

Java Database Connectivity (JDBC)

What is JDBC?

JDBC

- 1 The JDBC (Java Database Connectivity) API helps a ***Java program to access a database in a standard way***
- 2 JDBC is a ***specification*** that tells the database vendors how to write a driver program to interface Java programs with their database
- 3 A Driver written according to this standard is called the ***JDBC Driver***
- 4 All related classes and interfaces are present in the ***java.sql*** package
- 5 All JDBC Drivers ***implement the interfaces*** of java.sql

JDBC Architecture

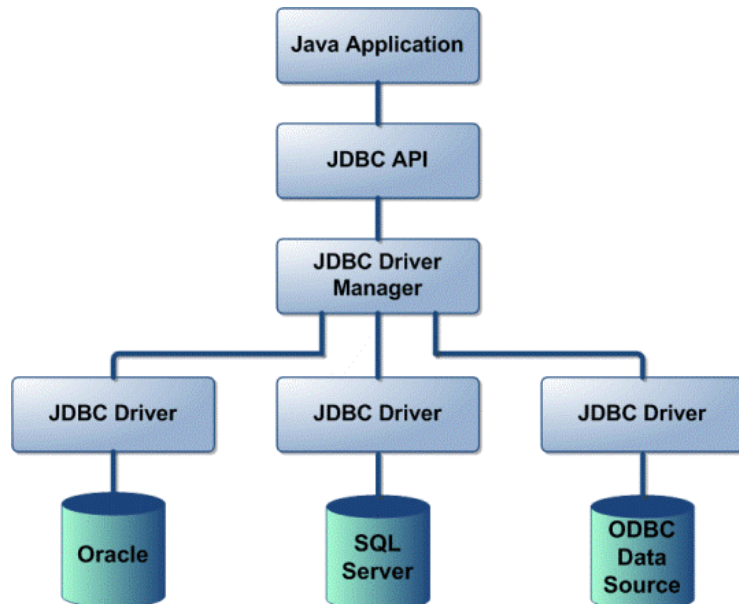
JDBC Architecture consists of two layers:

JDBC API: This provides the *application-to-JDBC Manager* connection.

JDBC Driver API: This supports the *JDBC Manager-to-Driver* Connection.

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

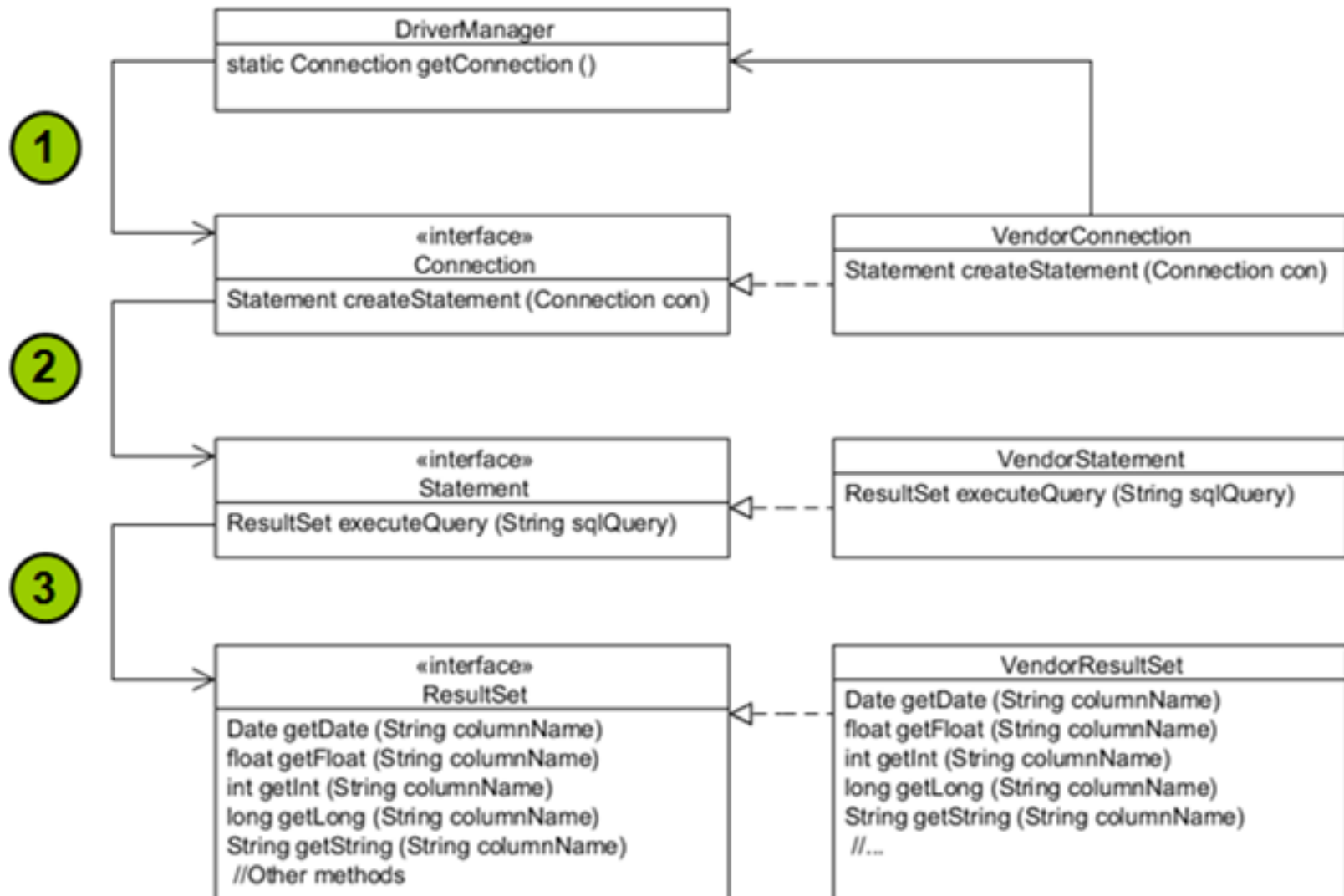


Java Database Connectivity (JDBC) API library Versions:

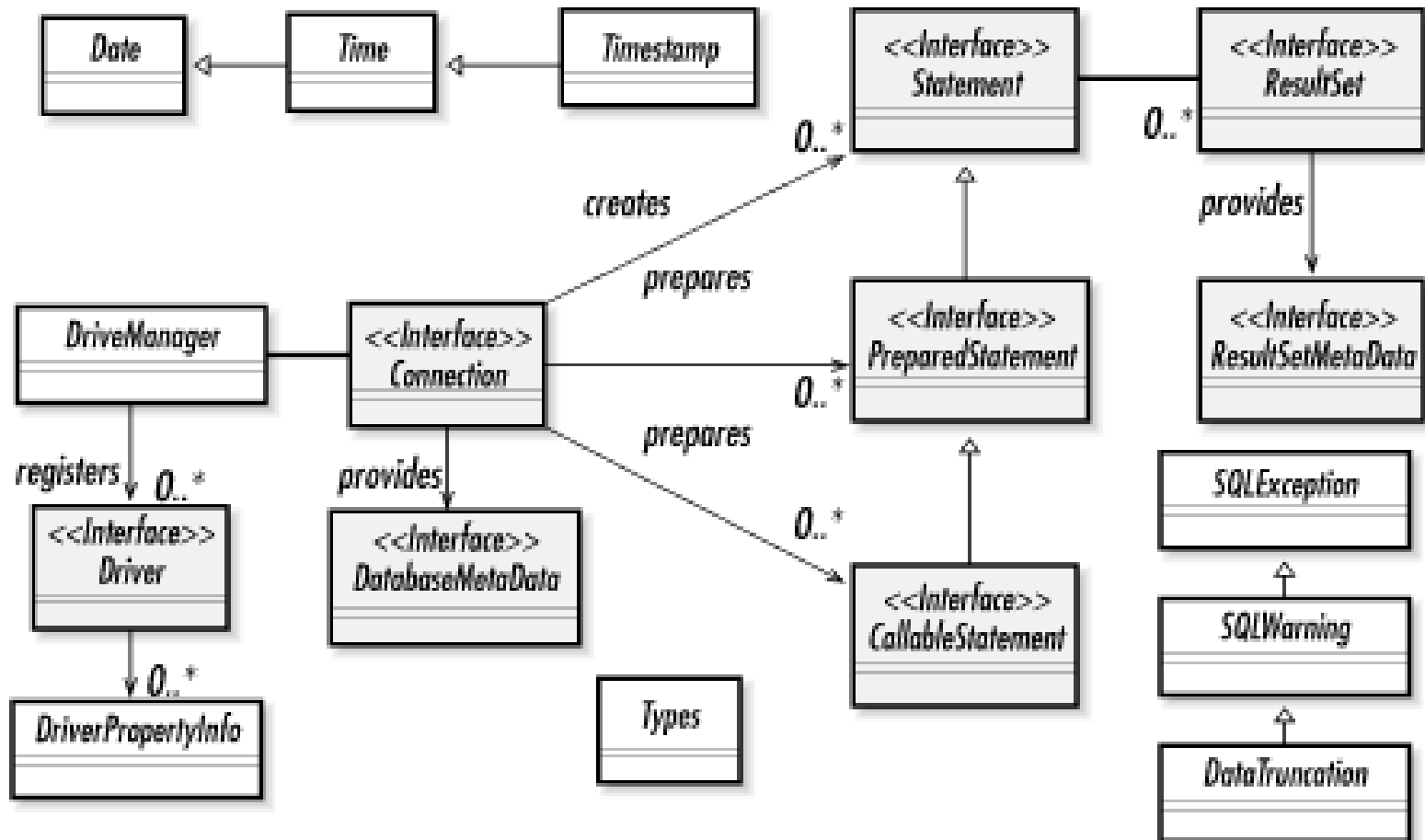
- The JDBC 1.0 API
- The JDBC 1.2 API
- The JDBC 2.0 API
- The JDBC 3.0 API
- Latest as on current date: The JDBC 4.0 API**

JDBC API

Using the JDBC API



JDBC API



JDBC API

The JDBC API is made up of some concrete classes, such as Date, Time, and SQLException, and a ***set of interfaces that are implemented in a driver class that is provided by the database vendor.***

Because the implementation is a valid instance of the interface method signature, once the database vendor's Driver classes are loaded, you can access them by following below sequence:

1. Use the class DriverManager to obtain a reference to a Connection object using the getConnection method . The typical signature of this method is *getConnection (url, name, password)*, where url is the JDBC URL, and name and password are strings that the database will accept for a connection.
2. Use the Connection object (implemented by some class that the vendor provided) to obtain a reference to a Statement object through the createStatement method. The typical signature for this method is *createStatement ()* with no arguments.
3. Use the Statement object to obtain an instance of a ResultSet through an *executeQuery (query)* method. This method typically accepts a string (query) where query is a static string SQL.

JDBC Driver Types

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:

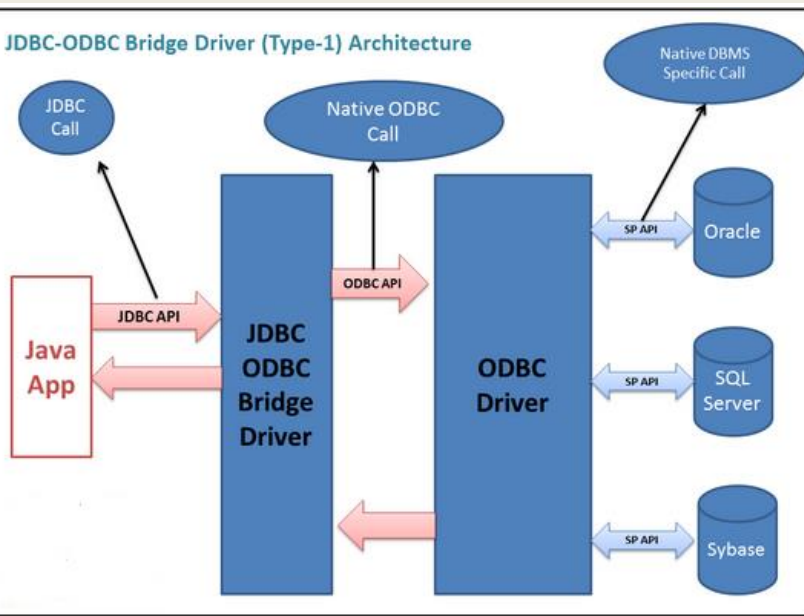
- Type 1 - JDBC-ODBC Bridge Driver
- Type 2 - JDBC-Native API Driver
- Type 3 - Network Protocol Driver
- Type 4 - Native Protocol Driver

Type 1: JDBC-ODBC Bridge Driver

In a **Type 1 driver**, a JDBC bridge is used to access ODBC(Open database Connectivity) drivers installed on each client machine. Using ODBC requires configuring on your system a *Data Source Name (DSN)* that represents the target database.

The JDBC type 1 driver, also known as a JDBC-ODBC Bridge converts JDBC methods into ODBC function calls. Sun has provided JDBC-ODBC Bridge driver, **`sun.jdbc.odbc.JdbcOdbcDriver`**

When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.



Advantage:

- (1) Can Connect to almost any database on any system, for which ODBC driver is installed.
- (2) Easier installation

Disadvantage:

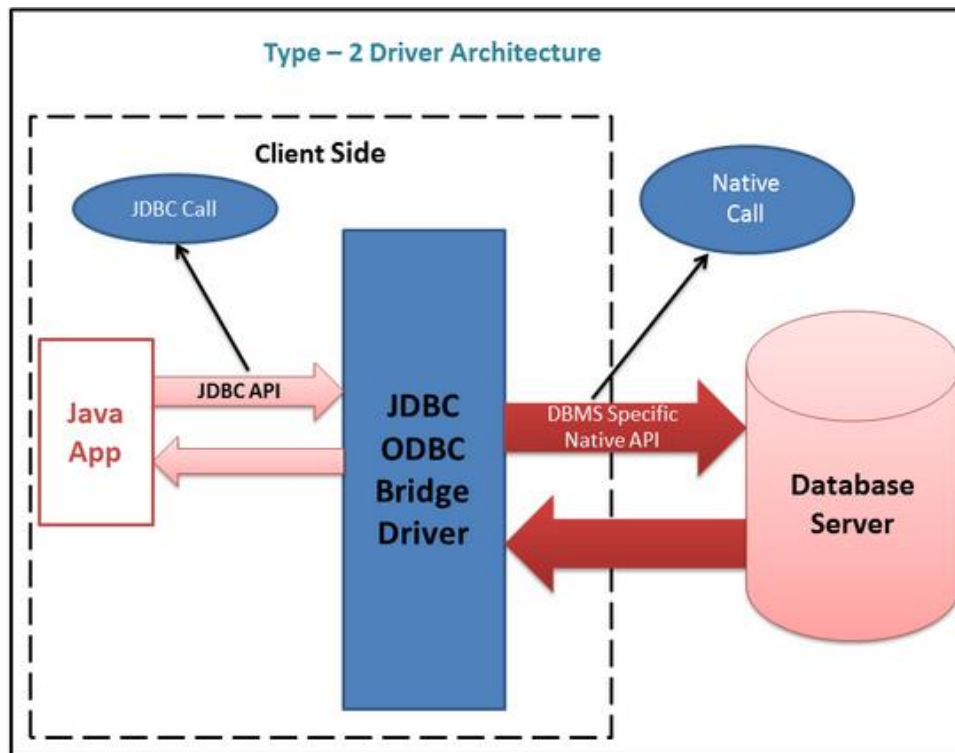
- (1) The ODBC Driver needs to be installed on the client machine.
- (2) This driver is platform dependent as it uses ODBC and ODBC depends on native libraries of the operating system
- (3) Not suitable for applets because the ODBC driver is required on all client machines.

Type 2: JDBC-Native API

In 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 .

The JDBC type 2 driver uses the libraries of the database which are available at client side and this driver converts the JDBC method calls into native calls of the database. Due to this reason this driver is also known as a Native-API driver.

This type of driver is now obsolete

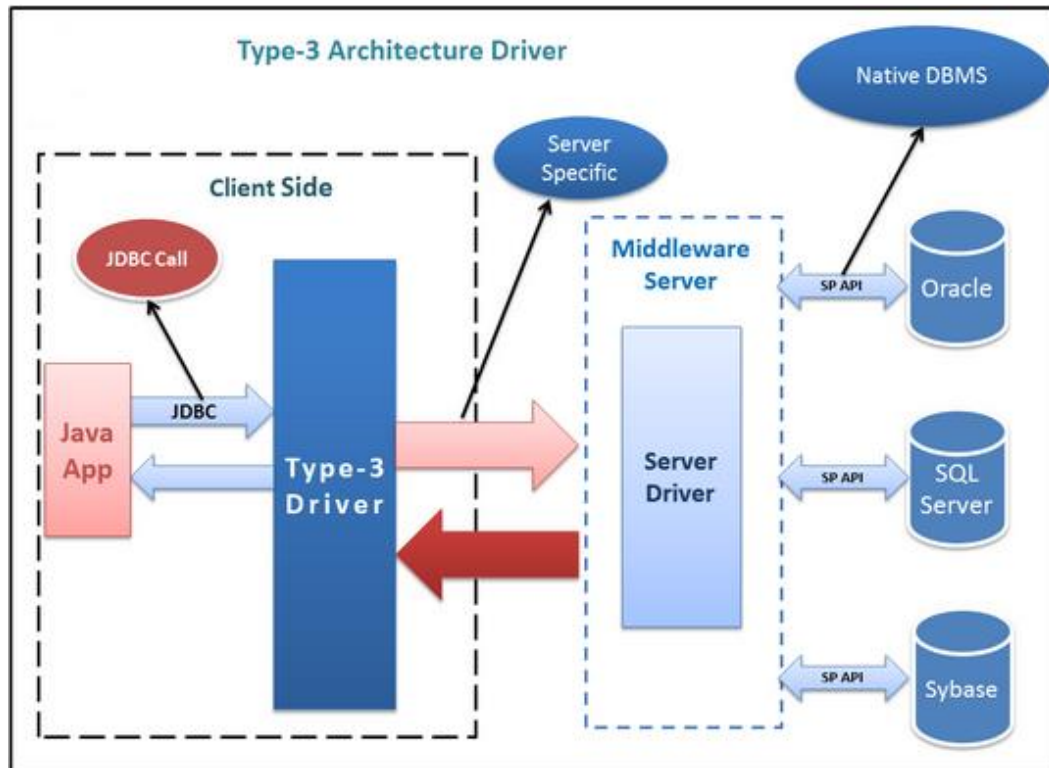


JDBCType 3 driver: Network-protocol Driver

The JDBC type 3 driver is installed in the middle tier(application server) .

This middle tier converts JDBC method calls into the vendor specific database protocols .

Type 3 driver can be used for multiple databases. Due to this reason these drivers are also known as a Network-Protocol drivers.



Advantage:

- (1) There is no need for the vendor database library on the client machine because middleware is database independent and it communicates with client.
- (2) Type 3 driver can be used in any web application as well as on internet also because there is no software required at client side.
- (3) A single driver can handle any database
- (4) The middleware server can also provide the typical services such as connections, auditing, load balancing, logging etc.

Disadvantage:

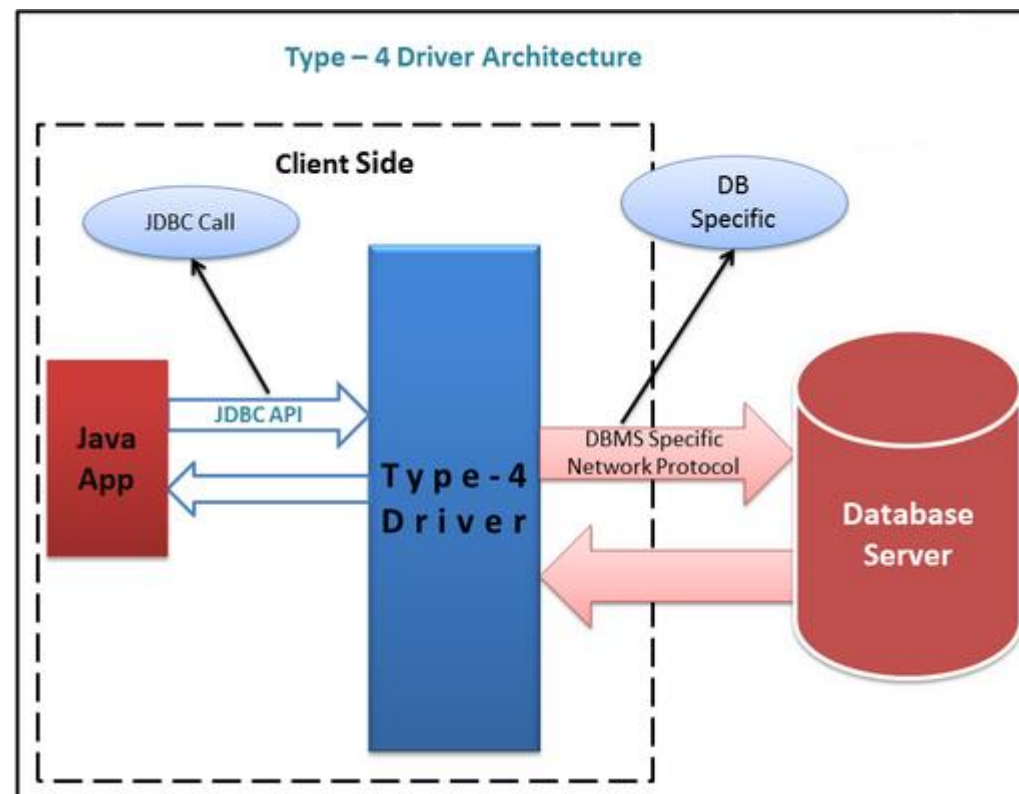
- (1) An Extra layer is needed.

JDBCType 4 driver: Native protocol Driver

Type 4 driver is a pure Java-based driver that communicates directly with 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.



MySQL's Connector/J driver is a Type 4 driver.

Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

Which Driver ?

- If you are accessing only one type of database, such as Oracle, Sybase, or IBM, the preferred driver is **Type 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 drivers are not yet available for your database.
- The **Type 1** driver is not considered as a deployment-level driver and is typically used for development and testing purposes only.

JDBC Type 4 Drivers of well-known Databases

1. **Oracle** : is an object-relational database management system produced and marketed by Oracle Corporation. ***Latest Version is: Oracle 11g***

Oracle JDBC Type 4 driver: ojdbc6.jar : contains classes for use in JDK 1.6
<http://www.oracle.com/technetwork/database/features/jdbc/index-091264.html>

2. **MySQL** : is open source SQL database management system, is developed, distributed, and supported by Oracle Corporation. ***Latest Version is: MySQL5.6.12***

MySQL JDBC Type 4 driver: mysql-connector-java-5.1.25
<http://dev.mysql.com/downloads/mysql/>

3. **JavaDB (Apache Derby)** : is an open source RDBMS implemented entirely in Java . ***Latest Version is: db-derby-10.9.1.0***

JavaDB JDBC Type 4 driver: derby.jar

Download the database from <http://db.apache.org/derby/releases/release-10.9.1.0.cgi>

- derby.jar is the JDBC Type 4 driver provided in <installation_path\ db-derby-10.9.1.0-bin\lib

4. **SQL * Server** : Relational database management system developed by Microsoft. ***Latest Version: SQL Server 2012***

SQL*Server JDBC Type 4 driver: Microsoft JDBC Driver 4.0 is the Official Microsoft JDBC driver for SQL Server ([sqljdbc_4.0.2206.100_enu.exe](#))

<http://www.microsoft.com/en-us/sqlserver/default.aspx>

Type-4 Driver Classes, Port Numbers and URI of well-known databases

Type-4 driver classes of well-known databases

SQL Server : **com.microsoft.jdbc.sqlserver.SQLServerDriver**
Oracle : **oracle.jdbc.OracleDriver**
MySQL : **com.mysql.jdbc.Driver**

Database	Default Port Number	URI (Using Type-4 Drivers)
MySQL	3306	jdbc:mysql://localhost:3306
JavaDB (Derby)	1527	jdbc:derby://localhost:1527
Oracle	1521	jdbc:oracle:thin:@//server:1521/SERVICE_NAME
SQL * Server	1443	jdbc:sqlserver://localhost:1433

The purpose of ports is to uniquely identify different applications or processes running on a single computer and thereby enable them to share a single physical connection to a [packet-switched network](#) like the [Internet](#).

Building a JDBC application

Following are the six steps involved in building a JDBC application:

- **Import the packages** : Include the packages containing the JDBC classes needed for database programming. Most often, *import java.sql.**
- **Register/Loading the JDBC driver** : Initialize a driver so that we can open a communications channel with the database.
- **Open a connection** : Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.
- **Execute a query** : Requires using an object of type Statement for building and submitting an SQL statement to the database.
- **Extract data from result set** : Use appropriate *ResultSet.getXXX()* method to retrieve the data from the result set.
- **Clean up the environment** . Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

Loading the type 4 driver:

```
Class.forName("Driver_classname");
```

Ex.

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

The class Loader of JVM loads the specified Driver class into the memory at run-time.

DriverManager class

DriverManager class -

- Manages all the JDBC Drivers that are loaded in the memory
- Helps in dynamic loading of Drivers

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

Methods in *DriverManager* class -

- getConnection() : to establish a connection to a database.
 - Connection getConnection(String url)
 - Connection getConnection(String url, String userID, String password)
- registerDriver(java.sql.Driver)

```
Connection connection=  
DriverManager.getConnection("jdbc:mysql://localhost:3306/electronics","root","root");
```


DriverManager class: Connecting through Type 4 drivers

Connecting to MySQL database

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

Class Loader of JVM loads the specified Driver class into the memory at run-time

```
Connection connection=
```

```
DriverManager.getConnection("jdbc:mysql://localhost:3306/tutorial","root","root");
```

password

URI

User-id

Establishes connection to the specified database and returns Connection object

Connecting to Oracle database

```
Class.forName("oracle.jdbc.OracleDriver"); // Load the Driver class
```

```
Connection connection =
```

```
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl","scott","tiger");
```

Port number

SID
(system identifier)

Connection Interface

④ **Connection interface** - defines methods for interacting with the database via the established connection.

- A connection object represents a connection with a database.
- A connection session includes the SQL statements that are executed and the results that are returned over that connection.
- A single application can have one or more connections with a single database, or it can have many connections with many different databases.

We obtain the connection object by the following statement:

```
Connection connection =  
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl","scott","tiger");
```

Connection Interface methods

● The different methods of Connection interface are:

- `close()` - closes the database connection
- `createStatement()` - creates an SQL Statement object
- `prepareStatement()` - creates an SQL PreparedStatement object.
(PreparedStatement objects are precompiled SQL statements)
- `prepareCall()` - creates an SQL CallableStatement object using an SQL string.
(CallableStatement objects are SQL stored procedure call statements)

```
Statement statement = connection.createStatement(); //Creates a SQL statement
```

Statement Interface

Statement

- ④ A statement object is used to send SQL statements to a database.
- ④ Three kinds :
 - **Statement**
 - Execute simple SQL without parameters
 - **PreparedStatement**
 - Used for pre-compiled SQL statements with or without parameters
 - **CallableStatement**
 - Execute a call to a database stored procedure or function

PreparedStatement and **CallableStatement** are the interfaces that extend **Statement** interface

Statement Interface methods

- **Statement interface** - defines methods that are used to interact with database via the execution of SQL statements.
- The different methods are:
 - **executeQuery(String sql)** - executes an SQL statement (SELECT) that queries a database and returns a ResultSet object.
 - **executeUpdate(String sql)** - executes an SQL statement (INSERT, UPDATE, or DELETE) that updates the database and returns an int, the row count associated with the SQL statement
 - **execute(String sql)** - executes an SQL statement that is written as String object
 - **getResultSet()** - used to retrieve the ResultSet object

OracleConnection Utility class

Java Project : JDBCProject

```
public class OracleConnection {  
    static{  
        try {  
            Class.forName("oracle.jdbc.OracleDriver");  
        } catch (ClassNotFoundException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
    }  
    public static Connection getConnection(){  
        try {  
            Connection connection =  
DriverManager.getConnection("jdbc:oracle:thin:@//localhost:1521/orcl","scott","tiger");  
            return connection;  
        } catch (SQLException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
            return null;  
        }  
    }  
}
```

**After creating the java project,
load the JDBC Type 4 jar file into project's
class path.**

Checking the connectivity to oracle database

```
public class OracleConnectionTester {  
  
    public static void main(String[] args) {  
        Connection connection=null;  
        try{  
            connection = OracleConnection.getConnection();  
            if(connection!=null){  
                System.out.println("Connection is successful");  
            }else{  
                System.out.println("Unable to connect");  
            }  
        }  
        catch(Exception exception){  
            exception.printStackTrace();  
        }  
    }  
}
```

Java Project : JDBCProject

```
        finally{  
            try {  
                if(connection!=null){  
                    connection.close();  
                }  
            } catch (SQLException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

ResultSet Interface

🔗 **ResultSet Interface** - maintains a pointer to a row within the tabular results.

The **next()** method is used to successively step through the rows of the tabular results.

🔗 The different methods are:

- **getBoolean(int)** - Get the value of a column in the current row as a Java boolean.
- **getByte(int)** - Get the value of a column in the current row as a Java byte.
- **getDouble(int)** - Get the value of a column in the current row as a Java double.
- **getInt(int)** - Get the value of a column in the current row as a Java int.

ResultSet Interface

program

Result set(Memory)

Table (Hard Drive)

1	Smith	New York
2	Clark	Sydney
3	Ravi Kumar	Mumbai
.	.	.
.	.	.

```
ResultSet resultSet= statement.executeQuery("select * from tutorial.city");
System.out.println("Sno   Name   City");
while(resultSet.next()){
    System.out.print(resultSet .getInt(1)+ " "); // resultSet.getInt("Sno");
    System.out.print(resultSet .getString(2)+ " ");
    System.out.print(resultSet .getString(3));
    System.out.println(" ");
}
```

- A ResultSet object maintains a cursor pointing to its current row of data.
- Initially the cursor is positioned before the first row.
- The next() method moves the cursor to the next row, and because it returns false when there are no more rows in the ResultSet object, it can be used in a while loop to iterate through the result set.
- The ResultSet interface provides *getter* methods (getBoolean, getLong, and so on) for retrieving column values from the current row. Values can be retrieved using either the index number of the column or the name of the column.

Adding JDBC Type-4 driver to your project in Eclipse IDE

- Right-Click on the project name
- Click on **Properties**
- In the **Properties for *Project Name* window**, Select **Java Build Path**
- Click on **Libraries Tab**
- Click on **Add External JARs... button**
- Opens **File dialog box**, open the folder that contains **JDBC driver jar** file, **select the file** and click on **Open button**
- You will observe **JDBC driver jar** file placed in the **Libraries** tab, click on **OK button**
- In the Package Explorer, Under your Project Name, you will find JDBC jar file in **Referenced Libraries**

Type-4 driver Example : Querying table

Demo Activity

Java Project: JDBCProject

```
public class EmployeeDetails {
public static void main(String [] args) throws Exception{
Connection connection=null; ResultSet resultSet=null;
try{
connection = MySqlConnection.getConnection();
Statement statement = connection.createStatement(); //Create a SQL statement
resultSet = statement.executeQuery("select * from employee"); // Execute the query
System.out.println("Empno  Ename  Job  Hiredate  Salary  Commission  Deptno");
while(resultSet.next()){
    System.out.print(resultSet.getInt(1)+ " ");
    System.out.print(resultSet.getString(2)+ " ");
    System.out.print(resultSet.getString(3)+" ");
    System.out.print(resultSet.getDate(4)+" ");
    System.out.print(resultSet.getDouble(5)+" ");
    System.out.print(resultSet.getDouble(6)+" ");
    System.out.println(resultSet.getInt(7)+" ");
    System.out.println(" ");
}
}
catch(SQLException exception){ exception.printStackTrace();}
finally{
if(connection != null){
connection.close(); resultSet.close();}}}}
```

statement.executeUpdate method

Demo Activity

Java Project: JDBCProject

Class name : UpdateEmployee

.....

```
int count= statement.executeUpdate("delete from employee where empno=1014");  
System.out.println("No. of rows deleted: "+ count);
```

```
int count= statement.executeUpdate("insert into employee  
values(1015,'James','PROGRAMMER','05-SEP-2010',4500.0,null,10)");  
System.out.println("No. of rows inserted: "+ count);
```

```
int count= statement.executeUpdate("update employee set comm=100.00 where comm  
is null");  
System.out.println("No. of rows updated: "+ count);
```

.....

//in the catch block,

Note: Oracle date format is : DD-MON-YYYY

Commit and Rollback

Transaction Management using JDBC

- ❶ By default, auto commit mode of the connection reference is set to true
- ❷ A transaction can be done as follows using methods of the Connection interface

```
...  
connection.setAutoCommit(false);    //by default it is true  
  
try{  
    //Statements  
    connection.commit();  
}  
catch(Exception exception){  
    connection.rollback();  
}
```

statement.executeUpdate method

Demo Activity

Java Project: JDBCProject

Class name : UpdateEmployee

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

int count= statement.executeUpdate("update employee set salary =salary+0.10*salary");
System.out.println("No. of rows updated: "+ count);

int count= statement.executeUpdate("delete from employee where empno=1014");
System.out.println("No. of rows deleted: "+ count);

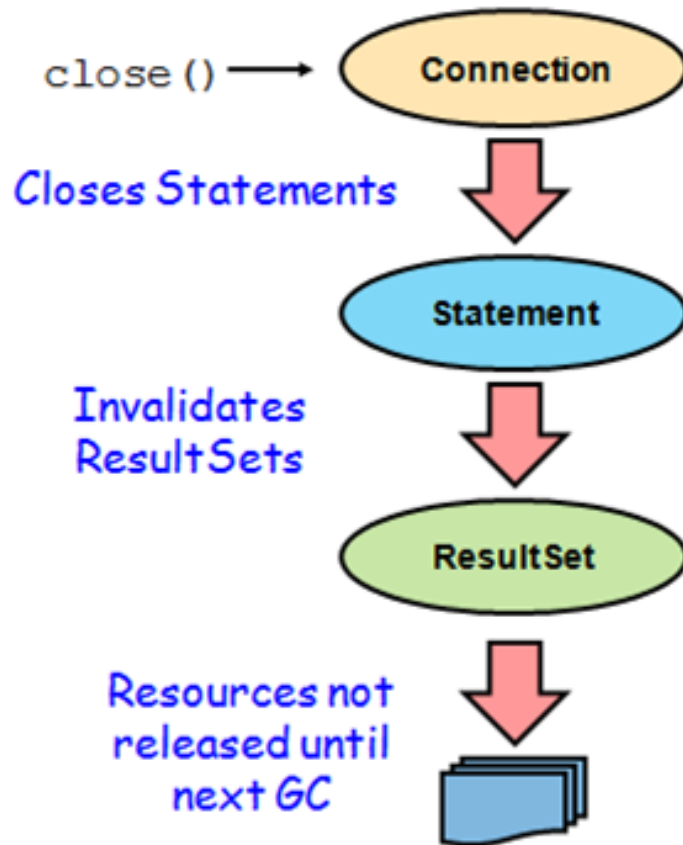
int count= statement.executeUpdate("insert into employee values(1020,'Jim','CLERK','15-SEP-
2012',1500.0,null,10)");
System.out.println("No. of rows inserted: "+ count);

connection.commit();
}
.....
// In the catch block, connection.rollback();
.....
```

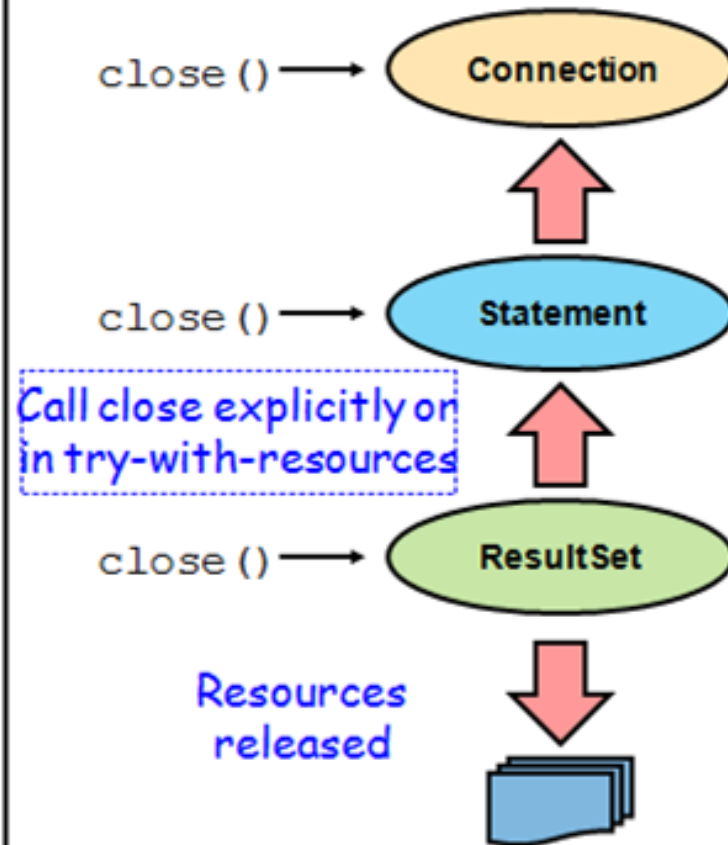
Closing JDBC objects

Closing JDBC Objects

One Way



Better Way



PreparedStatement Interface

- *PreparedStatement interface* -- helps us to work with *precompiled SQL statements*
- Precompiled SQL statements are *faster* than normal statements
- So, if a SQL statement is to be repeated, it is better to use PreparedStatement
- Some values of the statement can be represented by a *?* character which can be replaced later using *setXXX* method

Example:

```
String sql = "SELECT * FROM tutorial.employee WHERE empno = ?" ;  
PreparedStatement preparedStatement = connection.prepareStatement(sql);  
preparedStatement.setInt(1,4); // here 1 indicates 1st placeholder and 4 indicates empno  
resultSet = preparedStatement.executeQuery();
```


PreparedStatement example

Demo Activity

Java Project: JavaJDBCProject

```
.....  
String str= "SELECT * FROM EMPLOYEE where salary > ? and job =?";  
Connection connection=null;  
PreparedStatement preparedStatement = null;  
ResultSet resultSet =null;  
try{  
connection = MySqlConnection.getConnection();  
preparedStatement = connection.prepareStatement(str);  
preparedStatement.setDouble(1,2000.50);  
preparedStatement.setString(2, "DEVELOPER");  
resultSet = preparedStatement.executeQuery();  
  
if(resultSet != null{  
    // display employee details  
}  
}
```

.....

Scrollable and Updatable ResultSet

- A default ResultSet object is not updatable and has a cursor that moves forward only. Thus, we can iterate through it only once and only from the first row to the last row.
- It is possible to produce ResultSet objects that are scrollable and/or updatable.
- The following code fragment illustrates how to make a result set that is scrollable and insensitive to updates by others, and that is updatable.

```
// ResultSet object will be scrollable, will not show changes made by others and will be updatable
```

```
Statement stmt = connection.createStatement(  
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

Scrollable and Updatable ResultSet

- A set of updater methods were added to this interface in the JDBC 2.0 API
- The updater methods may be used in two ways:
 1. to update a column value in the current row. In a scrollable ResultSet object, the cursor can be moved backwards and forwards, to an absolute position, or to a position relative to the current row.
- The following code fragment updates the NAME column in the fifth row of the ResultSet object rs and then uses the method ***updateRow()*** to update the data source table from which rs was derived.

Do not
use *

```
resultSet=statement.executeQuery("select empno,ename,job,salary,deptno from  
employee");
```

```
resultSet.absolute(5); // moves the cursor to the fifth row of rs
```

```
// updates the NAME column of row 5 to be Ravi Kumar
```

```
resultSet.updateString("ENAME", "Ravi Kumar");
```

```
resultSet.updateRow(); // updates the row in the data source
```

Scrollable and Updatable ResultSet

2. To insert column values into the insert row. An **updatable ResultSet object** has a *special row associated* with it that serves as a **staging area for building a row to be inserted**.

The following code fragment moves the cursor to the insert row, builds a three-column row, and inserts it into rs and into the data source table using the method *insertRow()*.

```
rs.moveToInsertRow(); // moves cursor to the insert row
// updates the first column of the insert row to be David Smith
rs.updateString(1, "David Smith");
rs.updateInt(2,35); // updates the second column to be 35
rs.updateBoolean(3, true); // updates the third column to true
rs.insertRow();
rs.moveToCurrentRow();
```

ResultSetMetaData

Demo Activity

To access the meta data information of database objects, use ResultSetMetaData interface.

Create instance of ResultSetMetaData as shown below :

```
ResultSetMetaData resultSetMetaData=resultSet.getMetaData();
```

By invoking the instance methods , we can access information such as column names, number of columns etc.

```
.....  
ResultSetMetaData resultSetMetaData = resultSet.getMetaData();  
int count=resultSetMetaData.getColumnCount();  
  
System.out.println("No. of columns in employee table: "+ count);  
System.out.println("Column Names...");  
for(int i=1;i<=count;i++){  
System.out.println(resultSetMetaData.getColumnName(i));  
}
```

CallableStatement Interface

- **CallableStatement interface** -- helps us to call stored procedures and functions

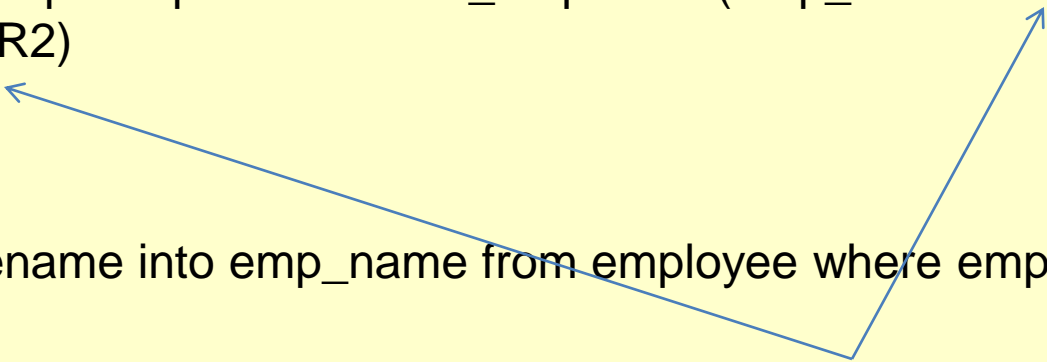
```
CallableStatement callableStatement = connection.prepareCall("execute proc ?");  
callableStatement.setInt(50);  
callableStatement.execute();
```



In Oracle

Creating stored procedure in Oracle

```
create or replace procedure PR_empName(emp_id IN NUMBER,emp_name OUT  
VARCHAR2)  
AS  
BEGIN  
    select ename into emp_name from employee where empno=emp_id;  
END; /
```

Two blue arrows originate from the parameter declarations in the SQL code. One arrow points from 'emp_id IN NUMBER' to the 'IN' parameter mode definition section. The other arrow points from 'emp_name OUT VARCHAR2' to the 'OUT' parameter mode definition section.

Formal parameter types are unconstrained in oracle

Parameter Modes:

IN (is the default mode) : IN indicates that a parameter can be passed into stored procedures but any modification inside stored procedure does not change parameter

OUT : indicates that stored procedure can change this parameter and pass back to the calling program.

IN OUT : We can pass parameter into stored procedure and get it back with the new value from calling program

The syntax of defining a parameter in stored procedure is as follows:

MODE parameter_name parameter_mode parameter_type

Another Example : stored procedure in Oracle

```
create or replace procedure get_emp_details(p_empno IN number, p_name OUT varchar2,  
    p_desig OUT varchar2,p_sal OUT number )  
AS  
begin  
select ename,job,sal into p_name,p_desig,p_sal  
from emp where empno = p_empno;  
end//
```


Calling Oracle procedure from a java program

```
String call = "{ call PR_empName(?,?)}";  
CallableStatement cstmt = connection.prepareCall(call);  
cstmt.setInt(1, id);  
cstmt.registerOutParameter(2, oracle.jdbc.OracleTypes.VARCHAR);  
cstmt.executeQuery();  
name = cstmt.getString(2);
```

Stored function in Oracle

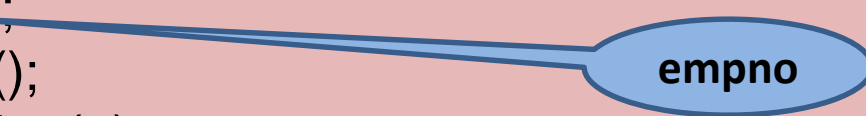
```
create or replace function getEmployeeTotalPay(p_empno IN number)
return number
as
v_total number:=0;
begin
select sal+nvl(comm,0) into v_total
from emp where empno=p_empno;
return v_total;
exception
when others then
    return 0;
end;
/
```

Creating stored function in Oracle

```
create or replace function get_empName(emp_id IN NUMBER) RETURN  
VARCHAR2  
AS  
    emp_name employee.ename%type;  
BEGIN  
    select ename into emp_name from employee where empno=emp_id;  
    return emp_name;  
end;/
```

Calling Oracle function from a java program

```
String call = "{ ? := call get_empName(?)}";  
CallableStatement cstmt = connection.prepareCall(call);  
cstmt.registerOutParameter(1, oracle.jdbc.OracleTypes.VARCHAR);  
cstmt.setInt(2, 1010);  
cstmt.executeQuery();  
name = cstmt.getString(1);
```



The diagram illustrates the mapping of the Java code to the Oracle function. An arrow originates from the integer value 1010 in the `cstmt.setInt(2, 1010);` line and points to a blue oval containing the text `empno`, indicating that the value 1010 is being passed as the `emp_id` parameter to the `get_empName` function.

Calling stored function in Oracle

```
private static double getEmployeeTotalPay(int empid) {
String sql="{call ? :=getEmployeeTotalPay(?)}";
try(Connection connection=MyOracleConnection.getConnection();
CallableStatement callableStatement=connection.prepareCall(sql)
){
callableStatement.registerOutParameter(1,
oracle.jdbc.OracleTypes.NUMBER);
callableStatement.setInt(2, empid);
callableStatement.executeQuery();
System.out.println("Total Pay= "+callableStatement.getDouble(1));

}
catch(SQLException e){
e.printStackTrace();
}
catch(Exception e){
e.printStackTrace();
}
return 0;
}
```

Converting java.util.Calendar to java.sql.Date object

Converting java.util.GregorianCalendar object to java.sql.Date object:

Ex. `java.util.Calendar hdate = new java.util.GregorianCalendar(2014,2,15);`

1. Convert java.util.Calendar to java.util.Date

`java.util.Date mdate = hdate.getTime();`

2. Convert java.util.Date to java.sql.Date

`java.sql.Date ndate = new java.sql.Date(mdate.getTime());`

Converting java.sql.Date to java.util.Calendar object

// convert java.sql.Date to java.util.Date

`java.sql.Date sdate = resultSet.getDate("birthdate");`

`java.util.Date udate = new java.util.Date(sdate.getTime());`

OR

`java.util.Date date = java.sql.Date.valueOf(LocalDate);`

// convert java.util.Date to java.util.Calendar

`java.util.Calendar cdate = Calendar.getInstance();`

`cdate.setTime(udate);`

java.time.LocalDate to java.sql.Date and vice-versa

java.time.LocalDate of JDK 8

To convert from LocalDate to java.sql.Date :

```
LocalDate localDate= Local.of(2001,10,16);  
java.sql.Date.valueOf( localDate );
```

To convert from java.sql.Date to LocalDate:

```
sqlDate.toLocalDate();
```



Thank You!