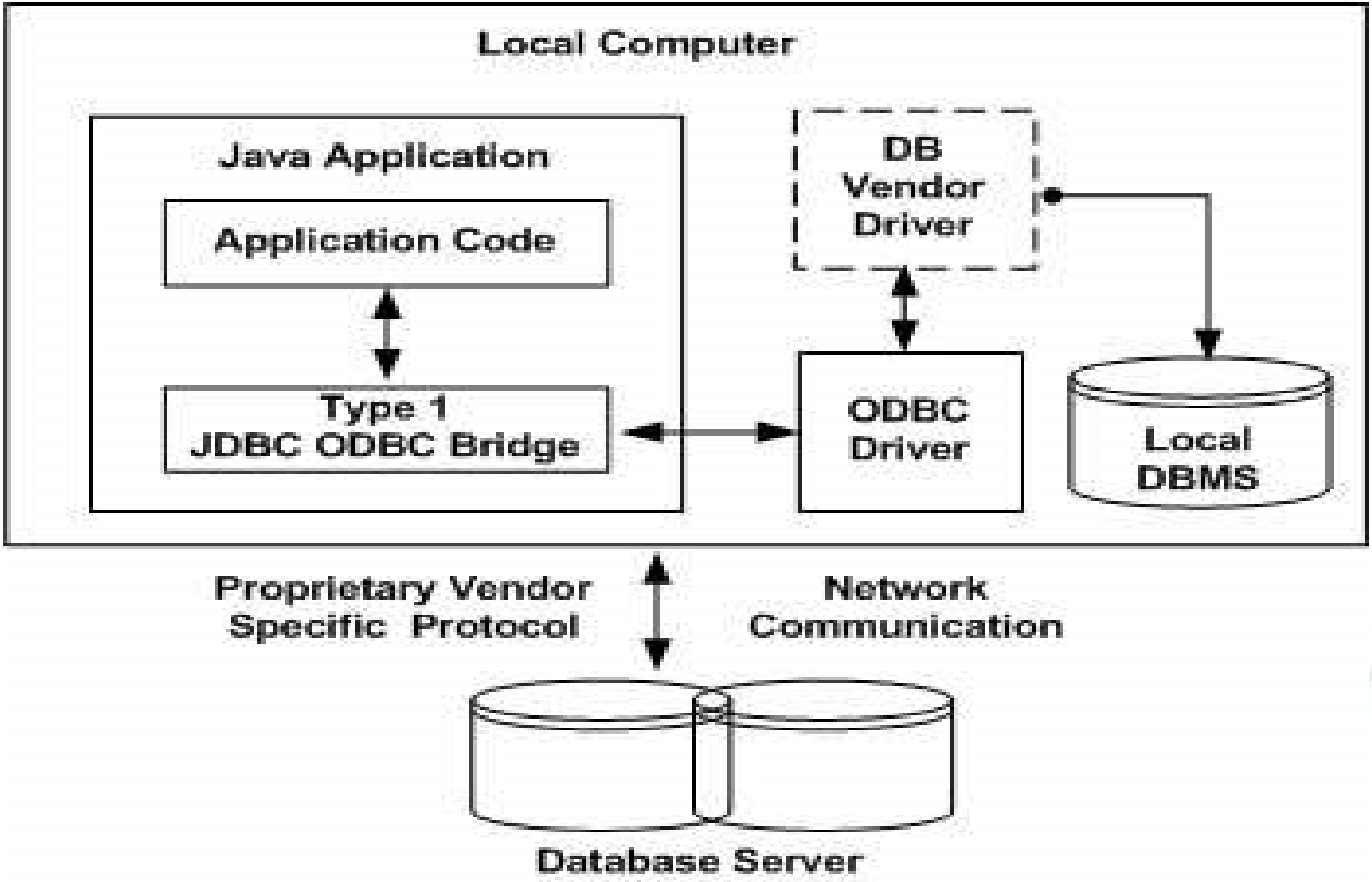
1.5.2 JDBC Drivers Types

- JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which are:
  - JDBC-ODBC bridge driver
  - Native-API driver (partially java driver)
  - Network Protocol driver (fully java driver)
  - Pure Java – Native Protocol Driver (Thin driver) (fully java driver)

1.5.2.1 Type 1: JDBC-ODBC Bridge Driver

- In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. It translates all JDBC calls into ODBC calls and sends them to the ODBC driver.
- Classes are already available within JDK (before JDK 8).
- Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.
- Usually used for testing (at the early stages of building a system) or when there are no other alternative driver types.
- Rather slow in performance due to the overhead of several layers.

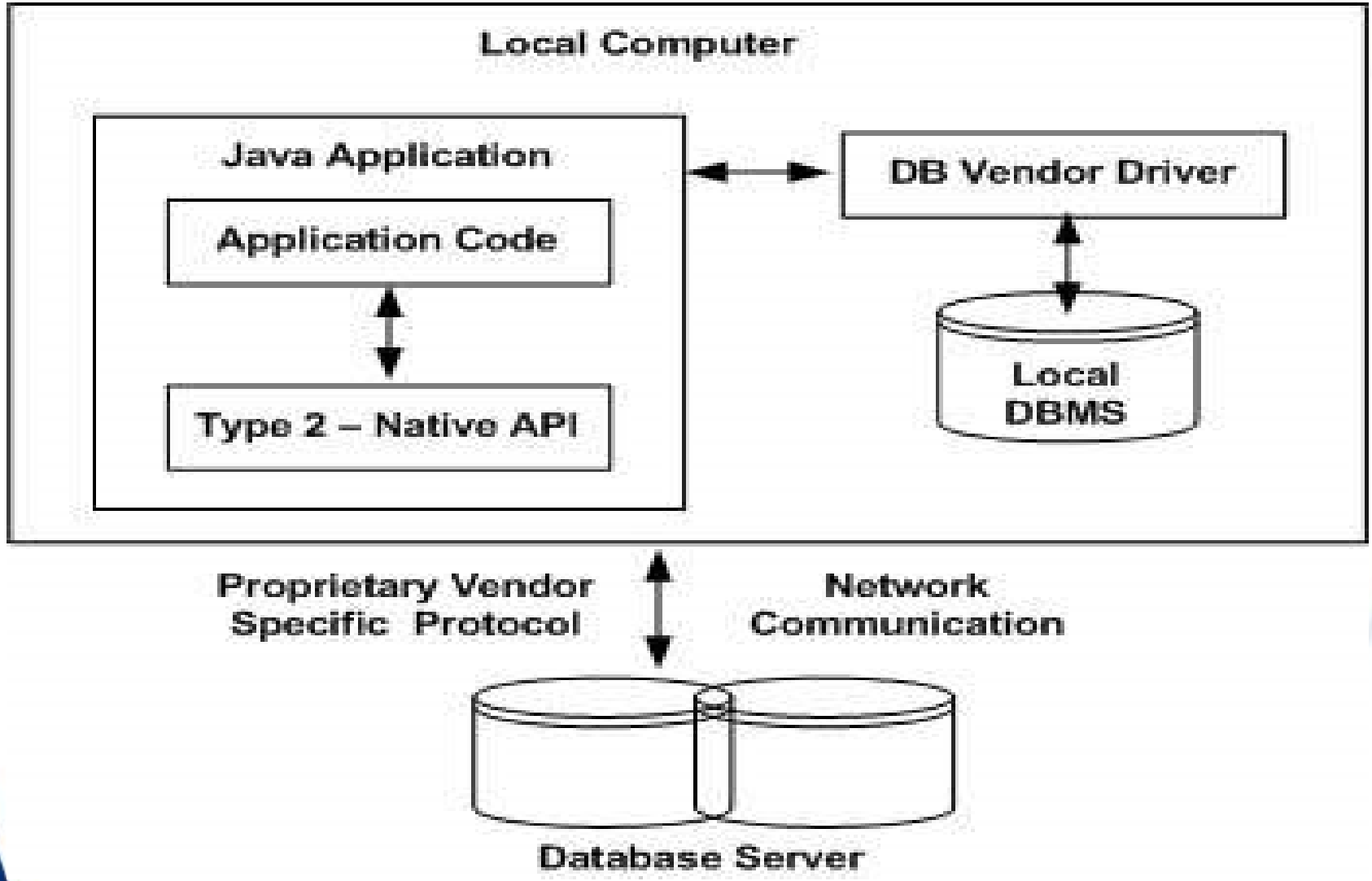
1.5.2.1 Type 1: JDBC-ODBC Bridge Driver



1.5.2.2 Type 2: Native-API driver (partially java driver)

- In a Type 2 driver, JDBC API calls are converted into native C/C++ 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.
- The vendor-specific driver must be installed on each client machine.
- If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.
- The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

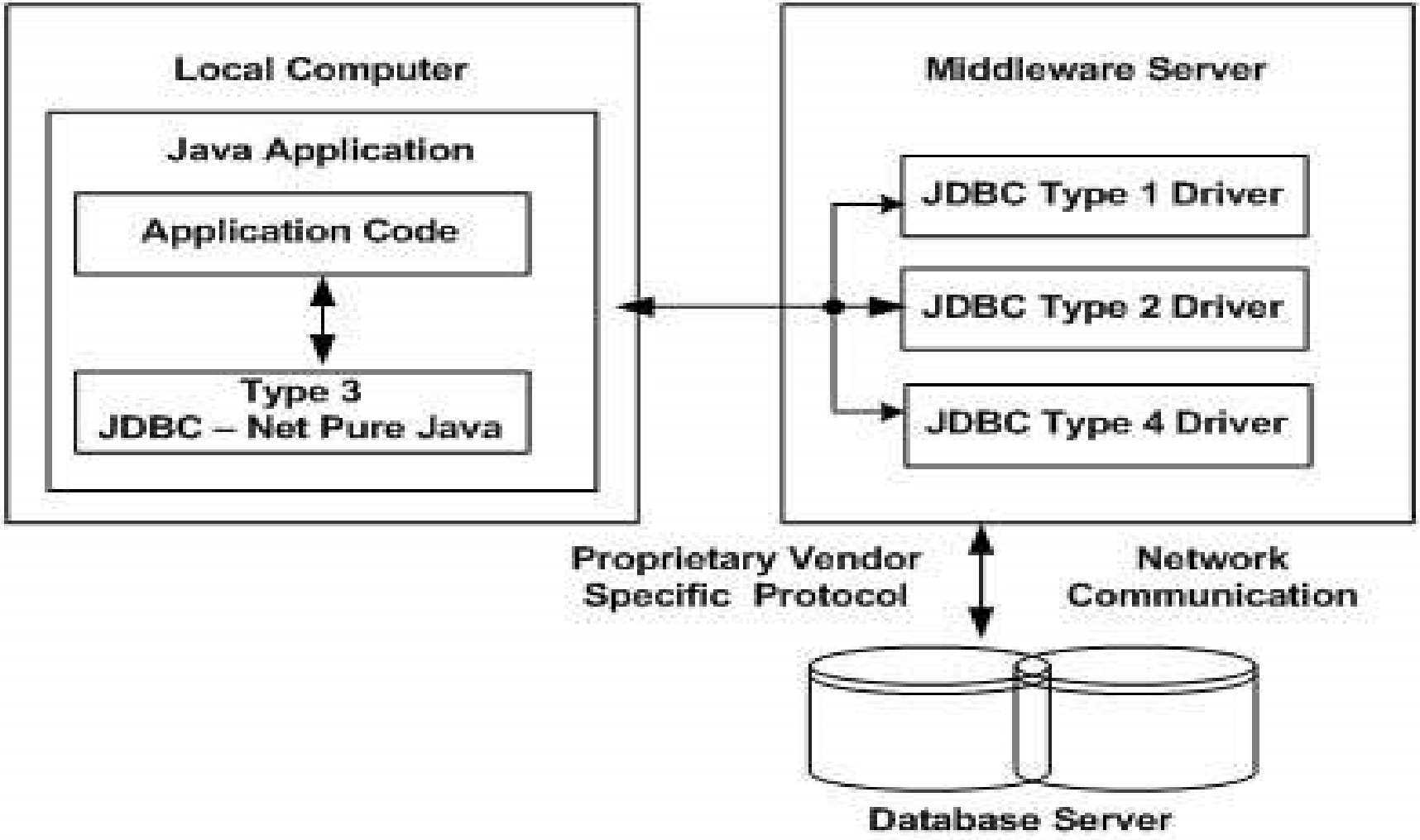
1.5.2.2 Type 2: Native-API driver (partially java driver)



1.5.2.3 Type 3: JDBC-Net Protocol driver (fully java driver)

- In a Type 3 driver, a three-tier approach is used to access databases.
- 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, and forwarded to the database server.
- This kind of driver requires no code installed on the client and a single driver can actually provide access to multiple databases.
- You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.
- Your application server might use a Type 1, 2, or 4 driver to communicate with the database.

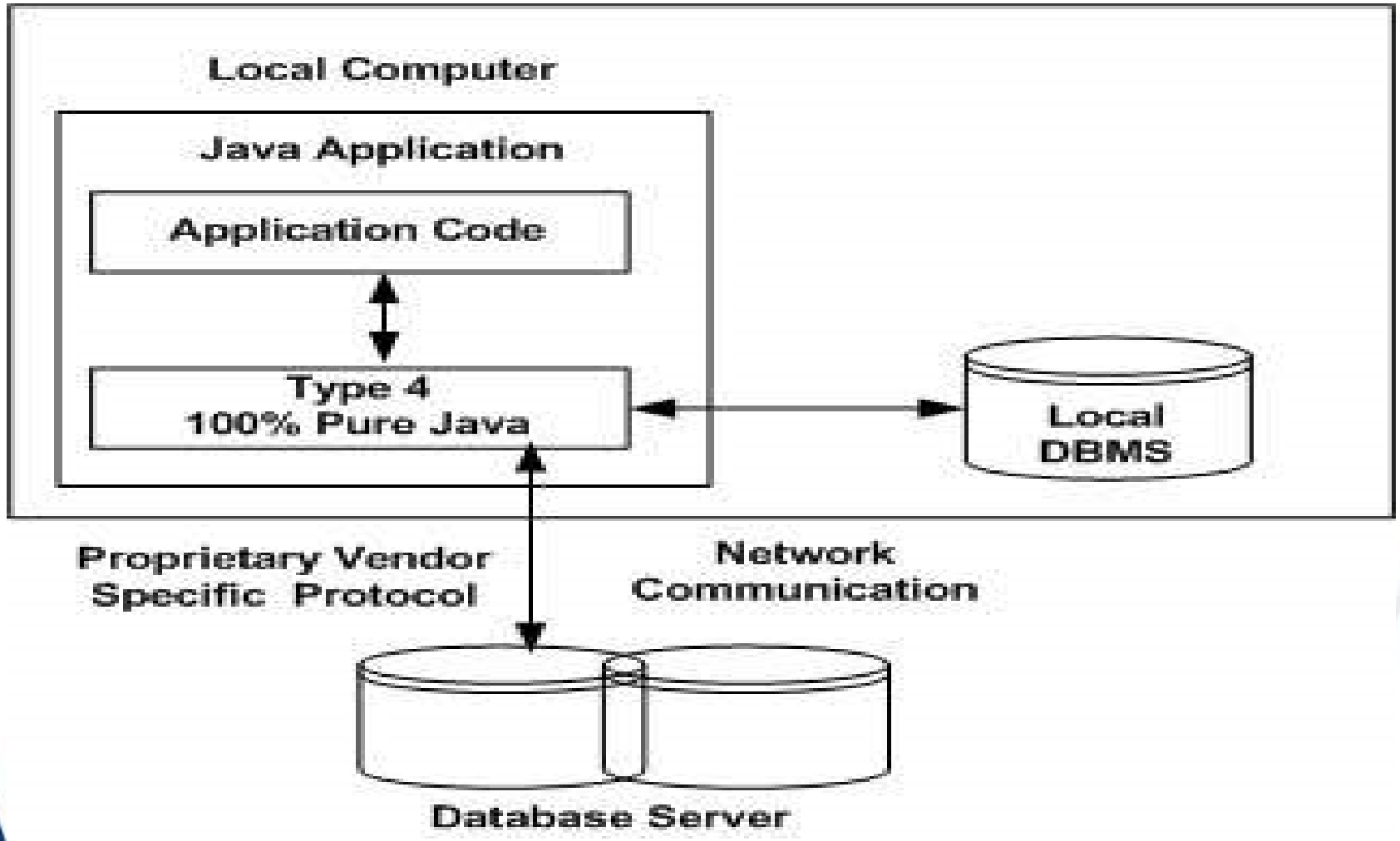
1.5.2.3 Type 3: JDBC-Net Protocol driver (fully java driver)



1.5.2.4 Type 4: Pure Java – Native Protocol Driver (fully java driver)

- In a Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through socket connection.
- This is the highest performance driver available for the database and is usually provided by the vendor itself.
- This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.
- Oracle thin Driver and MySQL's Connector/J driver are examples for type 4.

1.5.2.4 Type 4: Pure Java – Native Protocol Driver (fully java driver)



1.5.3 Which Driver should be Used?

- If you are accessing one type of database, such as Oracle, Sybase, or IBM DB2, 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.
- **Note**: Before JDK 8, there was implementation for type1 built-in JDK, starting from JDK 8 you need to find an implementation for it.

1.6 JDBC Versions History

- **JDBC Versions Released:**
  1. The JDBC 1.0 API.
  2. The JDBC 1.2 API.
  3. The JDBC 2.0 Optional Package API.
  4. The JDBC 2.1 core API.
  5. The JDBC 3.0 API.
  6. The JDBC 4.0 API.
  7. The JDBC 4.1 API.
  8. The JDBC 4.2 API.

1.6.1 Features of JDBC 1.0 API

- The JDBC 1.0 API was the first officially JDBC API launched consists of the java classes and interfaces that you can open connections to particular databases.
- This version includes a completely redesigned administration console with an enhanced graphical interface to manage and monitor distributed virtual databases.

1.6.2 Features of JDBC 1.2 API

- It supports Updatabale `ResultSets`.
- The `DatabaseMetaData` code has been refactored to provide more transparency with regard to the underlying database engine.
- New pass through schedulers for increased performance.

1.6.3 Features of The JDBC 2.0 Optional Package API

- The use of `DataSource` interface for making a connection.
- Use of JNDI to specify and obtain database connections.
- It allows us to use Pooled connections, that is we can reuse the connections.
- In this version the distributed transactions is possible.
- It provides a way of handling and passing data using `Rowset` technology.
- Became released in JDK 1.2.

1.6.4 Features of the JDBC 2.1 core API

- These features are present only in a JDK 1.2 or higher environment.
- Scroll forward and backward in a result set or has the ability to move to a specific row.
- Instead of using SQL commands, we can make updates to a database tables using methods in the Java programming language
- We can use multiple SQL statements in a database as a unit, or batch.
- It uses the SQL3 datatypes as column values. SQL3 types are Blob, Clob, Array, Structured type, Ref.

1.6.4 Features of the JDBC 2.1 core API

- Increased support for storing persistent objects in the java programming language.
- Supports for time zones in `Date`, `Time`, and `Timestamp` values.
- Full precision for `java.math.BigDecimal` values.

1.6.5 Features of JDBC 3.0 API

- These features are present only in a JDK 4 or higher environment.
- Reusability of prepared statements by connection pools.
- In this version there is number of properties defined for the `ConnectionPoolDataSource`. These properties can be used to describe how the `PooledConnection` objects created by DataSource objects should be pooled.
- A new concept has been added to this API is of save points.
- Retrieval of parameter metadata.
- These features are present only in a JDK 6 or higher environment.
- It has added a means of retrieving values from columns containing automatically generated values.

1.6.5 Features of JDBC 3.0 API

- Added a new data type i.e. `java.sql.BOOLEAN`.
- Passing parameters to `CallableStatement`.
- The data in the `Blob` and `Clob` can be altered.
- `DatabaseMetaData` API has been added.
- Auto- loading of JDBC driver class.
- Connection management enhancements.

1.6.6 Features of JDBC 4.0

- Support for `RowId` SAL type.
- SQL exception handling enhancements.
- `DataSet` implementation of SQL using Annotations.
- SQL XML support

1.6.7 Features of JDBC 4.1

- JDBC 4.1 adds some functionality to the core API over version 4.0.
- `java.sql.Connection` interface: JDBC 4.1 features
- JDBC 4.1 features are present only in a JDK 7 or higher environment.

1.6.8 Features of JDBC 4.2

- It is part of Java SE 8.
- Addition of REF_CURSOR support.
- Addition of `java.sql.DriverAction` Interface
- Addition of the `java.sql.SQLType` Interface
- Addition of the `java.sql.JDBCType` Enum
- Add Support for large update counts
- Changes to the existing interfaces
- `Rowset` 1.2: Lists the enhancements for JDBC `RowSet`.

Lesson 2

Processing SQL Statements with JDBC

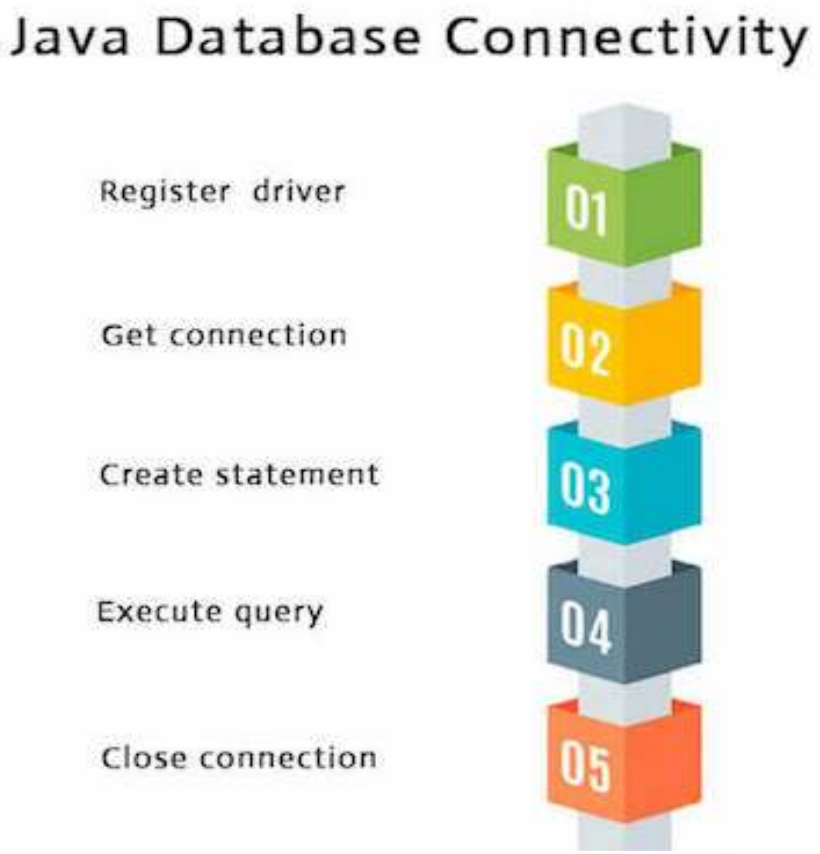
Java™ Education and Technology Services

Course Outlinesc

Invest In Yourself, Develop Your Career

Processing SQL Statements with JDBC

- There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:
  - Register the Driver class
  - Create connection
  - Create statement
  - Execute queries
  - Close connection



2.1Register the driver class (Loading the Driver)

- To begin with, you first need load the driver or register it before using it in the program.
- This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.
- Registration is to be done once in your program. You can register a driver in one of two ways mentioned below:
  - `Class.forName()`
  - `DriverManager.registerDriver()`

2.1.1 Class.forName()

- Here we load the driver's class file into memory at the runtime. No need of using new or creation of object .The following examples uses Class.forName() to load the Oracle driver and MySQL driver respectively:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

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

2.1.2 DriverManager.registerDriver()

- DriverManager is a Java inbuilt class with a static member register. Here we call the constructor of the driver class at compile time.
- The following examples uses `DriverManager.registerDriver()` to register the Oracle driver and MySQL driver respectively:

```
DriverManager.registerDriver(new
        oracle.jdbc.driver.OracleDriver());

DriverManager.registerDriver(new
        com.mysql.cj.jdbc.Driver());
```

2.2 Create the Connection

- After you've installed the appropriate driver, it is time to establish a database connection using JDBC.
- The programming involved to establish a JDBC connection is fairly simple. Here are these simple four steps:
  - **Import JDBC Packages:** Add import statements to your Java program to import required classes in your Java code.
  - **Register JDBC Driver:** This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests. (Illustrated in above slides).
  - **Database URL Formulation:** This is to create a properly formatted address that points to the database to which you wish to connect.
  - **Create Connection Object:** Finally, code a call to the DriverManager object's **`getConnection()`** method to establish actual database connection.

2.2.1 Import JDBC Packages

- To use the standard JDBC package, which allows you to select, insert, update, and delete data in SQL tables, add the following imports to your source code:

```
import java.sql.* ;  // for standard JDBC programs
import java.math.*; // for BigDecimal & BigInteger
```

2.2.2 Register JDBC Driver

- Registering the driver is the process by which the driver's class file is loaded into the memory, so it can be utilized as an implementation of the JDBC interfaces. (The two approaches are illustrated in previous slides in this lesson):
  - `Class.forName()`
  - `DriverManager.registerDriver()`

2.2.3 Database URL Formulation

- After you've loaded the driver, you can establish a connection using the **`DriverManager.getConnection()`** method.
- One of the form as follow:

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

  - **user** – username from which your sql command prompt can be accessed.
  - **password** – password from which your sql command prompt can be accessed.
  - **con**: is a reference to `Connection` interface.
  - **url** : *Uniform Resource Locator*. A database URL is an address that points to your database.
- At first we have to understand the meaning of database URL and see some examples of connection string.

2.2.3 Database URL Formulation

- Formulating a database URL is where most of the problems associated with establishing a connection occurs.
- The following table lists down the popular JDBC driver names and database URL:

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	**jdbc:mysql**://hostname:port/ 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:**hostname: port Number/databaseName
Postgresql	org.postgresql.Driver	**jdbc:postgresql:**//HOST/DATABASE

2.2.4 Create Connection Object

- For easy reference for `getConnection` method, let me list the three overloaded `DriverManager.getConnection()` methods:
  - **`getConnection(String url)`**
  - **`getConnection(String url, Properties prop)`**
  - **`getConnection(String url, String user, String password)`**

2.2.4.1 Using a URL with a username and password

- The most commonly used form of `getConnection()`, for example:

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
// String URL = "jdbc:mysql://localhost:3306/java";
String USER = "username";
String PASS = "password"
Connection conn=DriverManager.getConnection(URL, USER, PASS);
```

2.2.4.2 Using Only a Database URL

- In this case, the database URL includes the username and password and has the following general form (For Oracle driver):

```
jdbc:oracle:driver:username/password@database
jdbc:mysql://username:password@host:port/database
```

- The above example will be as follow:

```
String URL="jdbc:oracle:thin:username/password@amrood:1521:EMP";
//String URL= "jdbc:mysql://username:password@localhost:3306/java";
Connection conn = DriverManager.getConnection(URL);
```

2.2.4.3 Using a Database URL and a Properties Object

- A Properties object holds a set of keyword-value pairs. It is used to pass driver properties to the driver during calling the method.

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
//String URL = "jdbc:mysql://localhost:3306/java";
Properties info = new Properties( );
info.put( "user", "username" );
info.put( "password", "password" );
Connection conn = DriverManager.getConnection(URL, info);
```

2.3 Create a Statement

- Once a connection is established you can interact with the database.
- The **`Statement`**, **`CallableStatement`**, and **`PreparedStatement`** interfaces define the methods that enable you to send SQL commands and receive data from your database.
- The following methods are commonly used in interface `Connection`:
  - **`Statement createStatement ()`**
  - **`CallableStatement prepareCall()`**
  - **`PreparedStatement prepareStatement()`**

2.3.1 Statement

- There are 3 overloading from **`createStstement()`** method:
  - **`createStatement()`**: Creates a `Statement` object for sending SQL statements to the database.
  - **`createStatement(int resultSetType, int resultSetConcurrency)`**: Creates a Statement object that will generate `ResultSet` objects with the given type and concurrency.
  - **`createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability)`**: Creates a Statement object that will generate `ResultSet` objects with the given type, concurrency, and holdability.
- In `Statement` interface, we will se how to execute SQL queries in different ways.

2.3.1 Statement

- The `Statement` interface has methods which are used to execute SQL queries, the following are some examples:
  - `boolean execute(String sql)`: Executes the given SQL statement, which may return multiple results; ***true*** if the first result is a `ResultSet` object; ***false*** if it is an update count or there are no results
  - `int[] executeBatch()`: Submits a batch of commands to the database for execution and if all commands execute successfully, returns an array of update counts containing one element for each command in the batch.
  - `ResultSet executeQuery(String sql)`: Executes the given SQL statement, which returns a single ResultSet object.
  - `int executeUpdate(String sql)`: Executes the given SQL statement, which may be an INSERT, UPDATE, or DELETE statement or an SQL statement that returns nothing, such as an SQL DDL statement.

2.3.2 CallableStatement

- There are 3 overloading from **`prepareCall()`** method:
  - **`prepareCall(String sql)`**: Creates a `CallableStatement` object for calling database stored procedures.
  - **`prepareCall(String sql, int resultSetType, int resultSetConcurrency)`**: Creates a `CallableStatement` object that will generate `ResultSet` objects with the given type and concurrency.
  - **`prepareCall(String sql, int resultSetType, int resultSetConcurrency,int resultSetHoldability)`**: Creates a `CallableStatement` object that will generate `ResultSet` objects with the given type, concurrency, and holdability.

2.3.3 PreparedStatement

- There are many overloading from **`prepareStatement()`** method, the mostly common used are 3 as the follow:
  - **`prepareStatement(String sql)`**: Creates a `PreparedStatement` object for sending parameterized SQL statements to the database..
  - **`prepareStatement(String sql, int resultSetType, int resultSetConcurrency)`**: Creates a `PreparedStatement` object that will generate `ResultSet` objects with the given type and concurrency.
  - **`prepareStatement (String sql, int resultSetType, int resultSetConcurrency,int resultSetHoldability)`**: Creates a `PreparedStatement` object that will generate `ResultSet` objects with the given type, concurrency, and holdability.

2.3.4 Recommended using of statement types

Interfaces	Recommended Use
Statement	Use it 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 it when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use it when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

2.4 Deal with ResultSet

- The SQL statements that read data from a database query, return the data in a result set.
- The **`java.sql.ResultSet`** interface represents the result set of a database query.
- A **`ResultSet`** object maintains a cursor that points to the current row in the result set.
- The term "result set" refers to the row and column data contained in a **`ResultSet`** object.
- Characteristics of ResultSet [**Scrollable, Updatable and Holdable**]

```
ResultSet rs = pst.executeQuery() ;
```

2.4.1 ResultSet Types

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.

**** Default value***

2.4.2 Concurrency of ResultSet

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

**** Default value***

2.4.3 Holdability of ResultSet

Holdability	Description
Result Set. HOLD_CURSORS_OVER_COMMIT *	ResultSet object will remain open when the current transaction is committed.
ResultSet. CLOSE_CURSORS_AT_COMMIT	ResultSet object will be closed when the current transaction is committed

**** Default value***

2.4.4 ResultSet Methods

- The methods of the ResultSet interface can be broken down into *three* categories :
  - **Navigational methods**: Used to move the cursor around.
  - **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. The updates can then be updated in the underlying database as well.

2.4.5.1 Navigation Methods

Method	Description
next()	This method moves the cursor forward one row in the `ResultSet` from the current position. The method returns `true` if the cursor is positioned on a valid row and `false` otherwise.
previous()	The method moves the cursor backward one row in the `ResultSet`. The method returns `true` if the cursor is positioned on a valid row and `false` otherwise.
first()	The method moves the cursor to the first row in the `ResultSet`. The method returns `true` if the cursor is positioned on the first row and `false` if the ResultSet is empty.
last()	The method moves the cursor to the last row in the `ResultSet`. The method returns `true` if the cursor is positioned on the last row and `false` if the `ResultSet` is empty.
beforeFirst()	The method moves the cursor immediately before the first row in the `ResultSet`. There is no return value from this method.
afterLast()	The method moves the cursor immediately after the last row in the `ResultSet`. There is no return value from this method.

2.4.4.2 Get Methods

- If the column you are interested in viewing contains an **int**, you need to use one of the :

getInt(2) : by column Index.

getInt(" EMPID") : by column Name.

```
getInt( ), getLong( )     - get Integer field value
getFloat( ), getDouble() - get floating pt. value
getString( )   - get Char or Varchar field value
getDate( )     - get Date or Timestamp field value
getBoolean( ) - get a Bit field value
getBytes( )    - get Binary data
getBigDecimal( ) - get Decimal field as BigDecimal
getBlob( )     - get Binary Large Object
getObject( )   - get any field value
```

2.4.4.3 Update Methods

- If the column you are interested in updating is an **int**, you need to use one of the :

  `updateInt( 2, 253 )` : by column Index.

  `updateInt( " EMPID" , 253)` : by column Name

- After that you have to call the **`updateRow( )`** method which updates the underlying database with the new contents of the current row of this `ResultSet` object.
- There are number of updater methods like the number of the getter methods which are exist in the **`ResultSet`** interface.

2.4.4.3 Update Methods

- **cancelRowUpdates()** : Cancels the updates made to the current row in this ResultSet object.
- **deleteRow()** : Deletes the current row from this ResultSet object and from the underlying database.
- **moveToInsertRow()** : The insert row is a special row associated with an updatable result set. It is essentially a buffer where a new row may be constructed by calling the updater methods prior to inserting the row into the result set. Only the *updater*, *getter*, and *insertRow* methods may be called when the cursor is on the insert row.
- **insertRow()** : Inserts the contents of the insert row into this ResultSet object and into the database.

2.5 Execute Queries

- Query here is an SQL Query . Now we know we can have multiple types of queries. Some of them are as follows:
  - Query for updating / inserting table in a database.
  - Query for retrieving data.
- The `executeQuery()` method of Statement interface is used to execute queries of retrieving values from the database. This method returns the object of ResultSet that can be used to get all the records of a table.
- The `executeUpdate(String sqlQuery)` method of Statement interface is used to execute queries of updating/inserting

2.6 Close Connection

- So finally we have sent the data to the specified location and now we are at the verge of completion of our task .
- By closing connection, objects of **`Statement`** and **`ResultSet`** will be closed automatically. The **`close()`** method of **`Connection`** interface is used to close the connection.

2.7 Working with Statement example 1

Java™ Education and Technology Services

The following code sample is for connecting to an Oracle Database Server using Oracle thin Driver (Type 4):

Note: https://www.oracle.com/technetwork/database/application-development/jdbc/downloads/index.html

```
import java.sql.* ;
public class FirstDatabaseApp
{
  public FirstDatabaseApp()
  {
    try
    {
1         DriverManager.registerDriver(new
                  oracle.jdbc.driver.OracleDriver());

2         Connection con = DriverManager.getConnection
                  ("jdbc:oracle:thin:@127.0.0.1:1521:xe",
                                  "scott","tigger");
             Statement stmt = con.createStatement() ;
3   String queryString = new String("select * from tab");
```

2.7 Working with Statement example 1

The following code sample is for connecting to an MySQL Database Server using Type 4 Driver :

Note: Download from: https://dev.mysql.com/downloads/connector/j/8.0.html

```
import java.sql.* ;
public class FirstDatabaseApp
{
  public FirstDatabaseApp()
  {
    try
    {
1         DriverManager.registerDriver(new
                  com.mysql.cj.jdbc.Driver());

2         Connection con = DriverManager.getConnection
                  ("jdbc:mysql://localhost:3306/sakila",
                                    "root","passwd");
            Statement stmt = con.createStatement() ;
3   String queryString = new String("select * from tab");
```

2.7 Working with Statement example 1

Java™ Education and Technology Services

```
4       ResultSet rs = stmt.executeQuery(queryString) ;
        while(rs.next())
        {
             System.out.println(rs.getString(1));
        }
5       stmt.close();
        con.close();
        }
    catch(SQLException ex)
    {
          ex.printStackTrace();
    }
  }
  public static void main(String args[])
  {
     new FirstDatabaseApp();
  }
}
```

2.8 Working with Statement example 2

```
import java.sql.*;
import java.util.*;
class Main
{   public static void main(String a[])
    {
        //Creating the connection
        String url = "jdbc:oracle:thin:@localhost:1521:xe";
        String user = "system";
        String pass = "12345";
        //Entering the data
        Scanner k = new Scanner(System.in);
        System.out.println("enter name");
        String name = k.next();
        System.out.println("enter roll no");
        int roll = k.nextInt();
        System.out.println("enter class");
        String cls =  k.next();
```

2.8 Working with Statement example 2

```
      //Inserting data using SQL query
      String sql = "insert into student1
            values('"+name+"','"+roll+"','"+cls+"')";
       Connection con=null;
       try        {
  DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
      // You may use: Class.forName("oracle.jdbc.OracleDriver");
      // Reference to connection interface
          con = DriverManager.getConnection(url,user,pass);
          Statement st = con.createStatement();
          int m = st.executeUpdate(sql);
if (m == 1)  System.out.println("inserted successfully : "+sql);
else         System.out.println("insertion failed");
          con.close();
       }
       catch(SQLException ex) {  System.err.println(ex); }
    }
}
```

Lab Exercise

Java™ Education
and Technology Services

Invest In Yourself,
Develop Your Career

Assignments

- Install MySQL server and download its driver type 4 (Connector/j).
- Write a java application for connect to a database on MySQL Server and execute a query which selects numbers of fields from a table and print it with any specific format, using load a specific driver and get the connection using DriverManager class.

Lesson 3

Connecting With DataSource Objects

Invest In Yourself,
Develop Your Career

3.1 Overview about `DataSource`

- `DataSource` objects are the preferred means of getting a connection to a data source.
- One of its advantages, `DataSource` objects can provide connection pooling and distributed transactions. This functionality is essential for enterprise database computing.
- This lesson shows you how to get a connection using the `DataSource` interface.

3.2 Using DataSource Objects to Get a Connection

- Objects instantiated by classes that implement the `DataSource` represent a particular DBMS or some other data source, like a file.
- If a company uses more than one data source, it will deploy a separate `DataSource` object for each of them.
- The `DataSource` interface is implemented by a driver vendor. It can be implemented in three different ways:
  - A basic `DataSource` implementation produces standard `Connection` objects that are not pooled or used in a distributed transaction.
  - A `DataSource` implementation that supports connection pooling produces `Connection` objects that participate in connection pooling, that is, connections that can be recycled.
  - A `DataSource` implementation that supports distributed transactions produces `Connection` objects that can be used in a distributed transaction, that is, a transaction that accesses two or more DBMS servers.

3.2 Using DataSource Objects to Get a Connection

- A JDBC driver should include at least a basic `DataSource` implementation. For example, the Java DB JDBC driver includes the implementation `org.apache.derby.jdbc.ClientDataSource` , for Oracle, `oracle.jdbc.pool.OracleDataSource` , and for MySQL, `com.mysql.cj.jdbc.MysqlDataSource`.
- A `DataSource` class that supports distributed transactions typically also implements support for connection pooling.

3.3 Deploying Basic DataSource Objects

- The system administrator needs to deploy `DataSource` objects. Deploying a `DataSource` object consists of three tasks:
  - Creating an instance of the `DataSource` class
  - Setting its properties
  - Registering it with a naming service that uses the Java Naming and Directory Interface (JNDI) API.

3.3.1 Creating an instance of the DataSource class

- Suppose a company has bought a driver from the JDBC vendor DB Access, Inc. This driver includes the class `com.dbaccess.BasicDataSource` that implements the `DataSource` interface. The following code excerpt creates an instance of the class `BasicDataSource`:

```
com.dbaccess.BasicDataSource ds = new
    com.dbaccess.BasicDataSource();
```

3.3.2 Setting its properties

- After the instance of `BasicDataSource` is deployed, a programmer can call the method `DataSource.getConnection` to get a connection to the company's database, CUSTOMER_ACCOUNTS. First, the system administrator creates the `BasicDataSource` object **`ds`** using the default constructor. The system administrator then sets three properties. Note that the following code is typically be executed by a *deployment tool*:

```
ds.setServerName("grinder");
ds.setDatabaseName("CUSTOMER_ACCOUNTS");
ds.setDescription("Cust. accounts database for billing");
```

- The variable **`ds`** now represents the database CUSTOMER_ACCOUNTS installed on the server. Any connection produced by the `BasicDataSource` object **`ds`** will be a connection to the database CUSTOMER_ACCOUNTS.

3.3.3 Registering DataSource Object with Naming Service That Uses JNDI API

- With the properties set, the system administrator can register the `BasicDataSource` object with a JNDI (Java Naming and Directory Interface) naming service. The following code registers the `BasicDataSource` object and binds it with the logical name `jdbc/billingDB`:

```
Context ctx = new InitialContext();
ctx.bind("jdbc/billingDB", ds);
```

- The first line creates an `InitialContext` object, which serves as the starting point for a name, similar to root directory in a file system. The second line associates, or binds, the `BasicDataSource` object ds to the logical name `jdbc/billingDB`. In the next code, you give the naming service this logical name, and it returns the `BasicDataSource` object. The logical name can be any string. In this case, the company decided to use the name `billingDB` as the logical name for the CUSTOMER_ACCOUNTS database.

3.4 Using Deployed DataSource Object

- After a basic `DataSource` implementation is deployed by a system administrator, it is ready for a programmer to use. This means that a programmer can give the logical data source name that was bound to an instance of a `DataSource` class, and the JNDI naming service will return an instance of that `DataSource` class.
- The method `getConnection` can then be called on that `DataSource` object to get a connection to the data source it represents. For example, a programmer might write the following two lines of code to get a `DataSource` object that produces a connection to the database CUSTOMER_ACCOUNTS:

3.4 Using Deployed DataSource Object

```
Context ctx = new InitialContext();
DataSource s=(DataSource)ctx.lookup("jdbc/billingDB");
```

- The first line of code gets an initial context as the starting point for retrieving a `DataSource` object.
- When you supply the logical name `jdbc/billingDB` to the method lookup, the method returns the `DataSource` object that the system administrator bound to `jdbc/billingDB` at deployment time.
- Calling the method `ds.getConnection` produces a connection to the CUSTOMER_ACCOUNTS database.

```
Connection con = s.getConnection("fernanda","brewed");
```

3.5 Deploying Other DataSource Implementations

- As we illustrated before, The DataSource interface is implemented by a driver vendor. It can be implemented in different ways:
  - standard,
  - supports connection pooling, and
  - supports distributed transactions

We will illustrate the other two types.

3.5.1 Deploying `ConnectionPoolDataSource`

- A system administrator or another person working in that capacity can deploy a `DataSource` object so that the connections it produces are pooled connections.
- To do this, he or she first deploys a `ConnectionPoolDataSource` object and then deploys a `DataSource` object implemented to work with it.
- With the `ConnectionPoolDataSource` and `DataSource` objects deployed, you can call the method `DataSource.getConnection` on the `DataSource` object and get a pooled connection. This connection will be to the data source specified in the `ConnectionPoolDataSource` object's properties.

3.5.2 Deploying Distributed Transactions

- `DataSource` objects can be deployed to get connections that can be used in distributed transactions.
- As with connection pooling, two different class instances must be deployed: an `XADataSource` object and a `DataSource` object that is implemented to work with it.

3.6 Advantages of DataSource Objects

- Because of its properties, a `DataSource` object is a better alternative than the `DriverManager` class for getting a connection. Programmers no longer have to hard code the driver name or JDBC URL in their applications, which makes them more portable.
- Also, `DataSource` properties make maintaining code much simpler. If there is a change, the system administrator can update data source properties and not be concerned about changing every application that makes a connection to the data source.

3.7 DataSource Example1 :Using properties file

- Normally, Java properties file is used to store project configuration data or settings. We may write and read the properties file using Java code, or we may edit it like any text file and read it by Java code.
- The following code is a simple example to create a properties file called "*db.properties*" using Java program.

3.7.1 Example 1:Using properties file

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.util.Properties;
public class App {
  public static void main(String[] args) {
      Properties prop = new Properties();
      OutputStream output = null;
      try {
          output = new FileOutputStream("db.properties");
            // set the properties value
            prop.setProperty("MYSQL_DB_URL",
                  "jdbc:mysql://localhost:3306/sakila");
            prop.setProperty("MYSQL_DB_USERNAME", "root");
            prop.setProperty("MYSQL_DB_PASSWORD", "passwd");
```

3.7.1 Example 1:Using properties file

```
// save properties to project folder
            prop.store(output, null);
        } catch (IOException io) {
            io.printStackTrace();
        } finally {
            if (output != null) {
                try {
                    output.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

        }
    }
}
```

3.7.1 Example 1:Using properties file

- The previous code generate the following output file:

```
#Wed Dec 5 10:45:59 CAT 2018
MYSQL_DB_USERNAME=root
MYSQL_DB_URL=jdbc\:mysql\://localhost\:3306/sakila
MYSQL_DB_PASSWORD=passwd
```

- The following java class is using the previous file to get a DataSource object from MySQL database which will used to connect to MySQL database.

3.7.1 Example 1:Using properties file

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.SQLException;
import java.util.Properties;
import javax.sql.DataSource;

public class MyDataSourceFactory {
  public static DataSource getMySQLDataSource () {
      Properties props = new Properties();
      FileInputStream fis = null;
      MysqlDataSource mysqlDS = null;
      try {
          fis = new FileInputStream("db.properties");
          props.load(fis);
          mysqlDS = new MysqlDataSource();
            // get the properties value
```

3.7.1 Example 1:Using properties file

```
                // get the properties value
mysqlDS.setURL(prop.getProperty("MYSQL_DB_URL“));
mysqlDS.setUser(prop.getProperty("MYSQL_DB_USERNAME“));
mysqlDS.setPassword(prop.getProperty("MYSQL_DB_PASSWORD“));
        } catch (IOException e) {
                        e.printStackTrace();
        }
                return mysqlDS;
    }
}
```

3.7.1 Example 1:Using properties file

- The previous class is a factory class which generate a DataSource object which is describing a data source from MySQL database server.
- Any java application can use the static method to get the object and use it to connect to the database as it will be discussing in the following few slides.

3.7.1 Example 1:Using properties file

```
import javax.sql.Connection;
import javax.sql.ResultSet;
import javax.sql.SQLException;
import javax.sql.Statement;
import javax.sql.DataSource;
public class DataSourceTest {
  public static void main (String []args) {
            testDataSource();
  }
      private static void testDataSource() {
            DataSource ds = null;
            ds = MyDataSourceFactory.getMySQLDataSource();
            Connection con = null;
            Statement stmt = null;
            ResultSet rs = null;
```

3.7.1 Example 1:Using properties file

```
try {  con = ds.getConnection();
       stmt = con.createStatement();
       rs = stmt.executeQuery("select * from city");
       while(rs.next()){
         System.out.println("City ID="+rs.getInt("city_id")+",
                            City="+rs.getString("city"));
       }
    }catch (SQLException e) {
            e.printStackTrace();
    }finally{
       try {
            if(rs != null) rs.close();
            if(stmt != null) stmt.close();
            if(con != null) con.close();
       } catch (SQLException e) {
            e.printStackTrace();
       }}}
```

3.7.2 Example 2:Using Web server (Tomcat 9)

- Practically, you may deploy a `DataSource` object through a web server, each one may has a specific way to do that.
- In this example we will illustrate how we can deploy it through Tomcat 9 web server. The following steps must be done before try to get a `DataSource` object using Servlet or JSP java program.
- In the main directory of Tomcat 9, you will find a subdirectory called *conf*. Inside this directory you will find three XML files: *web.xml, context.xml, and server.xml*. We will add the deployed `DataSource` with its properties as will be illustrated in the following slides.

3.7.2 Example 2:Using Web server (Tomcat 9)

- In *server.xml* you have to add the following text in the section of <**GlobalNamingResources**> :

```
<Resource
    name="jdbc/MYDB"
    auth="Container"
    type="javax.sql.DataSource"
    maxTotal="100"
    maxIdle="30"
    maxWaitMillis="10000"
    username="root"
    password="passwd"
    driverClassName="com.mysql.cj.jdbc.Driver"
    url="jdbc:mysql://localhost:3306/sakila"
/>
```

3.7.2 Example 2:Using Web server (Tomcat 9)

- In *context.xml* you have to add the following text in the section of <**Context**> :

```
<Resource
    name="jdbc/MYDB"
    auth="Container"
    type="javax.sql.DataSource"
    maxTotal="100"
    maxIdle="30"
    maxWaitMillis="10000"
    username="root"
    password="passwd"
    driverClassName="com.mysql.cj.jdbc.Driver"
    url="jdbc:mysql://localhost:3306/sakila"
/>
```

3.7.2 Example 2:Using Web server (Tomcat 9)

- In *web.xml* you have to add the following text in the end of the section of <**web-app**> :

```
<description>MySQL Test App</description>
<resource-ref>
      <description>DB Connection</description>
      <res-ref-name>jdbc/MYDB</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
</resource-ref >
```

3.7.2 Example 2:Using Web server (Tomcat 9)

- After finishing the editing in the previous files and save it, you have to write a simple web Java program to get the DataSource object and use it to connect to the database.
- The following Java code is a simple servlet do the same thing like the Example 1 but through web application (Using NetBeans IDE 8.2).

3.7.2 Example 2:Using Web server (Tomcat 9)

```
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

3.7.2 Example 2:Using Web server (Tomcat 9)

```
public class JDBCDataSourceExample extends HttpServlet {
@Override
protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException
{
        Context ctx = null;
        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;
        try{
        ctx = new InitialContext();
       Context envCtx = (Context) ctx.lookup("java:comp/env");
       DataSource ds = (DataSource) envCtx.lookup("jdbc/MYDB");
        con = ds.getConnection();
        stmt = con.createStatement();
        rs = stmt.executeQuery("select * from city");
        PrintWriter out = response.getWriter();
```

3.7.2 Example 2:Using Web server (Tomcat 9)

```
response.setContentType("text/html");
out.print("<html><body><h2>Cities Details</h2>");
out.print("<table border=\"1\" cellspacing=10 cellpadding=5>");
out.print("<th>City ID</th>");
out.print("<th>City Name</th>");
out.print("<th>Country ID</th>");
while(rs.next())
{   out.print("<tr>");
    out.print("<td>" + rs.getInt("city_id") + "</td>");
    out.print("<td>" + rs.getString("city") + "</td>");
    out.print("<td>" + rs.getInt("country_id") + "</td>");
    out.print("</tr>");
}
out.print("</table></body><br/>");
out.print("<h3>Database Details</h3>");
out.print("Database Product:“ +
 con.getMetaData().getDatabaseProductName() + "<br/>");
out.print("Database Driver: "+con.getMetaData().getDriverName());
out.print("</html>");
```

3.7.2 Example 2:Using Web server (Tomcat 9)

```
}catch(NamingException e){
        e.printStackTrace();
} catch (SQLException e) {
        e.printStackTrace();
}finally{
        try {
                rs.close();
                stmt.close();
                con.close();
                ctx.close();
        } catch (SQLException e) {
System.out.println("Exception in closing DB resources");
        } catch (NamingException e) {
System.out.println("Exception in closing Context");
        }

}
}
```

Lab Exercise

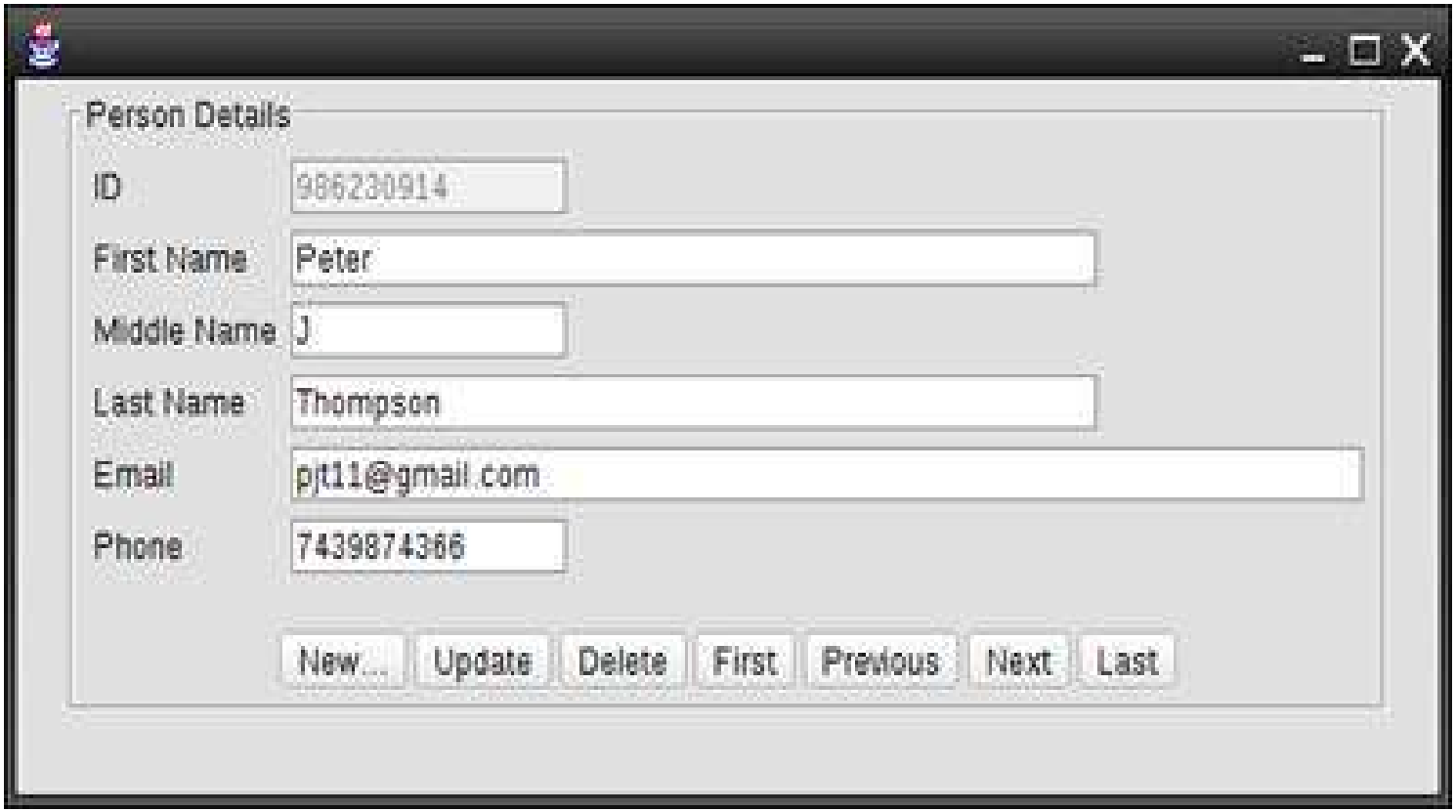
Java™ Education
and Technology Services

Invest In Yourself,
Develop Your Career

Assignments

- Write a program to connect to a database and execute a query for select different fields and print it in a specific format; using an object from DataSource to make the connection to the database, which is obtained from a properties file.
- Make a GUI Java application to retrieve a data of a table from database and view it record after record.
  - Using ***next***, ***previous***, ***first***, and ***last*** buttons to move among records.
  - Use the ***new*** and ***insert*** buttons to insert a new record to the table.
  - Use a ***modify*** button to modify an exist record in data base

Assignments



Lesson 4

Prepared Statement, Transaction, and Batch Update

Java™ Education and Technology Services

Course Outlinesc

Invest In Yourself, Develop Your Career

4.1 Using Prepared Statement

- **4.1.1 Overview of Prepared Statements**
  - Sometimes it is more convenient to use a **`PreparedStatement`** object for sending SQL statements to the database. This special type of statement is derived from the more general interface, **`Statement`**.
  - If you want to execute a **`Statement`** object many times, it usually reduces execution time to use a **`PreparedStatement`** object instead.
  - The main feature of a **`PreparedStatement`** object is that, it is given a SQL statement when it is created. The advantage to this is that in most cases, this SQL statement is sent to the DBMS right away, where it is compiled.

4.1.1 Overview of Prepared Statements

- As a result, the **`PreparedStatement`** object contains not just a SQL statement, but a SQL statement that has been precompiled. This means that when the **`PreparedStatement`** is executed, the DBMS can just run the **`PreparedStatement`** SQL statement without having to compile it first.
- Although **`PreparedStatement`** objects can be used for SQL statements with no parameters, you probably use them most often for SQL statements that take parameters. The advantage of using SQL statements that take parameters is that you can use the same statement and supply it with different values each time you execute it.

4.1.2 Creating a `PreparedStatement` Object

- The following line of codes creates a `PreparedStatement` object that takes two parameters:

```
PreparedStatement  pst = con.prepareStatement
( " INSERT INTO EMPLOYEE VALUES ( ? , ? )" );
```

4.1.3 Supplying Values for `PreparedStatement` Parameters:

- You must supply values in place of the question mark placeholders (if there are any) before you can execute a `PreparedStatement` object. Do this by calling one of the setter methods defined in the `PreparedStatement` class.
- The following statements supply the two question mark placeholders in the `PreparedStatement` named `pst`:

```
pst.setInt(1,500);
pst.setString(2 ,” Max “);
```

- The first argument for each of these setter methods specifies the question mark placeholder. In this example, `setInt` specifies the first placeholder and `setString` specifies the second placeholder.

4.1.4 Execute `PreparedStatement` object

- As with `Statement` objects, to execute a `PreparedStatement` object, call an execute statement:
  - `executeQuery`: if the query returns only one `ResultSet` (such as a SELECT SQL statement),
  - `executeUpdate`: if the query does not return a `ResultSet` (such as an UPDATE SQL statement), or
  - `execute`: if the query might return more than one `ResultSet` object.
- The following statements supply the two question mark placeholders in the `PreparedStatement` named `pst`:

```
pst.setInt(1,500);
pst.setString(2 ,” Max “);
pst.executeUpdate();
```

4.1.4 Execute `PreparedStatement` object

- Whereas `executeQuery` returns a `ResultSet` object containing the results of the query sent to the DBMS, the return value for `executeUpdate` is an `int` value that indicates how many rows of a table were updated. For instance, the following code shows the return value of `executeUpdate` being assigned to the variable `n`:

```
pst.setInt(1,500);
pst.setString(2 ,” Max “);
int n=pst.executeUpdate();
// n = 1 because one row had a change in it
```

4.1.4 Execute `PreparedStatement` object

- When the method `executeUpdate` is used to execute a DDL (data definition language) statement, such as in creating a table, it returns the `int` value of `0`. Consequently, in the following code fragment, which executes the DDL statement used to create the table POINT, `n` is assigned a value of `0`:

```
int n = pst.executeUpdate("CREATE TABLE
POINT (pId in, xPos int, yPos int)");
// n = 0
```

- Note that when the return value for `executeUpdate` is `0`, it can mean one of two things:
  - The statement executed was an update statement that affected zero rows.
  - The statement executed was a DDL statement.

4.2 Using Transactions

- **4.2.1 What is the Transaction:**
  - There are times when you do not want one statement to take effect unless another one completes. The way to be sure that either both actions occur or neither action occurs is to use a transaction.
  - A *transaction* is a *set* of one or more *statements* that is *executed as a unit*, so either *all* of the *statements* are executed, *or* *none* of the *statements* is executed.

4.2.2 Disabling Auto-Commit Mode

- When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and is automatically committed right after it is executed.
- To be more precise, the default is for a SQL statement to be committed when it is completed, not when it is executed. A statement is completed when all of its result sets and update counts have been retrieved. In almost all cases, however, a statement is completed, and therefore committed, right after it is executed.
- The way to allow two or more statements to be grouped into a transaction is to disable the auto-commit mode. This is demonstrated in the following code, where con is an active connection: `con.setAutoCommit( false );`

4.2.3 Committing Transactions

- After the auto-commit mode is disabled, no SQL statements are committed until you call the method `commit` explicitly.
- All statements executed after the previous call to the method `commit` are included in the current transaction and committed together as a unit.
- The `Connection` interface has the `commit` method which has the following form:

```
con.commit();
```

- or you may use the `setAutoCommit` method to make the committing as follow:

```
con.setAutoCommit(true);
```

4.2.4 Setting and Rolling Back to `Savepoints`

- The method `Connection.setSavepoint`, sets a `Savepoint` object within the current transaction. The `Connection.rollback` method is overloaded to take a `Savepoint` argument.

```
Savepoint save1 = con.setSavepoint();
con.rollback(save1);
```

- The method `Connection.releaseSavepoint` takes a `Savepoint` object as a parameter and removes it from the current transaction.
- Any savepoint that have been created in a transaction are automatically released and become invalid when the transaction is committed, or when the entire transaction is rolled back.

4.2.4 Setting and Rolling Back to `Savepoints`

- Calling the method `rollback` terminates a transaction and returns any values that were modified to their previous values. If you are trying to execute one or more statements in a transaction and get a `SQLException`, call the method `rollback` to end the transaction and start the transaction all over again.

4.3 Batch Update

- **4.3.1 Overview of Batch:**
  - A set of multiple update statements that is submitted to the database for processing as a batch.
  - Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.
  - When you send several SQL statements to the database at once, you reduce the amount of communication overhead, thereby improving performance.
  - JDBC drivers are not required to support this feature. You should use the `DatabaseMetaData.supportsBatchUpdates()` method to determine if the target database supports batch update processing. The method returns true if your JDBC driver supports this feature.

4.3.2 How can make a Batch?

- `Statement`, `PreparedStatement` and `CallableStatement` can be used to submit batch updates.
- The `addBatch()` method of `Statement`, `PreparedStatement`, and `CallableStatement` is used to add individual statements to the batch. The `executeBatch()` is used to start the execution of all the statements grouped together.
- Steps of implementing batch update using `Statement`, `PreparedStatement`, and `CallableStatement` interfaces:
  1. Disable auto-commit mode
  2. Create a Statement instance
  3. Add SQL commands to the batch
  4. Execute the batch commands
  5. Commit the changes to the database

4.3.2 How can make a Batch?

- The `executeBatch()` returns an array of integers, and each element of the array represents the update count for the respective update statement.
- Just as you can add statements to a batch for processing, you can remove them with the `clearBatch()` method. This method removes all the statements you added with the `addBatch()` method. However, you cannot selectively choose which statement to remove.

4.3.3 Batching with Statement Object

- Steps to use Batch Processing with `Statement` Object:
  1. Create a Statement object using either `createStatement()` methods.
  2. Set auto-commit to false using `setAutoCommit()`.
  3. Add as many as SQL statements you like into batch using `addBatch()` method.
  4. Execute all the SQL statements using `executeBatch()` method.
  5. Finally, commit all the changes using `commit()` method.
- The following code provides an example of a batch update using `Statement` object

4.3.3 Batching with Statement Object

```
// 1. Create statement object
Statement stmt = conn.createStatement();
// 2. Turn off auto-commit
        con.setAutoCommit( false );
//     Create SQL statement
String SQL ="INSERT INTO Employees (id, first, last, age)"
            + "VALUES(100,'Amr', 'Mostafa', 45)"
// 3. Add the above SQL statement in the batch
       stmt.addBatch(SQL);
//     Create more SQL statements
       SQL ="INSERT INTO Employees (id, first, last, age)"
            + "VALUES(101,'Ahmed', 'Mohamed', 35)"
       stmt.addBatch(SQL);
SQL = "UPDATE Employees SET age = 35 " + "WHERE id = 100";
       stmt.addBatch(SQL);
// 4. Execute the and hold the returned values
int[] count = stmt.executeBatch();
// 5. Explicitly commit statements to apply changes
con.commit();
```

4.3.4 Batching with PreparedStatement Object

- Steps to use Batch Processing with `PrepareStatement` Object:
  1. Create SQL statements with placeholders.
  2. Create `PrepareStatement` object using either `prepareStatement()` methods.
  3. Set auto-commit to false using `setAutoCommit()`.
  4. Add as many as SQL statements you like into batch using `addBatch()` method.
  5. Execute all the SQL statements using `executeBatch()` method.
  6. Finally, commit all the changes using `commit()` method.
- The following code provides an example of a batch update using `PrepareStatement` object

4.3.4 Batching with PreparedStatement Object

```
// 1. Create SQL statement with placeholders.
String SQL = "INSERT INTO Employees (id, first, last, age) "
+ "VALUES(?, ?, ?, ?)";
// 2. Create PrepareStatement object
PreparedStatemen pstmt = conn.prepareStatement(SQL);
// 3. Turn off auto-commit
        con.setAutoCommit( false );
// 4.a. Set the variables
pstmt.setInt( 1, 400 );
pstmt.setString( 2, “Ahmed" );
pstmt.setString( 3, “Medhat" );
pstmt.setInt( 4, 33 );
// 4.b. Add it to the batch
pstmt.addBatch();
// 4.a. Set the variables
pstmt.setInt( 1, 400 );
pstmt.setString( 2, “Bishoy" );
pstmt.setString( 3, “Adel" );
pstmt.setInt( 4, 27 );
```

4.3.4 Batching with PreparedStatement Object

```
// 4.b.Add it to the batch
pstmt.addBatch();

//add more batches
.
.
.
.

// 5. Execute the and hold the returned values
int[] count = pstmt.executeBatch();

// 6. Explicitly commit statements to apply changes
con.commit();
```

Lab Exercise

Java™ Education
and Technology Services

Invest In Yourself,
Develop Your Career

Assignments

1. Make a java application to do the following:
   1. Create a table with name “Employee”, with fields: Id, F_Name, L_Name, Gender, Age, Address, Phone_Number, Vaction_Balance “30 days for each employees”.
   2. Insert 5 rows – at least- with different data using PreparedStatement object.
2. Using Batching to do the following related to the above database:
   1. Modify the Vacation_Balance of employees over 45 years to be increased to 45 days rather than 30,
   2. For those employees, title the F_Name with Mr/Mrs.

Lesson 5

Dealing with RowSet Objects

Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

5.1 What is RowSet?

- A JDBC `RowSet` object is one of the JavaBeans components with multiple supports from JavaBeans, and it is a new feature in the `javax.sql` package.
- JavaBeans components are Java classes that can be easily reused and composed together into applications. Any Java class that follows certain design conventions is a JavaBeans component.
- JavaBeans component design conventions govern the properties of the class and govern the public methods that give access to the properties.
- A JavaBeans component property can be:
  - Read/write, read-only, or write-only
  - Simple, which means it contains a single value, or indexed, which means it represents an array of values
- JavaBeans support event handling mechanism when an event is *fire according to – for example- change of a property may be happened*.

5.1.1 Introduction to RowSet

- A JDBC `RowSet` object is one of the JavaBeans components with multiple supports from JavaBeans, and it is a new feature in the `javax.sql` package. By using the `RowSet` object, a database query can be performed automatically with the data source connection and a query statement creation.
- A JDBC `RowSet` object holds tabular data in a way that makes it more flexible and easier to use than a result set.
- Because a `RowSet` object follows the JavaBeans model for properties and event notification, it is a JavaBeans component that can be combined with other components in an application. As it compatible with other Beans, application developers can probably use a development tool to create a `RowSet` object and set its properties.

5.1.1 Introduction to RowSet

- Oracle has defined five `RowSet` interfaces for some of the more popular uses of a `RowSet`, and standard reference are available for these `RowSet` interfaces.
- These versions of the `RowSet` interface and their implementations have been provided as a convenience for programmers. Programmers are free to write their own versions of the `javax.sql.RowSet` interface, to extend the implementations of the five `RowSet` interfaces, or to write their own implementations. However, many programmers will probably find that the standard reference implementations already fit their needs and will use them as is.

5.1.1 Introduction to RowSet

- The following interfaces are extend `RowSet` interface:
  - `JdbcRowSet`
  - `CachedRowSet`
  - `WebRowSet`
  - `JoinRowSet`
  - `FilteredRowSet`

5.1.2 What can RowSet object do?

- All `RowSet` objects are derived from the `ResultSet` interface and therefore share its capabilities. What makes JDBC `RowSet` objects special is that they add these new capabilities:
  - Function as JavaBeans Component
  - Add Scrollability or Updatability

5.1.2.1 Function as JavaBeans Component

- All `RowSet` objects are JavaBeans components. This means that they have the following:
  - Properties
  - JavaBeans Notification Mechanism

5.1.2.1.1 Properties

- All RowSet objects have properties. A property is a field that has corresponding getter and setter methods. Properties are exposed to builder tools (such as those that come with the IDEs NetBeans, JDveloper and Eclipse) that enable you to visually manipulate beans.

5.1.2.1.2 JavaBeans Notification Mechanism

- `RowSet` objects use the JavaBeans event model, in which registered components are notified when certain events occur. For all `RowSet` objects, three events trigger notifications:
  - A cursor movement
  - The update, insertion, or deletion of a row
  - A change to the entire `RowSet` contents
- The notification of an event goes to all ***listeners***, ***components*** that have implemented the `RowSetListener` interface and have had themselves added to the `RowSet` object's list of components to be notified when any of the three events occurs.

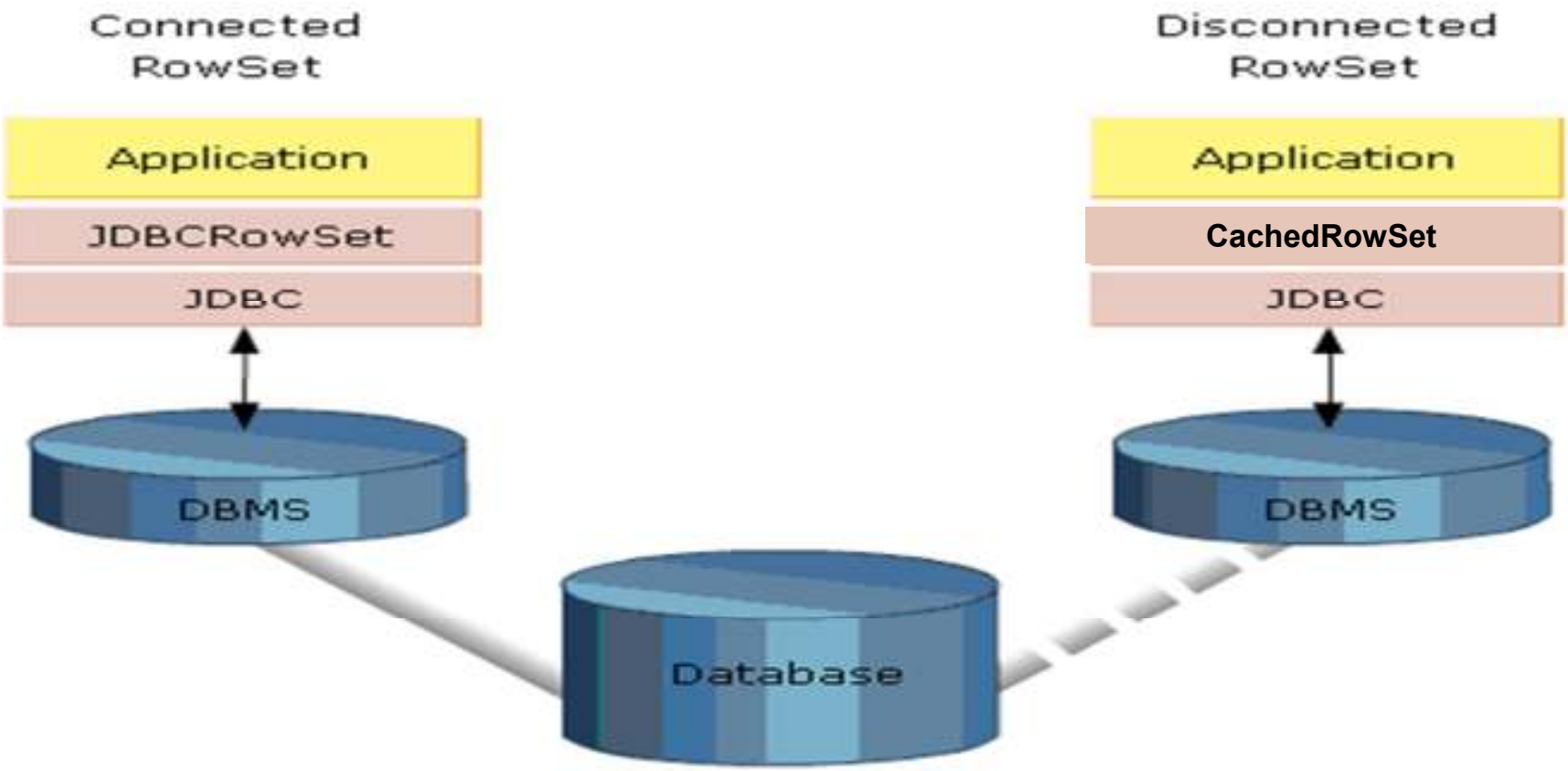
5.1.2.1.2 JavaBeans Notification Mechanism

- A listener could be a GUI component such as a bar graph. If the bar graph is tracking data in a `RowSet` object, the listener would want to know the new data values whenever the data changed. The listener would therefore implement the `RowSetListener` methods to define what it will do when a particular event occurs. Then the listener also must be added to the `RowSet` object's list of listeners. The following line of code registers the bar graph component `bg` with the `RowSet` object `rs`: **`rs.addRowSetListener(bg);`**
- Three types of events are supported by the `RowSet` interface:
  1. **`cursorMoved`** event : Generated whenever there is a cursor movement, which occurs when the `next()` or `previous()` methods are called.
  2. **`rowChanged`** event : Generated when a new row is *inserted*, *updated*, or *deleted* from the row set.
  3. **`rowsetChanged`** event : Generated when the whole row set is *created* or changed.

5.1.2.2 Add Scrollability or Updatability

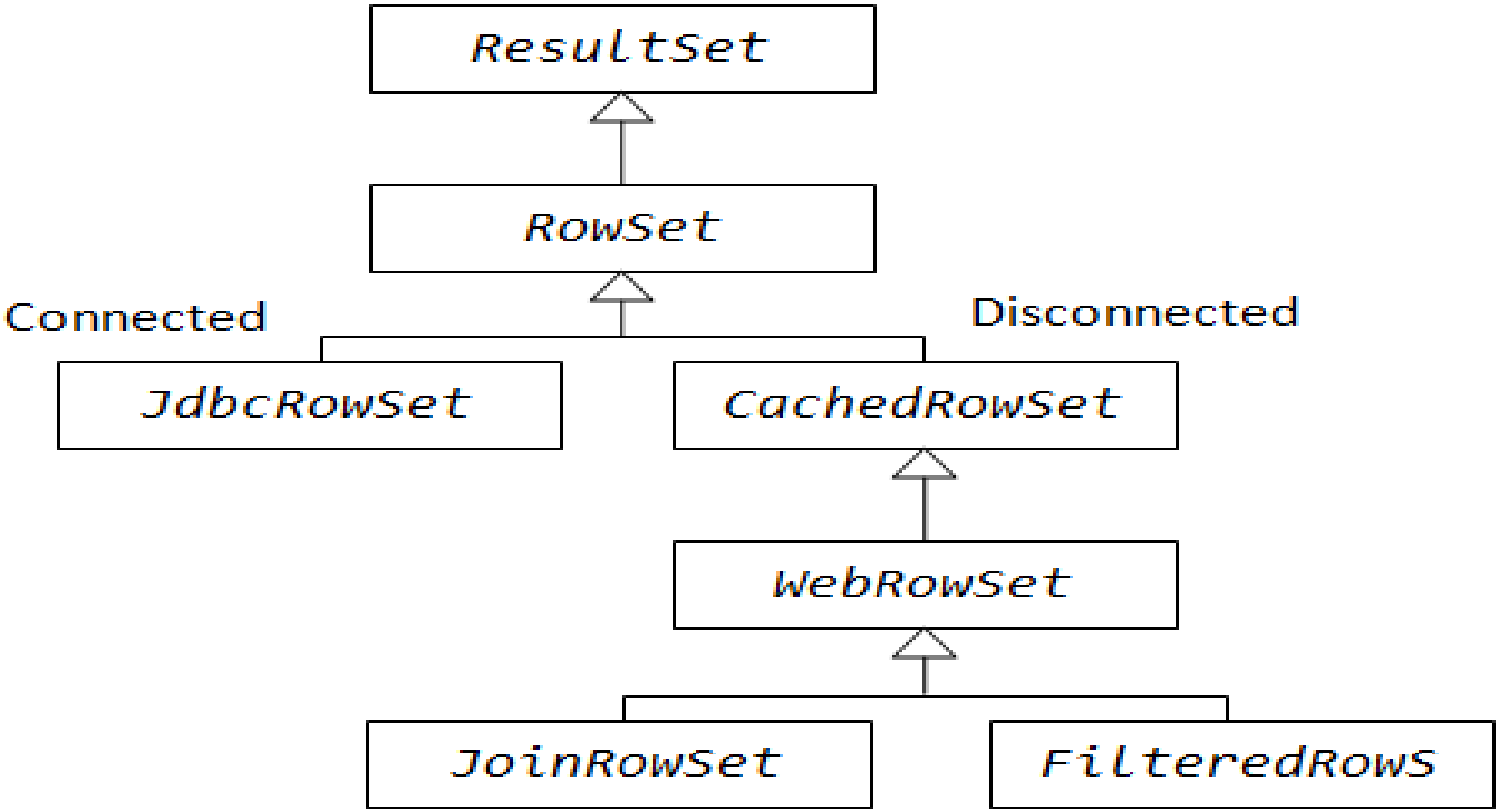
- Some DBMSs do not support result sets that can be scrolled (scrollable), and some do not support result sets that can be updated (updatable). If a driver for that DBMS does not add the ability to scroll or update result sets, you can use a `RowSet` object to do it.
- A `RowSet` object is scrollable and updatable by default, so by populating a `RowSet` object with the contents of a result set, you can effectively make the result set scrollable and updatable

5.1.3 Kinds of `RowSet` Objects



Connected and Disconnected RowSets

5.1.3 Kinds of `RowSet` Objects



5.1.3.1 Connected `RowSet` Objects

- A *connected* `RowSet` object uses a JDBC driver to make a connection to a relational database and maintains that connection throughout its life span.
- Only one of the standard `RowSet` implementations is a connected `RowSet` object: `JdbcRowSet`.
- A `JdbcRowSet` object is most similar to a `ResultSet` object and is often used as a wrapper to make an otherwise non-scrollable and read-only `ResultSet` object **scrollable** and **updatable**.

5.1.3.2 Disconnected `RowSet` Objects

- A *disconnected* `RowSet` object makes a connection to a data source only to read in data from a `ResultSet` object or to write data back to the data source. After reading data from or writing data to its data source, the `RowSet` object disconnects from it, thus becoming "**disconnected**." During much of its life span, a disconnected `RowSet` object has no connection to its data source and operates independently.
- The other four implementations are disconnected `RowSet` implementations.
- Disconnected `RowSet` objects have all the capabilities of connected `RowSet` objects plus they have the additional capabilities available only to disconnected `RowSet` objects.

5.1.3.2 Disconnected `RowSet` Objects

- The `CachedRowSet` interface defines the basic capabilities available to all disconnected `RowSet` objects. The other three are extensions of the `CachedRowSet` interface, which provide more specialized capabilities.
- A `CachedRowSet` object has all the capabilities of a `JdbcRowSet` object plus it can also do the following:
  - Obtain a connection to a data source and execute a query
  - Read the data from the resulting `ResultSet` object and populate itself with that data
  - Manipulate data and make changes to data while it is disconnected
  - Reconnect to the data source to write changes back to it
  - Check for conflicts with the data source and resolve those conflicts

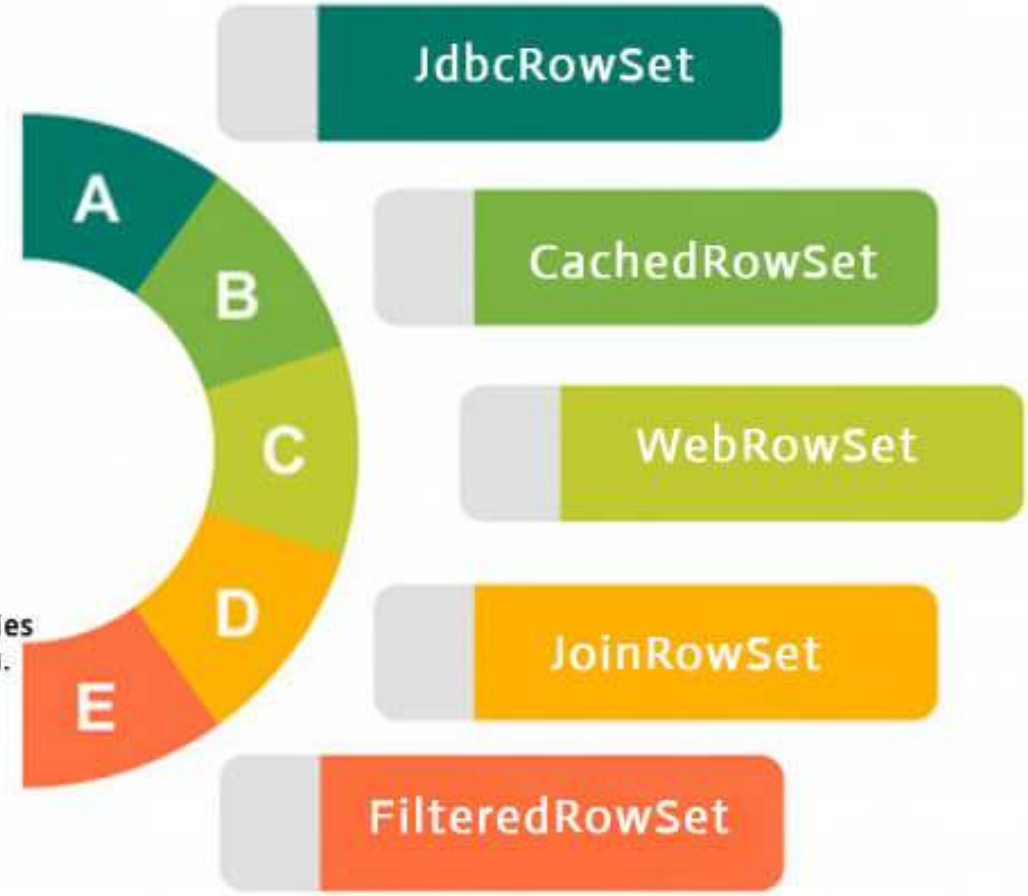
5.1.3.2 Disconnected `RowSet` Objects

- A `WebRowSet` object has all the capabilities of a `CachedRowSet` object plus it can also do the following:
  - Write itself as an XML document
  - Read an XML document that describes a `WebRowSet` object
- A `JoinRowSet` object has all the capabilities of a `WebRowSet` object (and therefore also those of a `CachedRowSet` object) plus it can also do the following:
  - Form the equivalent of a SQL JOIN without having to connect to a data source.
- A `FilteredRowSet` object likewise has all the capabilities of a `WebRowSet` object (and therefore also a `CachedRowSet` object) plus it can also do the following:
  - Apply filtering criteria so that only selected data is visible. This is equivalent to executing a query on a `RowSet` object without having to use a query language or connect to a data source.

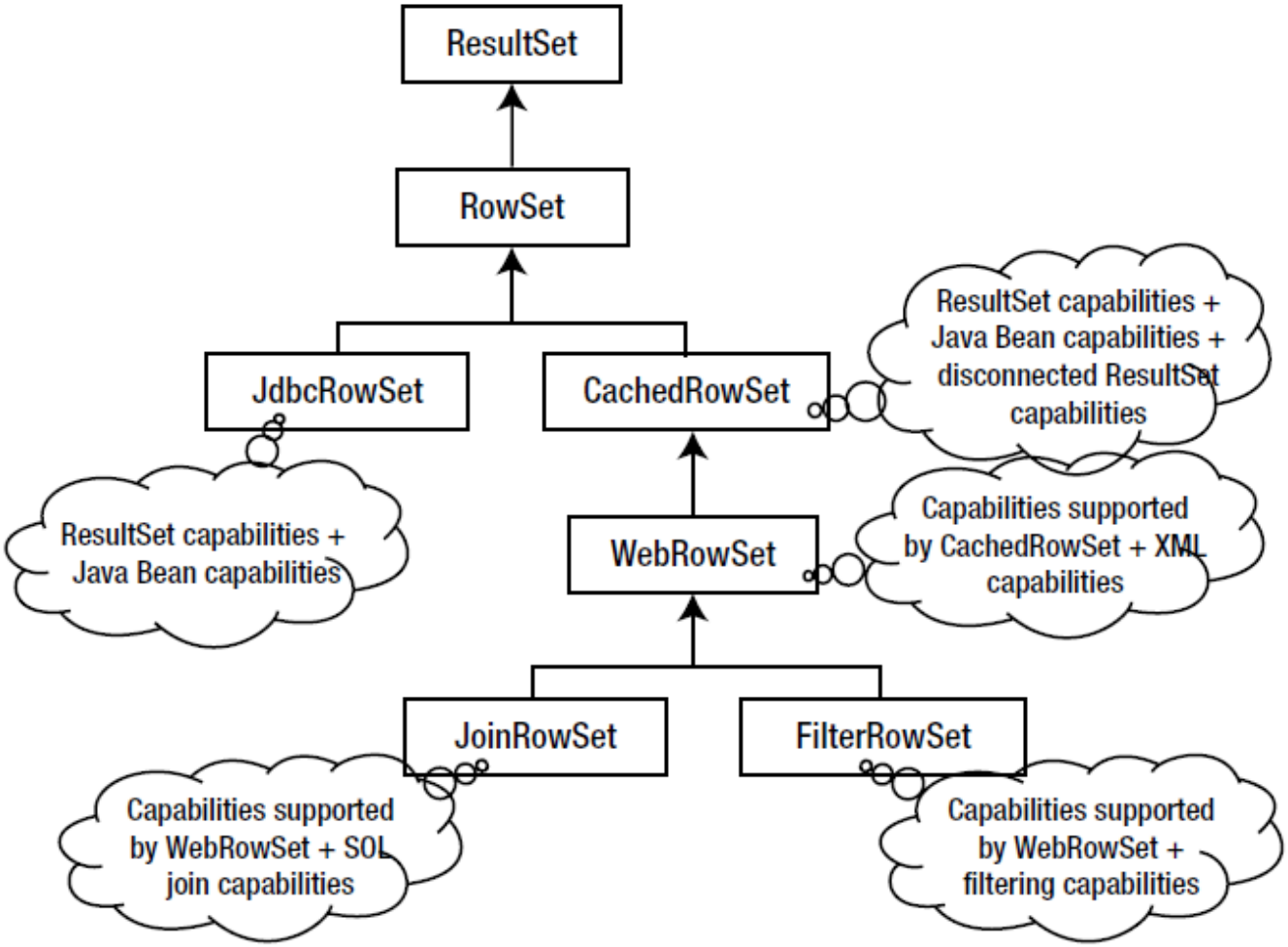
5.1.3 Kinds of RowSet Objects

JDBC RowSet

The instance of RowSet is the java bean component because it has properties and java bean notification mechanism. It is introduced since JDK 5.



5.1.3 Kinds of RowSet Objects



5.1.4 How to Create any type of RowSet?

- There are many ways to create objects of any type of `RowSet`:
  - First way by using the standard implementation of the interfaces; `JDBCRowSetImpl`, `CashedRowSetImpl`, `WebRowSetImpl`, ... etc. But these classes are internal proprietary API and may be removed in a future release, they are in package named: `com.sun.rowset.`
  - Other way by using `RowSetFactory` which is created by using `RowSetProvider`, as follow:

```
RowSetFactory aFactory=RowSetProvider.newFactory();
```

  - `RowSetFactory` has many methods to create the different types of `RowSet` as follow:

```
CachedRowSet crs = aFactory. createCachedRowSet();
JDBCRowSet jrs = aFactory. createJDBCRowSet();
FilteredRowSet crs=aFactory. createFilteredRowSet();
JoinCachedRowSet crs=aFactory. createJoinRowSet();
WebRowSet crs = aFactory. createWebRowSet();
```

- You may implement these interfaces or use a prepared other implementation.

5.2 Using JdbcRowSet Objects

- **5.2.1 Why We Are Using `JDBCRowSet` Objects?**
  - A JdbcRowSet object is an enhanced `ResultSet` object. It maintains a connection to its data source, just as a `ResultSet` object does.
  - The big difference is that it has a set of properties and a listener notification mechanism that make it a JavaBeans component.
  - One of the main uses of a `JdbcRowSet` object is to make a `ResultSet` object scrollable and updatable when it does not otherwise have those capabilities.

5.2.2 Creating `JdbcRowSet` Objects

- You can create a `JdbcRowSet` object in various ways:
  - By using the reference implementation constructor that takes a `ResultSet` object
  - By using the reference implementation constructor that takes a `Connection` object
  - By using the reference implementation default constructor
  - By using an instance of `RowSetFactory`, which is created from the class `RowSetProvider`

5.2.2.1 Passing ResultSet Objects

- The simplest way to create a `JdbcRowSet` object is to produce a `ResultSet` object and pass it to the `JdbcRowSetImpl` constructor. Doing this not only creates a `JdbcRowSet` object but also populates it with the data in the `ResultSet` object. Note: The `ResultSet` object that is passed to the `JdbcRowSetImpl` constructor must be scrollable.

```
stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
rs = stmt.executeQuery("select ID, Name from Employees");
JdbcRowSetImpl jdbcRs = new JdbcRowSetImpl(rs);
```

- A `JdbcRowSet` object created with a `ResultSet` object serves as a wrapper for the `ResultSet` object. Because the `RowSet` object `rs` is scrollable and updatable, `jdbcRs` is also scrollable and updatable. If you have run the method `createStatement` without any arguments, `rs` would not be scrollable or updatable, and neither would `jdbcRs`.

5.2.2.2 Passing Connection Objects

```
JdbcRowSetImpl jdbcRs = new JdbcRowSetImpl(con);
jdbcRs.setCommand("select ……");
jdbcRs.execute();
```

- The object `jdbcRs` contains no data until you specify a SQL statement with the method `setCommand`, then run the method `execute`.
- The object `jdbcRs` is scrollable and updatable; by default, `JdbcRowSet` and all other `RowSet` objects are scrollable and updatable unless otherwise specified. See Default `JdbcRowSet` Objects for more information about `JdbcRowSet` properties you can specify.

5.2.2.3 Using the Default Constructor

```
jdbcRs = new JdbcRowSetImpl();
jdbcRs.setCommand("select * from Employees");
jdbcRs.setUrl("jdbc:mysql://localhost:3306/sakila");
jdbcRs.setUsername("root");
jdbcRs.setPassword("passwd");
jdbcRs.execute();
```

- The object `jdbcRs` contains no data until you specify a SQL statement with the method `setCommand`, specify how the `JdbcResultSet` object connects the database, and then run the method `execute`.

5.2.2.3 Using the Default Constructor

- The new `JdbcRowSet` object using default constructorwill have the following properties:
  - **`type: ResultSet.TYPE_SCROLL_INSENSITIVE`** (has a scrollable cursor)
  - **`concurrency: ResultSet.CONCUR_UPDATABLE`** (can be updated)
  - **`escapeProcessing: true`** (the driver will do escape substitution before sending an SQL statement to the database; i.e. the driver will scan for any escape syntax and translate it into code that the particular database understands)
  - **`maxRows: 0`** (no limit on the number of rows)
  - **`maxFieldSize: 0`** (no limit on the number of bytes for a column value; applies only to columns that store BINARY, VARBINARY, LONGVARBINARY, CHAR, VARCHAR, and LONGVARCHAR values)
  - **`queryTimeout: 0`** (has no time limit for how long it takes to execute a query)
  - **`showDeleted: false`** (deleted rows are not visible; the deleted rows are set as invisible with the current set of rows.)

5.2.2.4 Using the `RowSetFactory` Interface

```
RowSetFactory myRSF = RowSetProvider.newFactory();
jdbcRs = myRSF.createJdbcRowSet();
jdbcRs.setUrl("jdbc:mysql://localhost:3306/sakila" );
jdbcRs.setUsername("root");
jdbcRs.setPassword("passwd");
jdbcRs.setCommand("select * from Employees");
jdbcRs.execute();
```

- The first statement creates the `RowSetProvider` object `myRowSetFactory` with the default `RowSetFactory` implementation, `com.sun.rowset.RowSetFactoryImpl`.
- If your JDBC driver has its own `RowSetFactory` implementation, you may specify it as an argument of the `newFactory` method.

5.2.2.4 Using the `RowSetFactory` Interface

- The remain statements create the `JdbcRowSet` object `jdbcRs` and configure its database connection properties.
- The `RowSetFactory` interface contains methods to create the different types of `RowSet` implementations available in `RowSet` 1.1 (Since JSE 7) and later:
  - `createCachedRowSet`
  - `createFilteredRowSet`
  - `createJdbcRowSet`
  - `createJoinRowSet`
  - `createWebRowSet`

5.2.3 How to use JdbcRowSet Objects

- You *update*, *insert*, and *delete* a row in a `JdbcRowSet` object the same way you update, insert, and delete a row in an updatable `ResultSet` object. Similarly, you navigate a `JdbcRowSet` object the same way you navigate a scrollable `ResultSet` object.
- Because a `JdbcRowSet` object has an ongoing connection to the database, changes it makes to its own data are also made to the data in the database.
- The following database operations could be done:
  - Navigating JdbcRowSet Objects
  - Updating Column Values
  - Inserting Rows
  - Deleting Rows

5.2.3.1 Navigating `JdbcRowSet` Objects

- A default `JdbcRowSet` object, however, can use all of the cursor movement methods defined in the `ResultSet` interface.
- For example:

```
jdbcRs.absolute(4);
jdbcRs.previous();
jdbcRs.first();
jdbcRs.afterLast();
```

5.2.3.2 Updating Column Values

- You update data in a `JdbcRowSet` object the same way you update data in a `ResultSet` object.
- For example:

```
jdbcRs.absolute(3);
jdbcRs.updateString("first", "Fatema");
jdbcRs.updateRow();
```

- The code moves the cursor to the third row and changes the value for the column FIRST to "Fatema", and then updates the database with the new name.
- Calling the method `updateRow` updates the database because `jdbcRs` has maintained its connection to the database.

5.2.3.3 Inserting Rows

- Because the `jdbcRs` object is always connected to the database, inserting a row into a `JdbcRowSet` object is the same as inserting a row into a `ResultSet` object: You move to the cursor to the *insert row*, use the appropriate *updater* method to set a value for each column, and call the method *`insertRow`*.
- For example:

```
jdbcRs.moveToInsertRow();
jdbcRs.updateInt("id", 301);
jdbcRs.updateString("first", "Hisham");
jdbcRs.updateString("last", "Farouk");
jdbcRs.updateInt("age", 33);
jdbcRs.insertRow();
```

- When you call the method `insertRow`, the new row is inserted into the `jdbcRs` object and is also inserted into the database.

5.2.3.4 Deleting Rows

- Deleting a row is just the same for a `JdbcRowSet` object as for a `ResultSet` object.
- For example:

```
jdbcRs.last();
jdbcRs.deleteRow();
```

5.3 Using CachedRowSet Objects

- **5.3.1 When we use `CashedRowSet`?**
  - A `CachedRowSet` object is special in that it can operate without being connected to its data source, that is, it is a *disconnected* `RowSet` object.
  - The `CachedRowSet` interface is the super interface for all disconnected `RowSet` objects, so everything demonstrated here also applies to `WebRowSet`, `JoinRowSet`, and `FilteredRowSet` objects.
  - A `CachedRowSet` object is capable of getting data from any data source that stores its data in a tabular format.

5.3.2 Setting Up CachedRowSet Objects

- Setting up a `CachedRowSet` object involves the following:
  - Creating `CachedRowSet` Objects
  - Setting `CachedRowSet` Properties
  - Setting Key Columns

5.3.2.1 Creating CachedRowSet Objects

- You can create a new `CachedRowSet` object in the different ways:
  - Using the Default Constructor
  - Using an instance of `RowSetFactory`, which is created from the class `RowSetProvider`.

5.3.2.1.1 Using the Default Constructor

```
CachedRowSet crs = new CachedRowSetImpl();
```

- The object `crs` has the same default values for its properties that a `JdbcRowSet` object has when it is first created. In addition, it has been assigned an instance of the default `SyncProvider` implementation, `RIOptimisticProvider`.

5.3.2.1.2 Using an instance of RowSetFactory

```
myRowSetFactory = RowSetProvider.newFactory();
crs = myRowSetFactory.createCachedRowSet();
```

5.3.2.2 Setting CachedRowSet Properties

- The following properties hold information necessary to obtain a connection to a database:
  - **username**: The name a user supplies to a database as part of gaining access
  - **password**: The user's database password
  - **url**: The JDBC URL for the database to which the user wants to connect
  - **datasourceName**: The name used to retrieve a DataSource object that has been registered with a JNDI naming service.
- These properties must be setting by the appropriate setter method.
- Another property that you must set is the command property, by `setCommand()`.

5.3.2.3 Setting Key Columns

- If you are going make any updates to the `crs` object and want those updates saved in the database, you must set one more piece of information: the key columns.
- Key columns are essentially the same as a primary key because they indicate one or more columns that uniquely identify a row. The difference is that a primary key is set on a table in the database, whereas key columns are set on a particular `RowSet` object.
- The following lines of code set the key columns for `crs` to the first column:

```
int [] keys = {1}; // 1 indicates the col No
crs.setKeyColumns(keys);
```

- The first column in the table **Employees** is **id**. It can serve as the key column, where this column is specified as a primary key in the definition of the **Employees** table. The method `setKeyColumns` takes an array to allow for the fact that it may take two or more columns to identify a row uniquely.

5.3.3 Populating CachedRowSet Objects

- Populating a disconnected `RowSet` object involves more work than populating a connected `RowSet` object. The following line of code populates `crs`: **`crs.execute();`**
- What is different is that the `CachedRowSet` implementation for the execute method does a lot more than the `JdbcRowSet` implementation; the `CachedRowSet` object's reader, to which the method execute delegates its tasks, does a lot more.
- Every disconnected `RowSet` object has a `SyncProvider` object assigned to it, and this `SyncProvider` object is what provides the `RowSet` object's reader (a `RowSetReader` object). When the `crs` object was created, it was used as the default `CachedRowSetImpl` constructor, which assigns an instance of the `RIOptimisticProvider` implementation as the default `SyncProvider` object.

5.3.4 What Reader Does?

- A newly created `CachedRowSet` object is not connected to a data source and therefore must obtain a connection to that data source in order to get data from it.
- The reference implementation of the default `SyncProvider` object (`RIOptimisticProvider`) provides a reader that obtains a connection by using the values set for the user name, password, and either the JDBC URL or the data source name, whichever was set more recently.
- Then the reader executes the query set for the command. It reads the data in the `ResultSet` object produced by the query, populating the `CachedRowSet` object with that data. Finally, the reader closes the connection.

5.3.5 Updating CachedRowSet Object

- **5.3.5.1 Updating Column Values:**
  - Updating data in a `CachedRowSet` object is just the same as updating data in a `JdbcRowSet` object.

```
crs.updateInt("age", 34);
crs.updateRow();
// Synchronizing the row
// back to the DB
crs.acceptChanges(con);
```

5.3.5.2 Inserting and Deleting Rows

- The code for inserting and deleting rows in a `CachedRowSet` object is the same as for a `JdbcRowSet` object.

```
crs.moveToInsertRow();
crs.updateInt("id", newItemId);
crs.updateString("first", "Amira");
crs.updateString("last", "Hussain");
crs.updateInt("age", 27);
crs.insertRow();
crs.moveToCurrentRow();
```

- For delete a row:

```
if (crs.getInt("id") == 301) {
        crs.deleteRow();
}
```

5.3.6 Updating Data Sources

- In the case of a disconnected `RowSet` object, the methods `updateRow`, `insertRow`, and `deleteRow` update the data stored in the `CachedRowSet` object's memory but cannot affect the data source. A disconnected `RowSet` object must call the method `acceptChanges` in order to save its changes to the data source.

```
crs.acceptChanges(con);
```

5.3.7 What Writer Does?

- Whereas the method execute delegates its work to the `RowSet` object's reader, the method `acceptChanges` delegates its tasks to the `RowSet` object's writer. In the background, the writer opens a connection to the database, updates the database with the changes made to the `RowSet` object, and then closes the connection.

5.3.8 Notifying Listeners

- Being a JavaBeans component means that a RowSet object can notify other components when certain things happen to it.
- **5.3.8.1 Setting Up Listeners**
  - A listener for a RowSet object is a component that implements the following methods from the RowSetListener interface:
    - cursorMoved
    - rowChanged
    - rowSetChanged
  - For setting a listener: `crs.addRowSetListener(bar);`
  - For stop notification: `crs.removeRowSetListener(bar);`

5.4 Using JoinRowSet Objects

- **5.4.1 Why we are using `JoinRowSet` objects?**
  - A `JoinRowSet` implementation lets you create a SQL JOIN between `RowSet` objects when they are not connected to a data source. This is important because it saves the overhead of having to create one or more connections.
  - The `JoinRowSet` interface is a sub interface of the `CachedRowSet` interface and thereby inherits the capabilities of a `CachedRowSet` object.
  - This means that a `JoinRowSet` object is a disconnected `RowSet` object and can operate without always being connected to a data source.

5.4.2 Creating JoinRowSet Objects

- A `JoinRowSet` object serves as the holder of a SQL JOIN. The following line of code shows to create a `JoinRowSet` object:

```
JoinRowSet jrs = new JoinRowSetImpl();
```

- The variable `jrs` holds nothing until `RowSet` objects are added to it.

5.4.3 Adding RowSet Objects

- Any RowSet object can be added to a JoinRowSet object as long as it can be part of a SQL JOIN.
- For Example:

```
emps = new ChachedRowSetImpl();
emps.setCommand("select * from Emplyees");
emps.execute();
mangs = new ChachedRowSetImpl();
mangs.setCommand("select * from Managers");
Mangs.execute();
jrs.addRowSet(emps, "id"); // or by col No
jrs.addRowSet(mangs, "id"); // or by col No
```

- The `JoinRowSet` interface provides constants for setting the type of JOIN, but currently the only type that is implemented in `JoinRowSetImpl` is `JoinRowSet.INNER_JOIN`.

5.5 Using FilteredRowSet Objects

- **5.5.1 Why we are using `FilteredRowSet` objects?**
  - A `FilteredRowSet` object lets you cut down the number of rows that are visible in a `RowSet` object so that you can work with only the data that is relevant to what you are doing.
  - You decide what limits you want to set on your data (how you want to "filter" the data) and apply that filter to a `FilteredRowSet` object.
  - A `FilteredRowSet` object is providing the following capabilities:
    - Ability to limit the rows that are visible according to set criteria.
    - Ability to select which data is visible without being connected to a data source.

5.5.2 Defining Filtering Criteria in Predicate Objects

- To set the criteria for which rows in a `FilteredRowSet` object will be visible, you define a class that implements the `Predicate` (`javax.sql.rowset.Predicate`) interface.
- This interface has 3 abstract methods:
  - `boolean evaluate(Object value, int column)`
  - `boolean evaluate(Object value, String column)`
  - `boolean evaluate(RowSet rs)`
- An object created with this class is initialized with the following:
  - The high end of the range within which values must fall
  - The low end of the range within which values must fall
  - The column name or column number of the column with the value that must fall within the range of values set by the high and low boundaries

5.5.3 Creating FilteredRowSet Objects

- The reference implementation for the `FilteredRowSet` interface, `FilteredRowSetImpl`, includes a default constructor, which is used in the following line of code to create the empty `FilteredRowSet` object `frs`:

```
FilteredRowSet frs = new FilteredRowSetImpl();
frs.setCommand("SELECT * FROM Employees");
frs.setUsername("root");
frs.setPassword("passwd");
frs.setUrl(urlString);
frs.execute();
```

5.5.4 Creating and Setting Predicate Objects

- Now `frs` is contained all the employees from the table, you can set selection criteria for narrowing down the number of rows in the `frs` object that are visible.
- The following line of code uses the `EmployeeFilter` class to create the object `myEmpFilter`, which checks the column AGE to determine which are to be near to 60 years old (55 till 59)

```
EmployeeFilter myEmpFilter = new EmployeeFilter();
frs.setFilter(myEmpFilter);
```

- To do the actual filtering, you call the method `next`, which in the reference implementation calls the appropriate version of the `Predicate.evaluate` method that you have implemented.
- If the return value is true, the row will be visible; if the return value is false, the row will not be visible.

5.5.5 Removing All Filters so All Rows Are Visible

- The following line of code unsets the current filter, effectively nullifying both of the `Predicate` implementations set on the `frs` object.

```
frs.setFilter(null);
```

5.6 Using WebRowSet Objects

- **5.6.1 Why we are using `WebRowSet` objects?**
  - A `WebRowSet` object is very special because in addition to offering all of the capabilities of a `CachedRowSet` object, it can write itself as an XML document and can also read that XML document to convert itself back to a `WebRowSet` object.
  - Because XML is the language through which disparate enterprises can communicate with each other, it has become the standard for Web Services communication. As a consequence, a `WebRowSet` object fills a real need by enabling Web Services to send and receive data from a database in the form of an XML document.

5.6.2 Creating and Populating WebRowSet Objects

- You create a new `WebRowSet` object with the default constructor defined in the reference implementation, `WebRowSetImpl`, as shown in the following line of code:

```
WebRowSet fullNamesList = new WebRowSetImpl();
```

- The full names list consists of the data in the columns FIRST and LAST from the table Employees. The following code fragment sets the properties needed and populates the fullNamesList object with the names list data:

```
fullNamesList.setCommand("select first, last from employees");
 fullNamesList.setURL(dataBaseURL);
 fullNamesList.setUsername(username);
 fullNamesList.setPassword(password);
 fullNamesList.execute();
```

5.6.3 Writing and Reading WebRowSet Object to XML

- To write a `WebRowSet` object as an XML document, call the method `writeXml`. To read that XML document's contents into a `WebRowSet` object, call the method `readXml`. Both of these methods do their work in the background, meaning that everything, except the results, is invisible to you.

- **5.6.3.1 Using the writeXml Method:**

```
FileOutputStream out = new
                FileOutputStream("emplist.xml");
fullNamesList.writeXML(out); // or you may use writer
FileWriter writer = new FileWriter("emplist.xml");
fullNamesList.writeXML(writer);
```

5.6.3.1 Using the readXml Method

```
FileInputStream in = new
     FileInputStream("emplist.xml");
fullNamesList.readXML(in); // or you may use reader
FileReader reader = new FileReader("emplist.xml");
fullNamesList.writeXML(reader);
```

Lab Exercise

Assignments

- Write a simple Java application to use JDBCRowSet object to perform selection statement from a database and view the result.
- Write a simple Java application to use CachedRowSet object to perform selection statement from a database and view the result, make update, insert, or delete and make it to be accepted by database.
- Write a simple Java application to use JoinRowSet object to perform 2 selection statement from a database join them, and view the result.
- Write a simple Java application to use WebRowSet object to perform selection statement from a database and view the result and save it in a XML file.

References

- The Java Tutorials, JDBC; JDBC Basics https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html
- Tutorials Point; JDBC Tutorial https://www.tutorialspoint.com/jdbc/index.htm
- Database Star, Ben Brumm: JDBC in Java https://www.databasestar.com/jdbc-in-java/
- Java-T-Point: https://www.javatpoint.com/java-jdbc
- Ying Bai, 2011, Practical Database Programming with Java, A John Wiley & Sons, Inc., Publication.

References

- JDBC - Java Database Connectivity Tutorial: RoseIndia
  https://www.roseindia.net/jdbc/
- Donald Bales, 2002, Java Programming with Oracle JDBC, O'Reilly
- Oracle Database JDBC Developer's Guide, Oracle help center.

https://docs.oracle.com/cd/E11882_01/java.112/e16548/overvw.htm#JJDBC28025