MODULE-IV

Annotations and JDBC

By:
Shivakumara T
Assistant Professor,
Department of MCA,
BMSITM, Bangalore- 560064

The following Topics will be discussing in this Module

1. Built-in Annotations with examples
2. custom annotation
3. Talking to Database
4. Immediate Solutions
5. Essential JDBC program
6. using prepared Statement Object
7. Interactive SQL tool
8. JDBC in Action
9. Result sets
10. Batch updates
11. Mapping,
12. Basic JDBC data types
13. Advanced JDBC data types
14. immediate solutions

Built-in Annotations with examples

Annotation

- Annotations are metadata or *data about data*. An annotation indicates that the declared element should be processed in some special way by a compiler, development tool, deployment tool, or during runtime.

- Annotation-based development is certainly one of the latest Java development trends

- Annotations can be applied to several Java Elements like,
  - package declarations,
  - class,
  - constructors,
  - methods,
  - fields,
  - Variables and etc.

Built-in Annotations with examples

Annotation

- The goal: Allow the programmer to provide additional information about the program – This information can be used by software engineering tools
- An annotation is a type – defined using an interface-like syntax
- An annotation can be specified whenever a modifier is allowed – Convention: before the public/static modifiers
- Annotations do not affect semantics of the class – But may affects semantics of things using the class (tools, code generation, runtime options, etc.)

Built-in Annotations with examples

The Basics

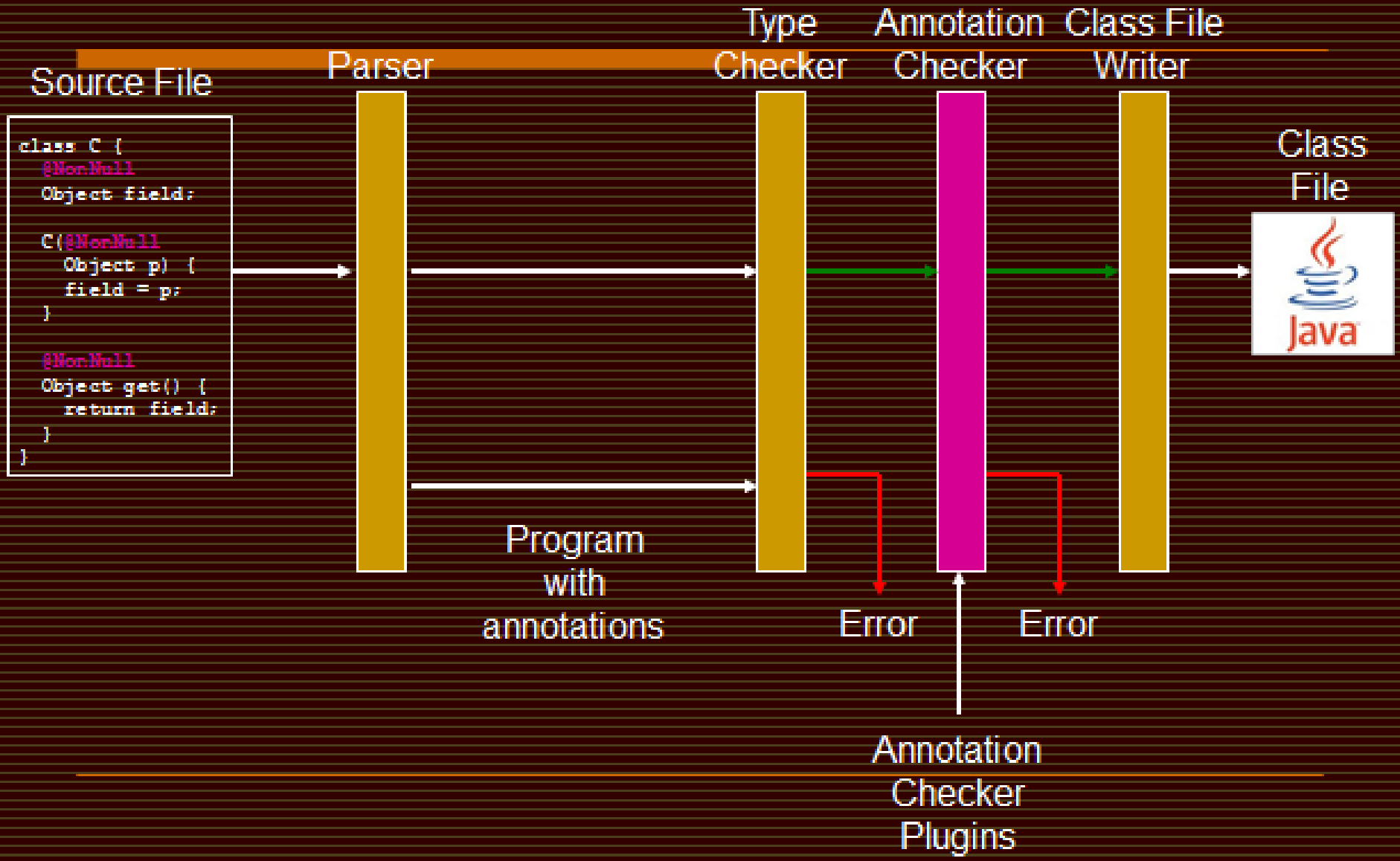
- **Example to Define an Annotation (Annotation type)**

```
public @interface MyAnnotation {
    String doSomething();
}
```

- **Example to Annotate Your Code (Annotation)**

```
MyAnnotation (doSomething="What to do")
public void mymethod() {
    .....
}
```

Structure of Java5 Compiler



custom annotation

Predefined Annotations

- @Override – Assert intention to override a method in a superclass – Compiler fails if not actually overriding
- Checks spelling, override vs. overload
- @Deprecated – Indicates that an element should not be used
- @SuppressWarning – Tells the compiler to suppress specific warnings

Annotation Types

- Marker
- Single-Element
- Full-value or multi-value

Marker

Marker annotations take no parameters. They are used to mark a Java element to be processed in a particular way.

- **Example:**

```
public @interface MyAnnotation {
}
```

- **Usage:**

```
@MyAnnotation
public void mymethod() {
        ....
}
```

Single-Element

- Single-element, or single-value type, annotations provide a single piece of data only. This can be represented with a data=value pair or, simply with the value (a shortcut syntax) only, within parenthesis.

- **Example:**

```
public @interface MyAnnotation {
 String doSomething();
 }
```

- **Usage:**

```
@MyAnnotation ("What to do")
 public void mymethod() {
 ....
 }
```

Full-value or multi-value

- Full-value type annotations have multiple data members.

- **Example:**

```
public @interface MyAnnotation {
  String doSomething();
   int count;
   String date();
 }
```

- **Usage:**

```
@MyAnnotation (doSomething=
      "What to do",
       count=1,
        date="09-09-2005")
public void mymethod() {
      ....      }
```

The Built-In Annotations

- Java defines seven built-in annotations.
- Four are imported from java.lang.annotation
  - @Retention,
  - @Documented,
  - @Target,
  - and @Inherited.
- Three are included in java.lang.
  - @Override,
  - @Deprecated,
  - and @SuppressWarnings.

Simple Annotations (or) Standard Annotations

- There are three types of simple annotations provided by JDK5. They are:
  - Override
  - Deprecated
  - Suppresswarnings

Override

- Override is a Marker Annotation type that can be applied to a method to indicate to the Compiler that the method overrides a method in a Superclass. This Annotation type guards the programmer against making a mistake when overriding a method. For eg The syntax ---@Override
- Example Program:

```
class Parent {
    public float calculate (float a, float b) {
        return a * b;
    }
}
```

Whenever you want to override a method, declare the Override annotation type before the method:

```
public class Child extends Parent {
    @Override
    public int calculate (int a, int b) {
        return (a + 1) * b;
    }
}
```

The Deprecated annotation

- This annotation indicates that when a deprecated program element is used, the compiler should warn you about it. Example 2 shows you the deprecated annotation.
- The syntax --- @Deprecated

- Example :

```
public class DeprecatedTest{
  @Deprecated
  public void serve() {

  }

}
```

If you use or override a deprecated method, you will get a warning at compile time.

```
public class DeprecatedTest2{
        public static void main(String[] args) {
            DeprecatedTest test = new DeprecatedTest();
            test.serve();
        }
}
```

The Suppresswarnings annotation

- SuppressWarnings is used to suppress compiler warnings. You can apply @SuppressWarnings to types, constructors, methods, fields, parameters, and local variables.
- The syntax --- @SuppressWarnings
- **Eg:**

```
import java.util.Date;

public class Main {
    @SuppressWarnings(value={"deprecation"})
    public static void main(String[] args) {
        Date date = new Date(2009, 9, 30);

        System.out.println("date = " + date);
    }
```

Documentation

- public class Generation3List extends {
  // Author: John Doe
  // Date: 3/17/2002
  // Current revision: 6
  // Last modified: 4/12/2004
  // By: Jane Doe
  // Reviewers: Alice, Bill, Cindy
  // class code goes here
  }

- Using Java Annotation
  - @Documented
    @interface ClassPreamble {
    String author();
    String date();
    int currentRevision() default 1;
    String lastModified() default "N/A";
    String lastModifiedBy() default "N/A";
    String[] reviewers(); }

```
@ClassPreamble (
        author = "John Doe",
        date = "3/17/2002",
        currentRevision = 6,
        lastModified = "4/12/2004",
        lastModifiedBy = "Jane Doe",
        reviewers = {"Alice", "Bob", "Cindy"}
)
```

The Target annotation

- @Target(ElementType.TYPE)—can be applied to any element of a class
- @Target(ElementType.FIELD)—can be applied to a field or property
- @Target(ElementType.METHOD)—can be applied to a method level annotation
- @Target(ElementType.PARAMETER)—can be applied to the parameters of a method
- @Target(ElementType.CONSTRUCTOR)—can be applied to constructors
- @Target(ElementType.LOCAL_VARIABLE)—can be applied to local variables
- @Target(ElementType.ANNOTATION_TYPE)—indicates that the declared type itself is an

JDBC Object

- Practically every J2EE application saves, retrieves, and manipulates information stored in a database using web services provided by a J2EE component.

- A J2EE component supplies database access using **Java data Objects** contained in the JDBC application programming interface (API).

- Java data objects have methods that open a connection to a database management system (DBMS) and then transmit messages (queries) to insert, retrieve, modify or delete data stored in a database.

The Concept of JDBC:

One of the major obstacles for Sun Microsystems, Inc. to Overcome was a language barrier.

Each DBMS defined its own low-level way to interact with programs to access data stored in its databases.

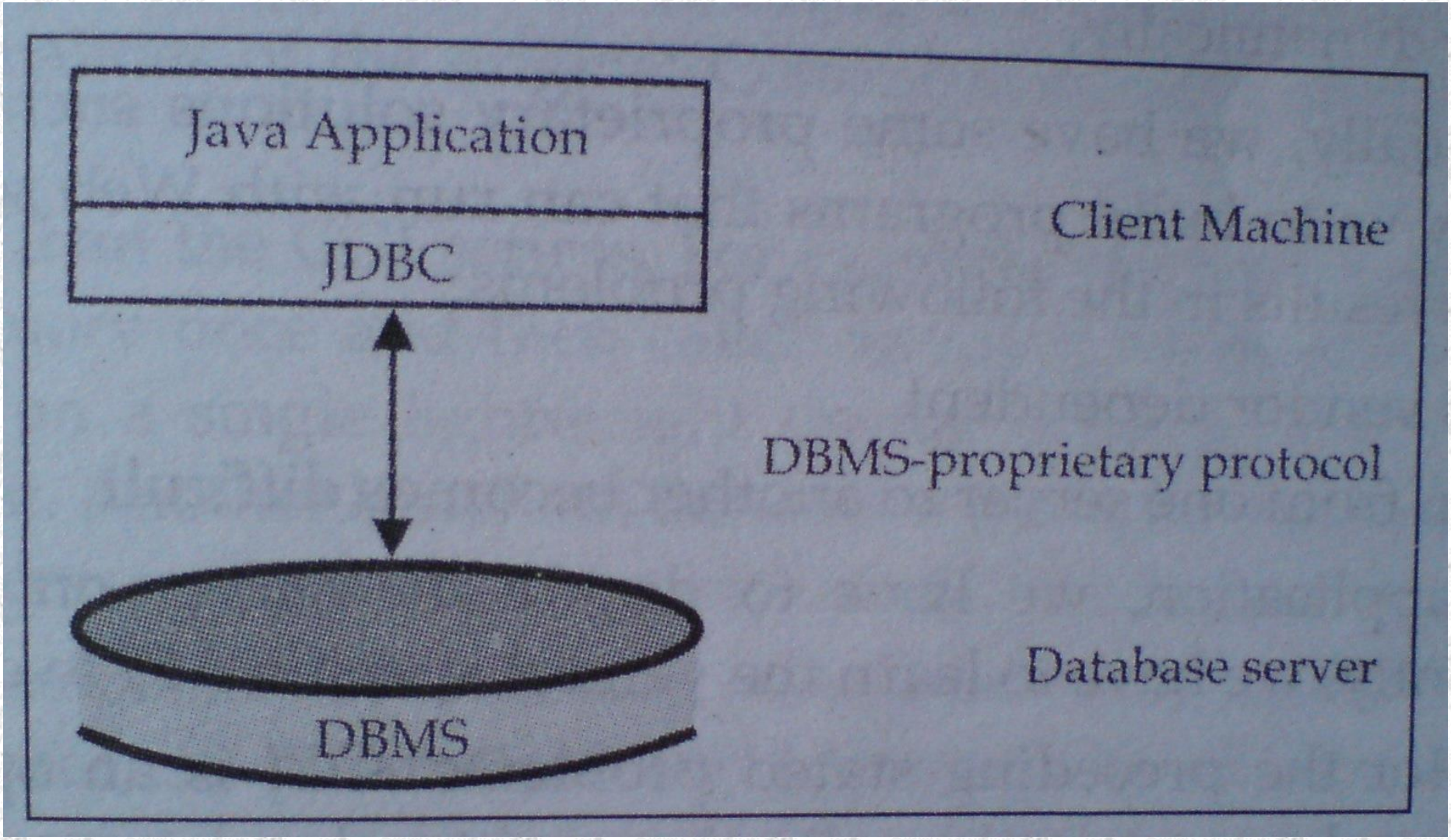
This meant low-level code written to communicate with an Oracle database might need
To be rewritten to access a DB2.

The JDBC driver developed by Sun Microsystems, Inc. wasn't a driver at all. It was a specification that described the detail functionality of a JDBC driver.

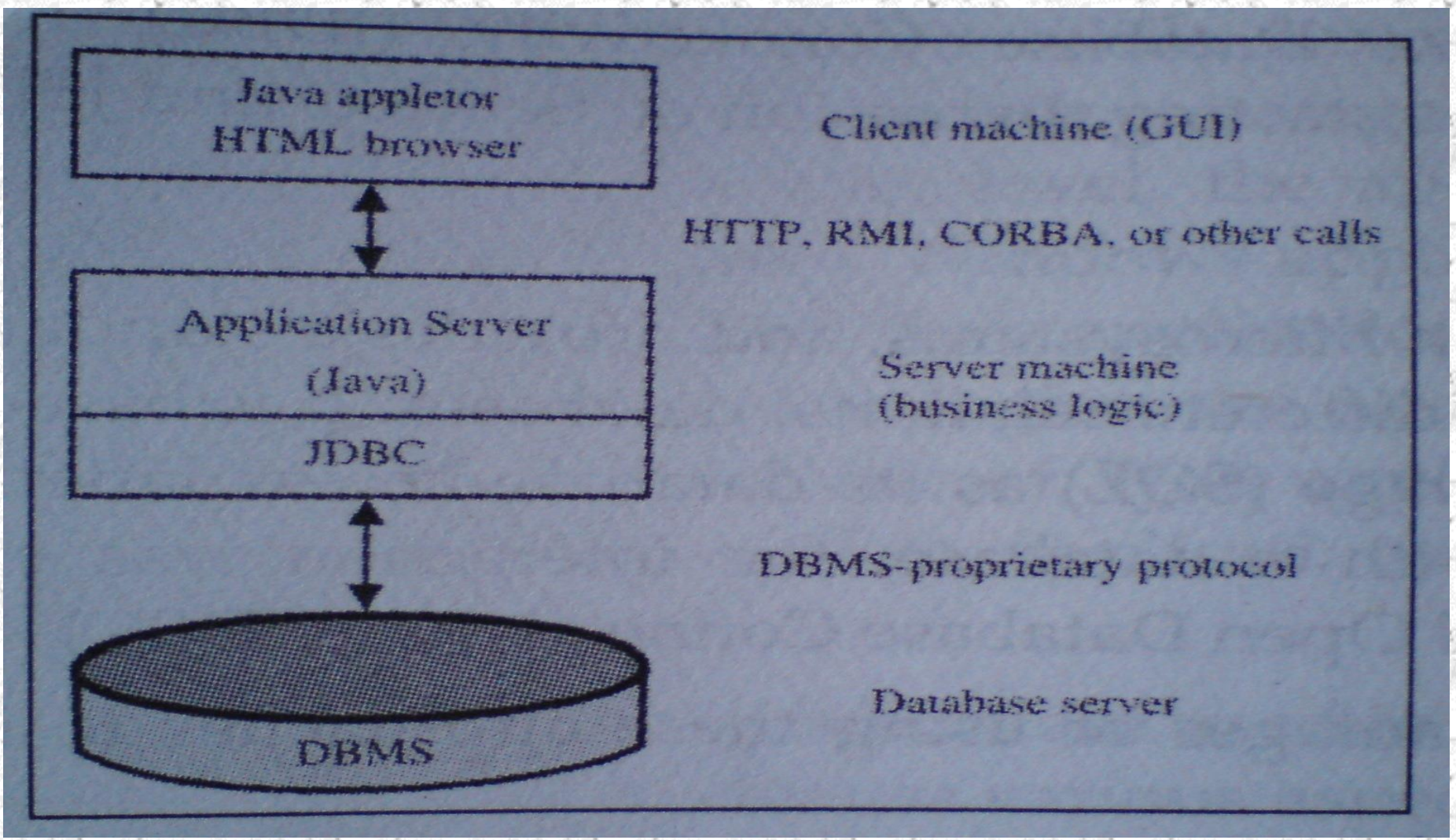
The specifications required a JDBC driver to be a translator that converted low-level proprietary DBMS messages to low-level messages understood by the JDBC API, and Vice versa.

Java programmers could use high-level Java Data Objects defined in the JDBC API to write a routine that interacted with the DBMS.

JDBC 2-tier Architecture



JDBC 3-tier Architecture



JDBC Driver types:

JDBC driver specification classifies JDBC drivers into four groups. Each group is referred to as a JDBC driver type.

Type1 JDBC-to-ODBC Driver:

Also called the JDBC/ODBC Bridge, is used to translate DBMS calls between the JDBC specification and ODBC specification.

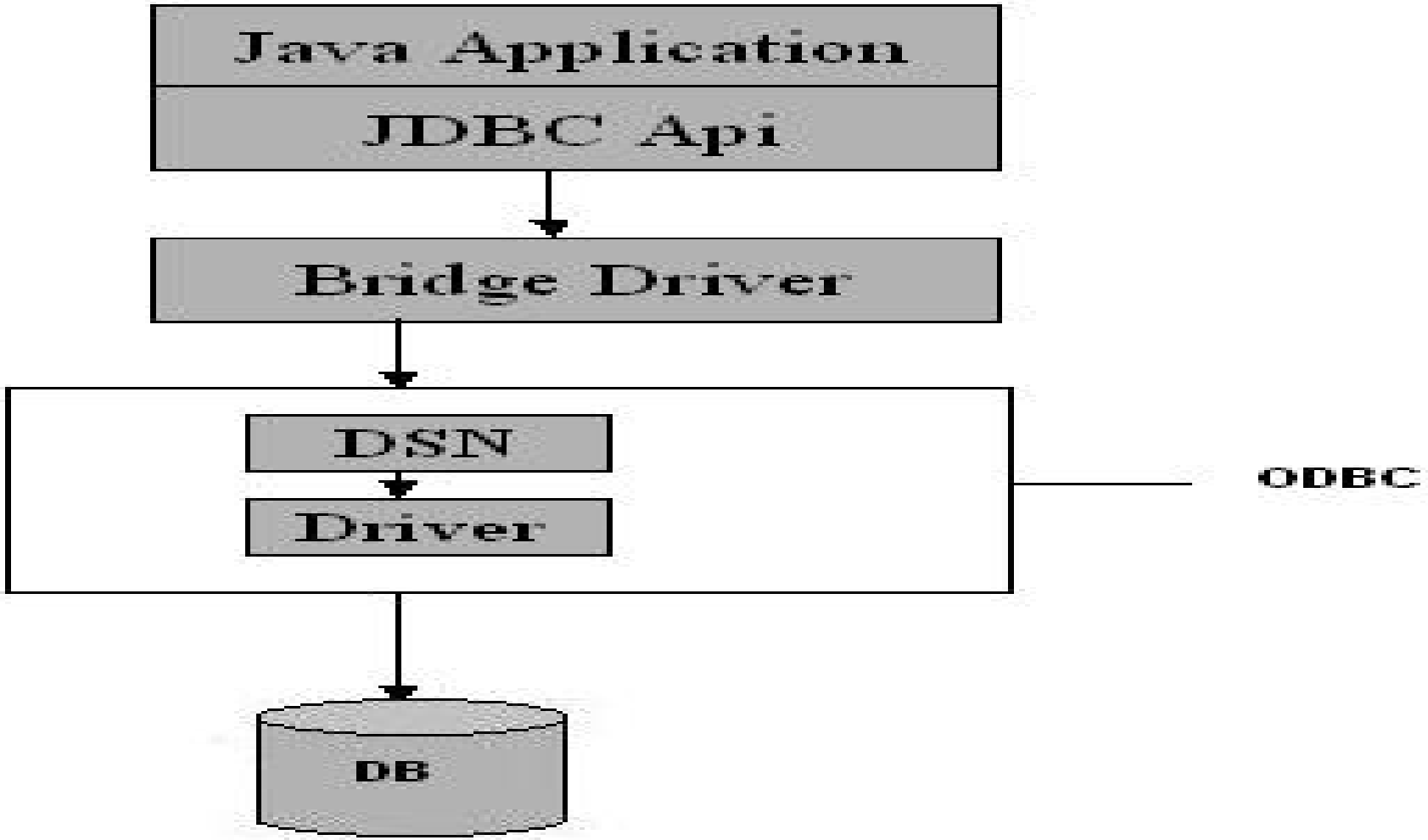
The JDBC-to-ODBC driver receives messages from J2EE component that conforms to the JDBC specification.

Those messages are translated by the JDBC-to-ODBC driver into the ODBC message format, which is then translated into the message format understood by the DBMS

Imp. Note:

Avoid using the JDBC/ODBC bridge in a mission-critical application because the extra translation might negatively impact performance

Type 1 JDBC Driver/JDBC-ODBC Bridge driver



Type 2 Java/Native code Driver:

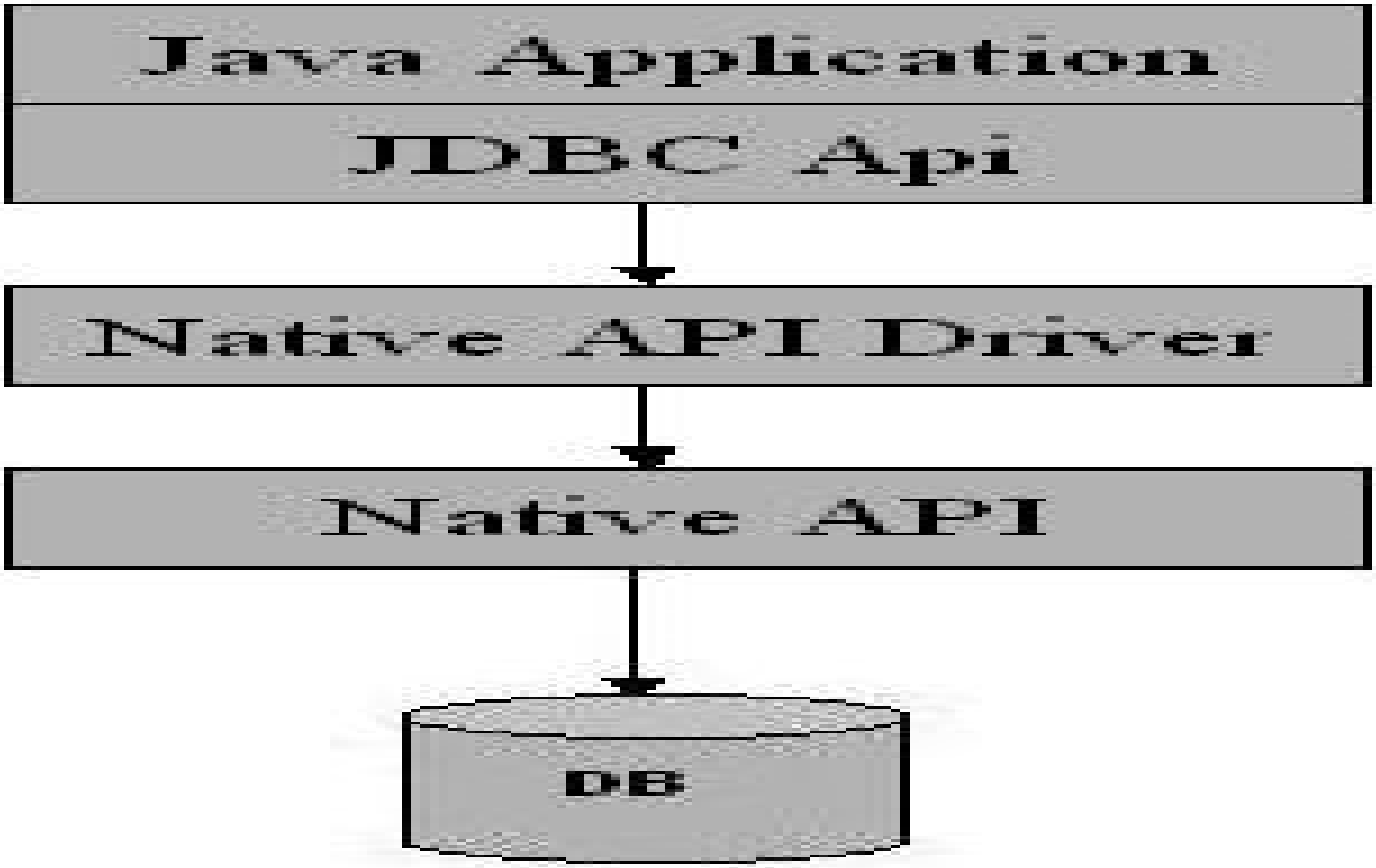
This driver uses Java classes to generate platform-specific code - i.e., code only understood by a specific DBMS.

The manufacturer of the DBMS provides both the Java/Native Code driver and API classes so the J2EE component can generate the platform- specific code.

Imp. Note:

Disadvantage of this driver is the loss of some portability of code. The API classes for this driver probably won't work with another manufacturer's DBMS.

Type 2 JDBC Driver/Native-API/partly Java driver



Examples of Type -2 Driver

- OCI(Oracle Call Interface) Driver
- Web logic OCI Driver for oracle
- Type -2 driver for Sybase

Type 3 JDBC Driver:

It is also referred to as the Java Protocol, is the most commonly used JDBC driver.

It converts SQL queries into JDBC-formatted statements.

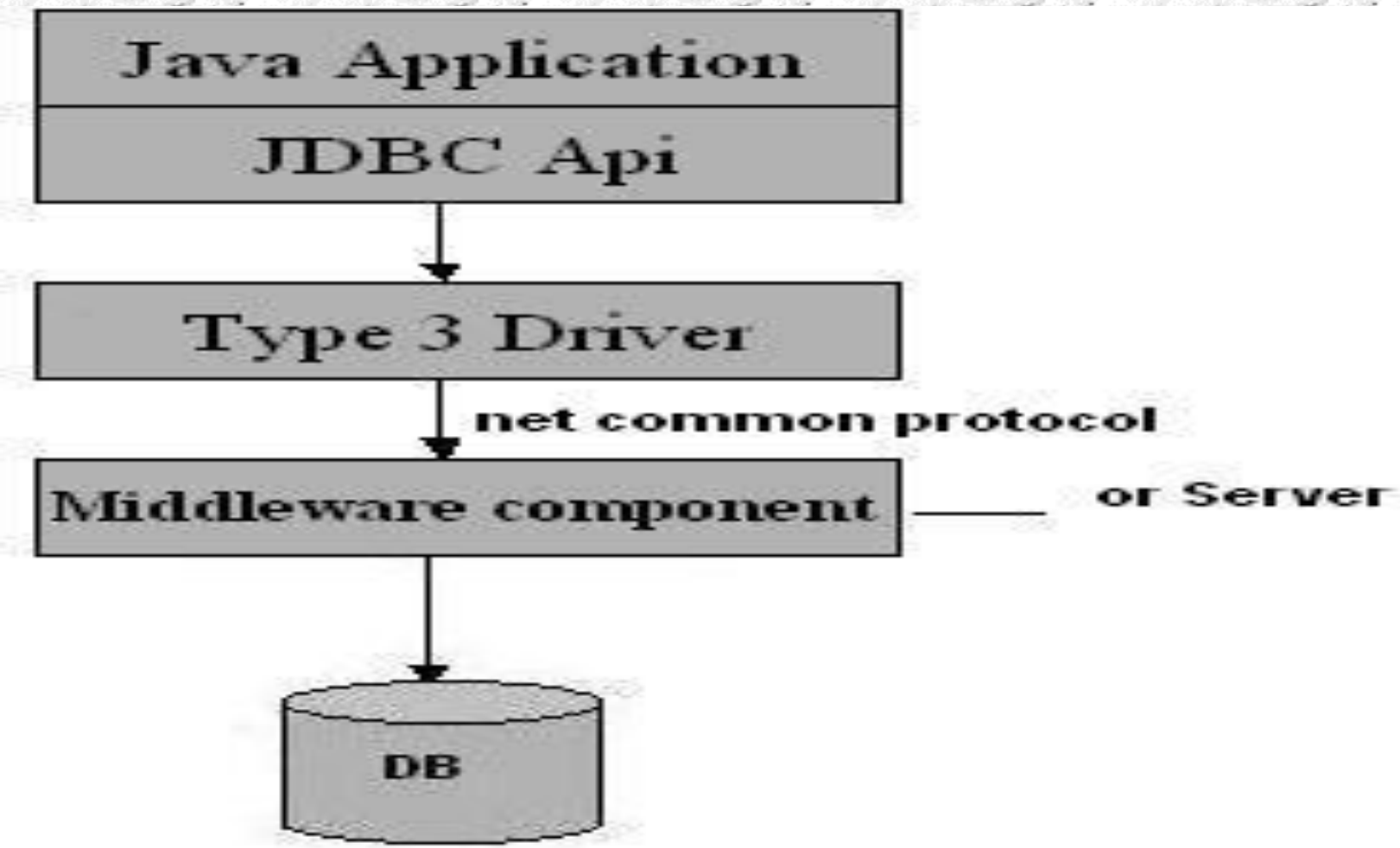
The JDBC-formatted statements are translated into the format required by the DBMS.

Examples of Type -3 Driver

IDS Driver

Weblogic RMI Server

Type 3 JDBC Driver/All Java/Net-protocol driver



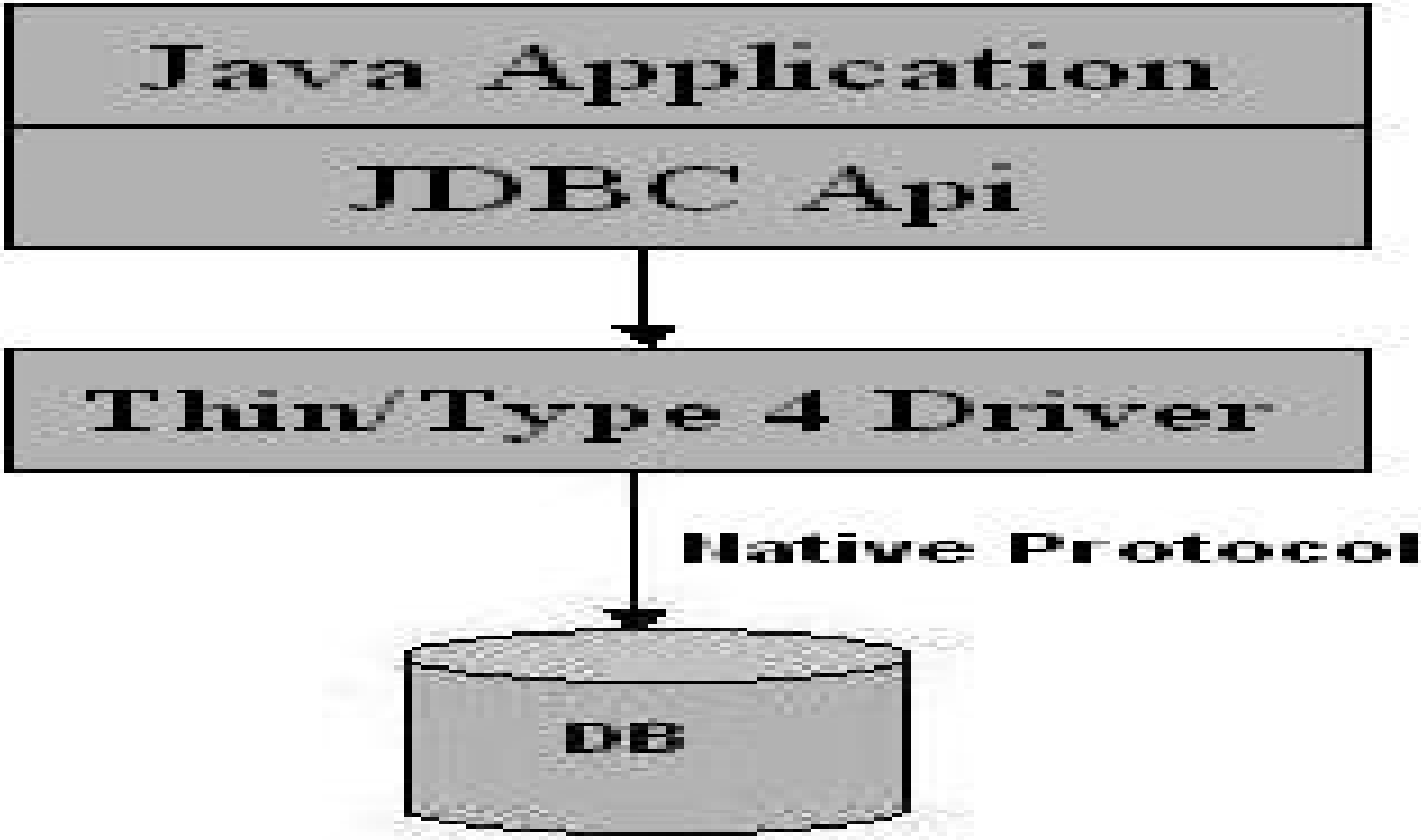
Type 4 JDBC Driver:

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

Note:

- This is the fastest way to communicate SQL queries to the DBMS.

Type 4 JDBC Driver/Native-protocol/all-Java driver



Example for Type 4 driver

- Thin Driver for oracle from Oracle Corporation
- Weblogic, MSsql server4 for MS Sql server from BEA systems

JDBC Packages:

The JDBC API is contained in two packages

1. **java.sql** - Contains core java data objects of the JDBC API.

   These include Java data objects that provide the basics for connecting to the DBMS & interacting with data stored in the DBMS. It is the part of J2SE.

2. **javax.sql** – is in the J2EE. Javax.sql package are java data objects that interact with Java Naming and Directory interface (JNDI) & java data objects that manage connection pooling among other advanced JDBC features

The *java.sql* Package

- The java.sql package is also known as the JDBC API.
- This package includes the interfaces and method to perform JDBC core operations.
- The classes in java.sql package can be categorized into:
- Connection Management: cm
- Database Access : da
- Data Types : dt
- Database Metadata : dm
- Exceptions and Warnings : ew

- Connection Management Class/Interfaces

Class/Interface
Java.sql.Connection
Java.sql.Driver
Java.sql.DriverManager

- Database Access class/Interfaces

Class/Interface
Java.sql.Callable Statement
Java.sql.PreparedStatement
Java.sql.ResultSet
Java.sql.statement

Data Types Class/Interfaces

Class/Interface
Java.sql.Blob
Java.sql.Clob
Java.sql.Types

Database Metadata Class/Intefaces

Class/Interface
Java.sql.DatabaseMetaData
Java.sql.ParameterMetaData
Java.sql.ResultSetMetaData

Exceptions and Warnings Class/Interfaces

Class/Interface
Java.sql.SQLException
Java.sql.SQLWarning

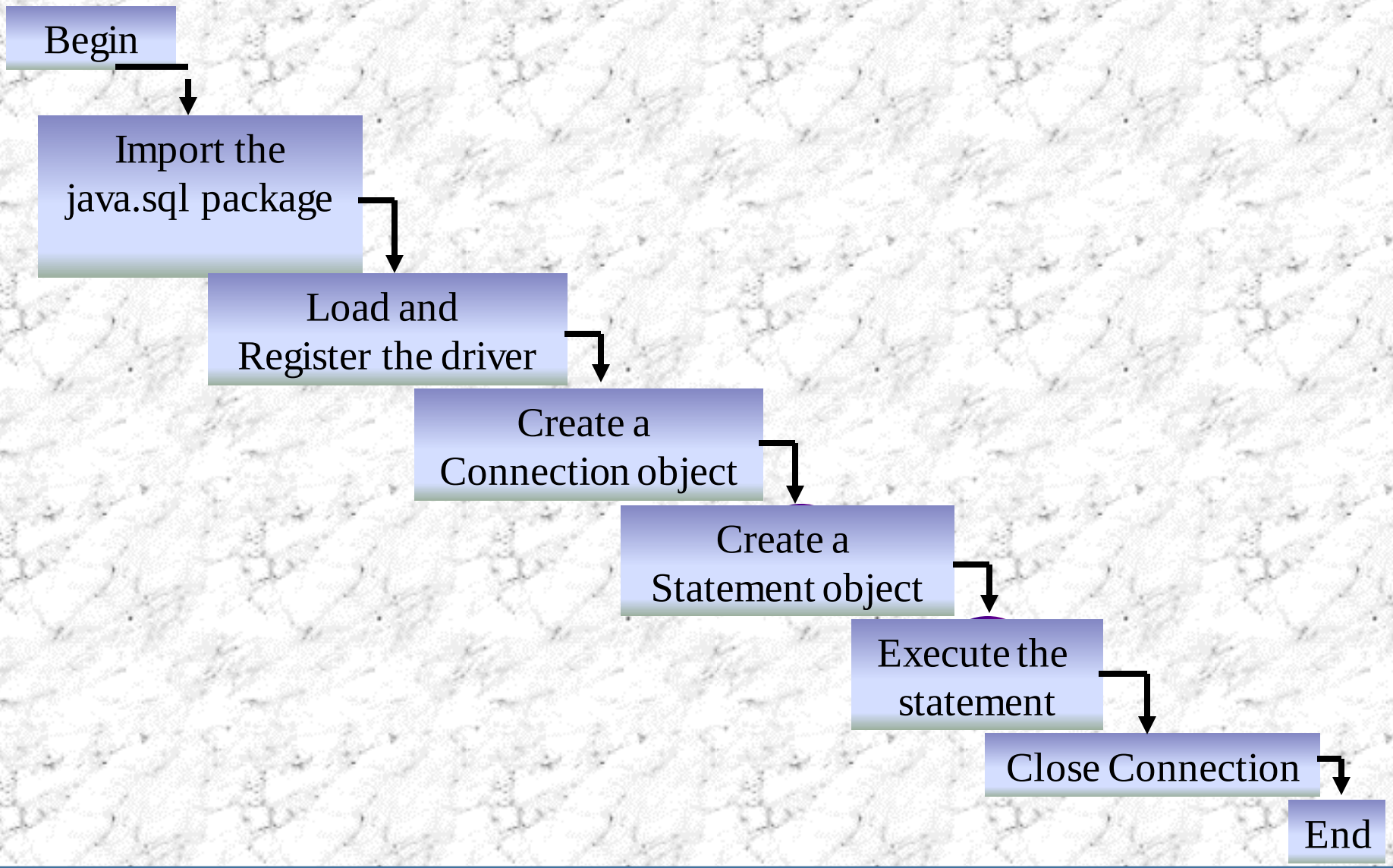
The *javax.sql* Package

- This API provides classes and interfaces for accessing server –side data sources & processing java programs.
- The javax.sql provides the following features:
  - ✓DataSource
  - ✓Connection & Statement Pooling
  - ✓Distributed transaction
  - ✓Rowsets

Overview of the JDBC process

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

Basic JDBC Steps



Popular JDBC driver names and database URL.

RDBMS	JDBC Driver name	Database URL format
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://hostname:3306 /databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname: port Number: databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2: hostname: portnumber/ databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds: hostname: portnumber/ databaseName

Database Connection:

A J2EE component does not directly connect to a DBMS

The J2EE component connects with the JDBC driver that is associated with the DBMS.

Before this connection is made, the JDBC driver must be loaded and Registered with the DriverManager.

- The purpose of loading and registering the JDBC driver is to bring the JDBC driver into the Java Virtual Machine (JVM).

- The JDBC driver is automatically registered with the DriverManager once the JDBC driver is loaded and is therefore available to the JVM and can be used by J2EE components.

- Class.forName() method is used to load the JDBC driver.
- The Class.forName() method throws a ClassNotFoundException if an error occurs when loading the JDBC driver.
- Errors are trapped using the catch{} block whenever the JDBC driver is being loaded

Cont..

- **Import the packages:**
  - *import java.sql.*;*
- **Load & Register the JDBC driver**
  - Class.forName(String URL); Listing 6-4
    eg: //mysql driver: "com.mysql.jdbc.Driver";
- **Establishing a Connection using DriverManager**
  - getConnection(String url, String username, String password)
  - Syntax of url: jdbc:<sub prootcol> : <info>
    "jdbc:mysql://localhost:3306/STUDENTS";

The Connection:

- After the jdbc driver is successfully loaded and registered, the J2EE component must connect to the database.
- The data source that the JDBC component will connect to is defined using the URL format.

- URL consists of three parts.
  - **jdbc** - which indicates that the JDBC protocol is to be used to read the URL.
  - **<suprotocol>** - Which is the JDBC driver name.
  - **<subname>** - Which is the name of the database

- The connection to the database is established by using one of the three getConnection() methods of the DriverManager object.

- The getConnection() method requests access to the database from the DBMS.

- It is up to the DBMS to grant or reject access.

- A connection object is returned by the getConnection() method if access is granted, otherwise getConnection() method throws a SQLException.

Ex: Connection con = DriverManager.getConnection(Database url)

- Sometimes the DBMS grants access to a database to anyone. In this case, the J2EE component uses the getConnection(String url) method.
- One parameter is passed to the method because the DBMS only needs the database identified.
- Other databases limit access to authorized users and require the J2EE to supply a user ID and password with the request to access the database.
- In this case, the J2EE component uses the getConnection(String url, String user, String password) method.
- For ex: slide42

- Sometimes DBMS requires information besides a user Id and password before the DBMS grants access to the database.

- This additional information is referred to as **properties** and must be associated with a Properties object , which is passed to the DBMS as a getConnection() parameter.

- Typically, properties used to access a database are stored in a text file, the contents of which are defined by the DBMS manufacturer.

- The J2EE component uses a FileInputStream object to open the file and then uses the Properties object load() method to copy the properties into a Properties object.

- For example:

```
Connection Db;
Properties props = new Properties();
try {
FileInputStream propFileStream = new
   fileInputStream("DBProps.txt");
props.load(propFileStream);
} catch(IOException e){……}
```

```
try
{
Class.forName(com.mysql.jdbc.Driver);
Db=DriverManager.getConnection(url,props);
}
catch(ClassNotFoundException e) {……}
```

TimeOut:

Competition to use the same database is a common occurrence in the J2EE environment and can lead to performance degradation of a J2EE application.

If the J2EE application that needs database access requests service from an appropriate J2EE component. In turn, the J2EE component attempts to connect to the database.

Due to various reasons, if j2ee component fails to connect to database, then the J2EE component can set a timeout period after which the DriverManager will cease to attempt to connect to the database.

The public static void DriverManager.setLoginTimeout(int seconds) method is used to retrieve from the DriverManager the maximum time the DriverManager is set to wait until it times out. And it returns an int that represents seconds

Cont..

- Creating a JDBC Statement Object
  - *Statement stmt=connection.createStatement();*
- Execute SQL Statements

  //using executeQuery()
  - *String query = "select col1, col2…..coln from table_name";*
  - *ResultSet results=stmt.executeQuery(query);*

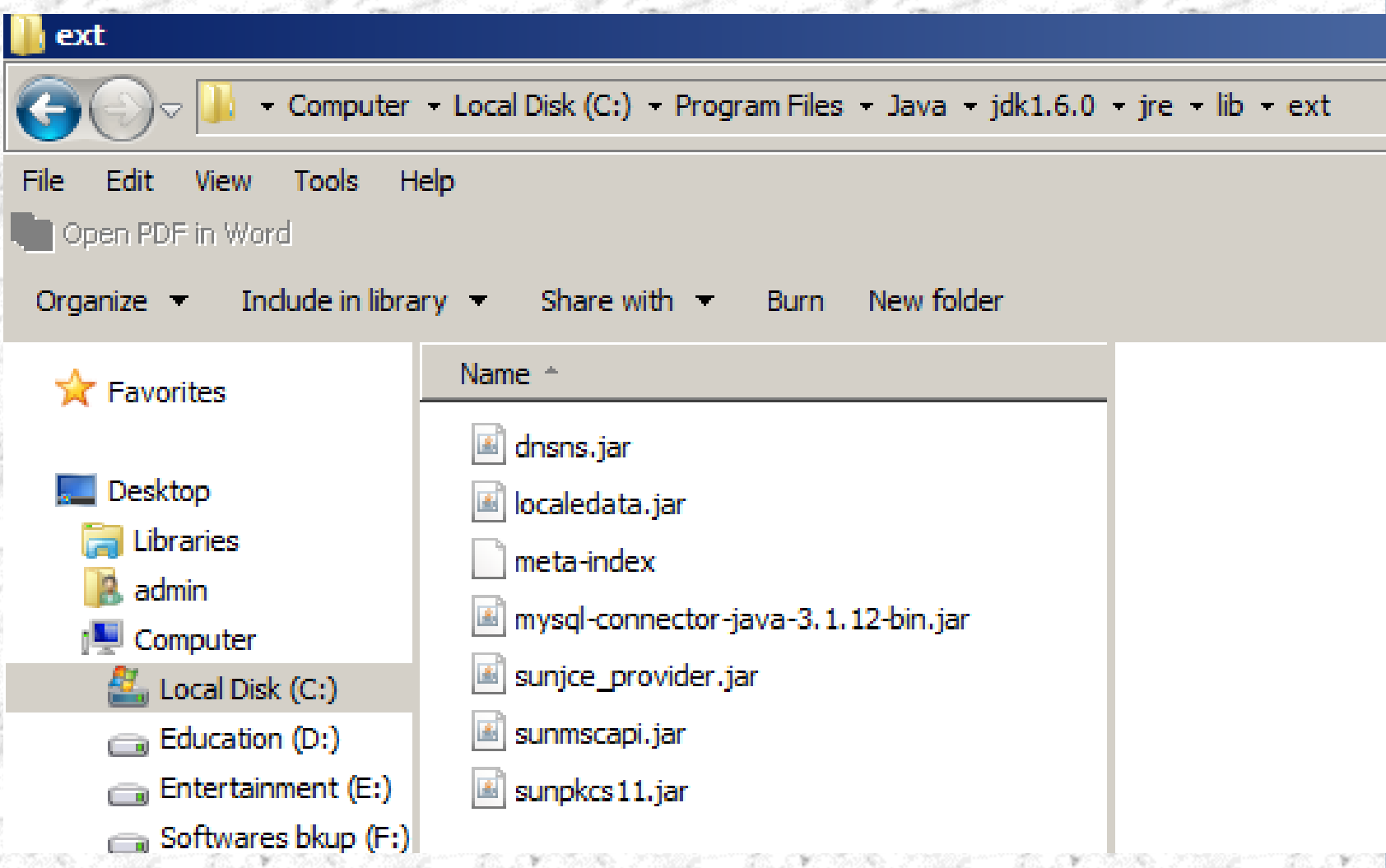
  // usng executeUpdate()
  - *String query="insert into table_name values (value1, value2,……..valuen);*
  - *Int count=stmt.executeUpdate(sql);*
- *Closing the Connection*
  - *Connection.close();*

Accessing MySQL on NetBeans using JDBC

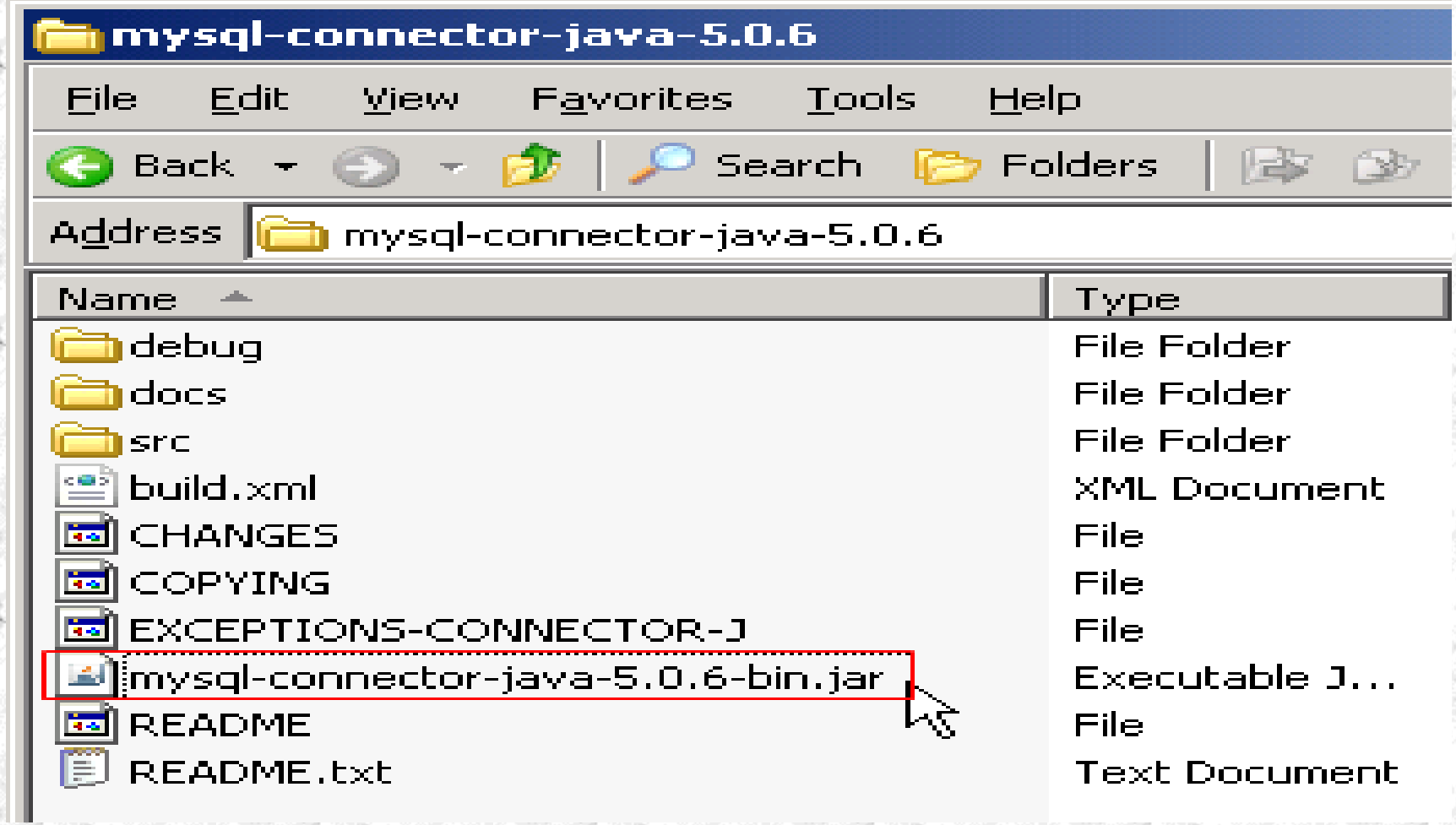
- **Requirements:**
  - MySQL Connector/J, licensed under the GPL or a commercial license from MySQL AB.
  - NetBeans with JRE (Java Runtime Environment).
- Step 1 Installation:
  - Install NetBeans IDE 7.0.1 / 6.9.1 /…..
  - Download MySQL Connector/J, name *'mysql-connector-java-5.0.6.zip'*
  - Extract the zip file to a folder, you'll see file *'mysql-connector-java-5.0.6-bin.jar'*
  - Example to "*C:\Program Files\Java\jdk1.6.0_02\lib*" directory.

If you work with notepad/edit plus editors

Copy msql-connector.jar file into the following path:



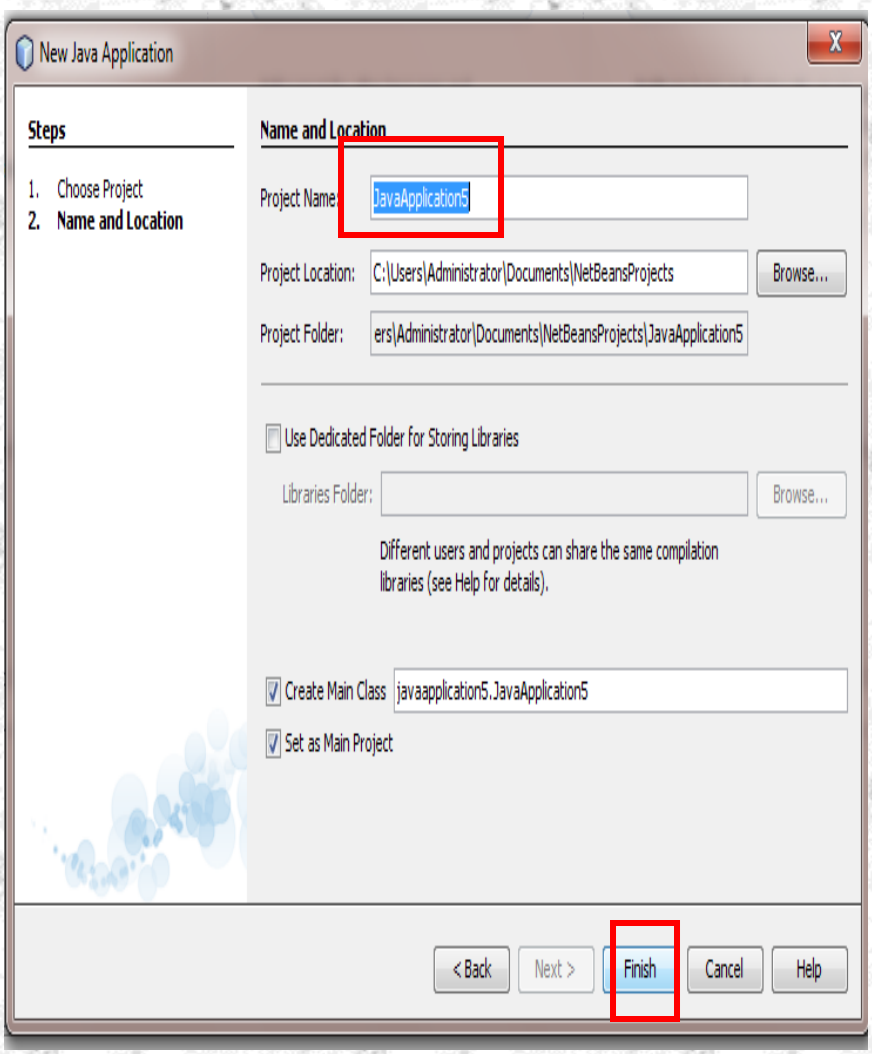
Cont...


mysql-connector-java-5.0.6
File Edit View Favorites Tools Help
Back Search Folders
Address mysql-connector-java-5.0.6
Name
Type
debug
File Folder
docs
File Folder
src
File Folder
build.xml
XML Document
CHANGES
File
COPYING
File
EXCEPTIONS-CONNECTOR-J
File
mysql-connector-java-5.0.6-bin.jar
Executable J...
README
File
README.txt
Text Document

Cont...

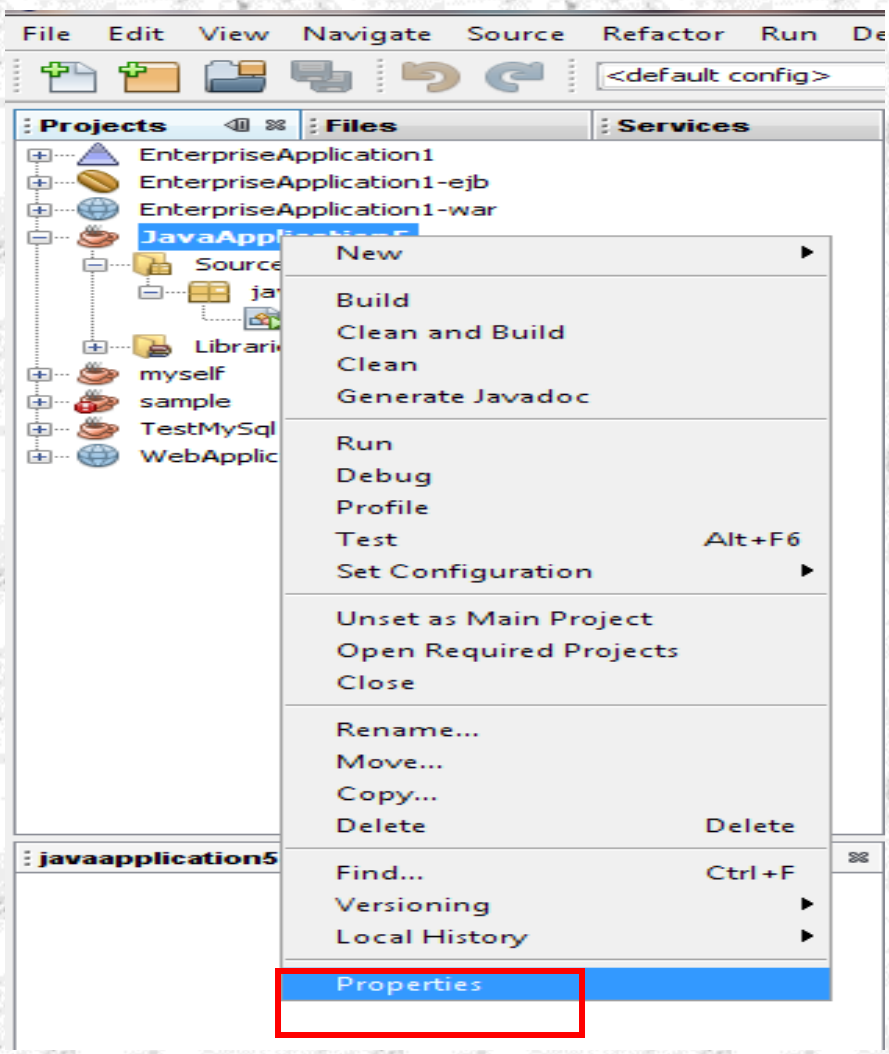
- Step 2: Add JDBC Driver to the project on NetBeans (Add a library).

1. Create New Projetct

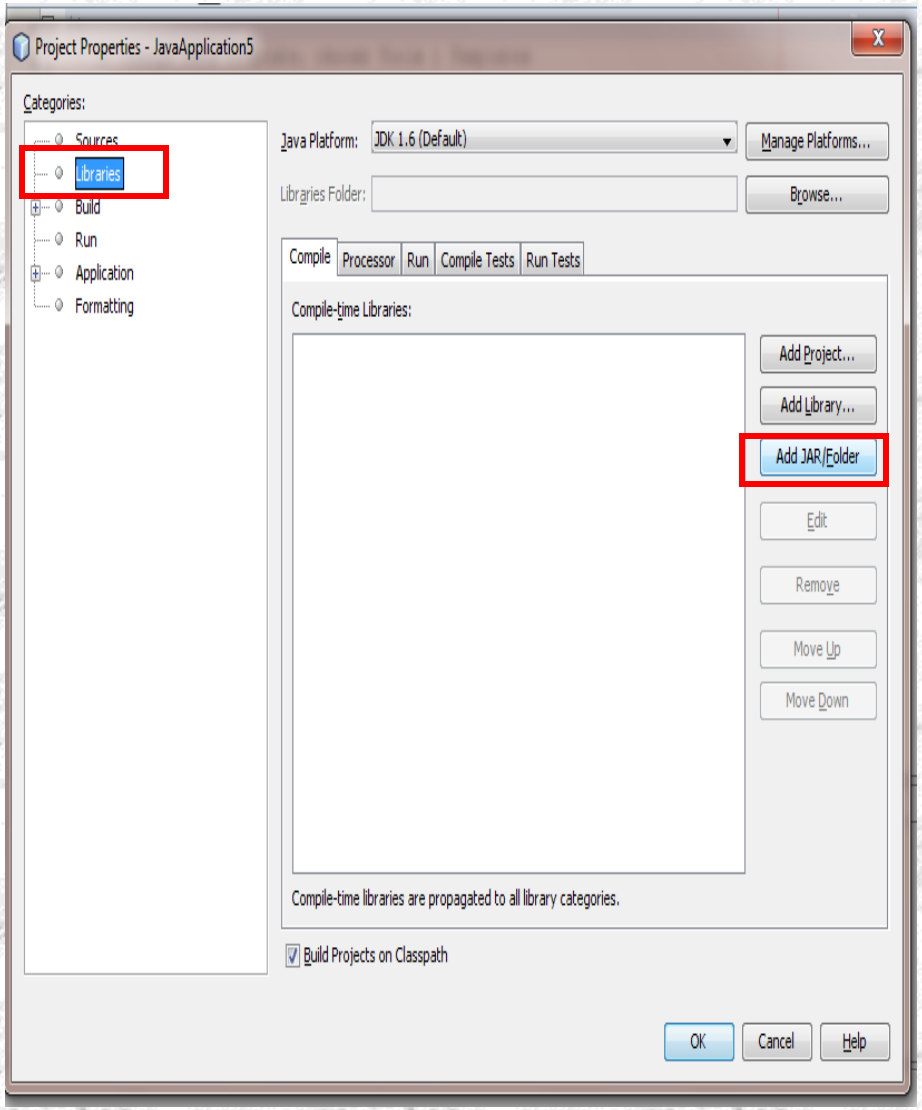


Cont...

2. In Projects window, right click the project name and select Properties.

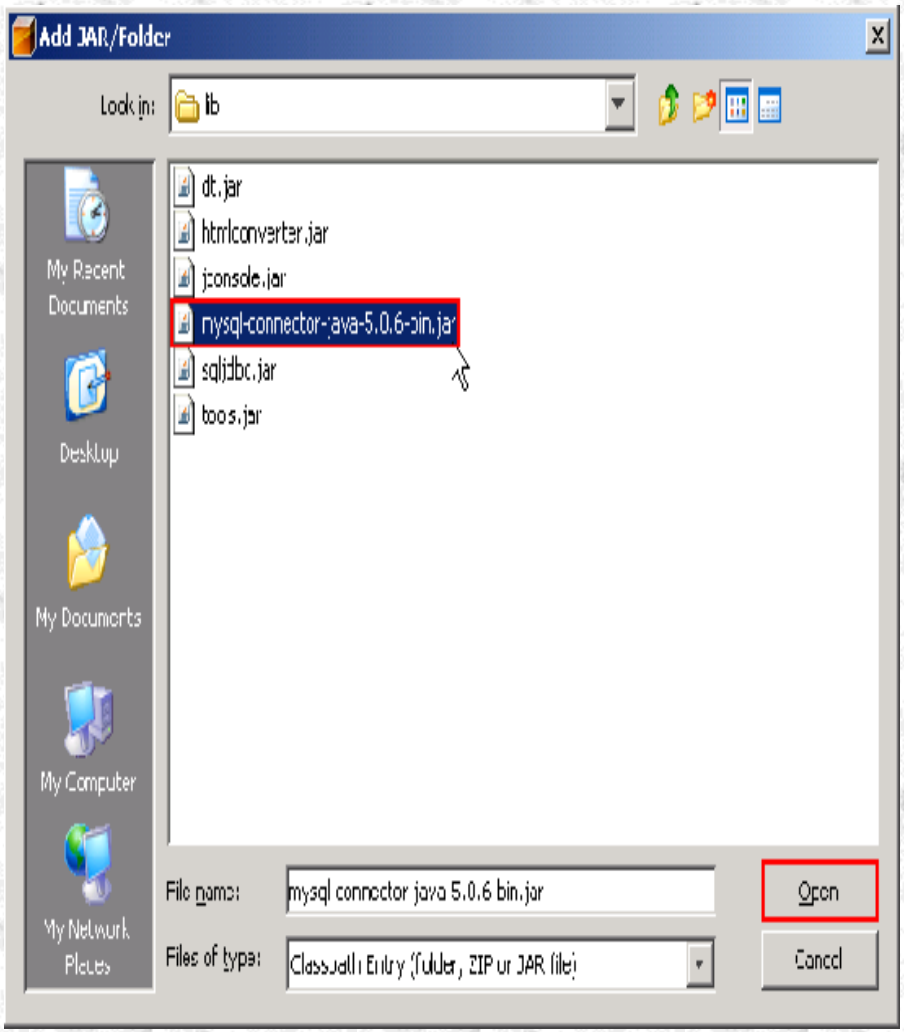


3. Project Properties window appears. The Categories on left side, select Libraries. And on right side in Compile tab, click Add JAR/Folder.



Cont...

4. New Window appears, browse to the file 'mysql-connector-java-5.0.6-bin.jar' and click Open.



Creating a simple JDBC Application

- Obtaining the connection
- Get the utility objects, such as Statement, PreparedStatement and CallableStatement to execute SQL Statements
- Execute the required SQL Statements
- Close the connection

```
package studentdb;
import java.sql.*;

public class studentdb{
  private Connection con;
  private Statement st;
 // private ResultSet rs;
  public studentdb()
  {
    String url = "jdbc:mysql://localhost:3306/studentdb";
    String userID = "root";
    String password = "root";
    try {
      Class.forName( "com.mysql.jdbc.Driver");
      con = DriverManager.getConnection(url,userID,password);
    }
    catch (ClassNotFoundException error) {
      System.err.println("Unable to load the MySql Driver" + error);
      System.exit(1);
    }
```

```
catch (SQLException error) {
     System.err.println("Cannot connect to the database." + error);
     System.exit(2);
    }
   try
   {      //Enter Example Code Here (like create, insert, delete, update,
view, index, drop, join etc.....
     String query1;
       query1 = "CREATE TABLE studenttable ("+
              "regno char(11),"+
              "name char(50)," +
              "sem char(1)," +
              "percent char(3))";
    st  = con.createStatement();
   st.execute(query1);
    con.close();
  }
```

```
catch ( SQLException error )
    {
      System.err.println("SQL  error." + error);
    }
    }
public static void main ( String args [] )
    {
      final studentdb  sql1 = new studentdb  ();
      System.exit  ( 0 ) ;
    }
}
```

Connection Pool:

- Connecting to a database is performed on a per-client basis.

- Each client must open its own connection to a database and the connection cannot be shared with unrelated clients.

Leaving a connection open might prevent another client from accessing the database should the DBMS have available a limited number of connections.

Connecting and re-connecting is simply time-consuming and cause performance degradation.

A connection pool is a collection of database connections that are opened once and loaded into memory so these connections can be reused without having to reconnect to the DBMS.

Clients use the DataSource interface to interact with the Connection pool. The connection pool itself is implemented By the application server, which hides from the client details On how the connection pool is maintained.

There are two types of connections made to the database

1. Physical connection.
2. Logical connection

The physical connection, which is made by the application Server using PooledConnection objects.

PooledConnection objects are cached and reused.

A logical connection is made by a client calling the DataSource.getConnection() method, which connects to a PooledConection object that already has a physical Connection to the database. For ex:

```
Context ctext = new InitialContext();
DataSource pool = (DataSource) ctext.lookup(“
java:comp/env/jdbc/pool”);
Connection db = pool.getConnection();
//place code to interact with the database here
db.close();
```

Statement Objects:

Once a connection to the database is opened, the J2EE Component creates and sends a query to access data Contained in the database. The query is written using SQL.

One of three types of statement objects is used to execute The query. These objects are:

1. Statement: Which executes a query immediately
2. PreparedStatement: Which is used to execute a compiled query
3. CallableStatement: Which is used to execute store procedures.

Statement Object:

It is used whenever a J2EE component needs to Immediately execute a query without first having the Query compiled.

It contains the executeQuery() method, which is passed The query as an argument. The query then transmitted To the DBMS for processing.

The executeQuery() method returns one ResultSet Object that contains rows, columns, and metadata that Represent data requested by query.

The ResultSet object also contains methods that are used To manipulate Setdata in the Result.

The execute() method of the statement object is used when There may be multiple results returned.

The executeUpdate() method is used to execute queries That contain UPDATE and DELETE SQL statements, Which changes values in a row and removes a row Respectively.

The executeUpdate() method returns an integer indicating The number of rows that were updated by the query.

The executeUpdate() is used to INSERT, UPDATE, DELETE, and DDL statements.

The createStatement() method of the Connection object is Called to return a Statement object.

The following example illustrate how to use the executeUpdate() method of the Statement object.

PreparedStatement Object:

A SQL query must be compiled before the DBMS Processes the query. Compiling occurs after one of the Statemtent Object's execution methods is called.

Compiling a query is an overhead that is acceptable if The query is called once.

However, the compiling process can become an expensive Overhead if the query is executed several times by the Same instance of the J2EE component during the same Session.

A query can be precompiled and executed by using the PreparedStatement object.

However, a question mark is used as a placeholder for a Value that is inserted into the query after the query is Compiled.

It is this value that changes each time the query is executed.

The preparedStatement() method of the Connection object is Called to return the PreparedStatement object.

The preparedStatement() method is passed the query, which Is then precompiled.

The setXXX() method of the PreparedStatement object is Used to replace the question mark with the value passed To the setXXX() method.

XXX represents the data types of the value that is being Passed to the setXXX() method.

The setXXX() requires two parameters.
The first parameter is an integer that identifies the position Of the question mark placeholder and the second Parameter is the value that replaces the question mark Placeholder.

Listing 6-11.java

The advantage of using the PreparedStatement object is That the query is precompiled once and the setxxx() Method called as needed to change the specified values Of the query without having to recompile the query.

The PreparedStatement object also has an execute() & executeUpdate() method.

The precompiling is performed by the DBMS and is Referred to as "late binding" . When the DBMS receives The request, the DBMS attempts to match the query to A previously compiled query. If found, then parameters Passed to the query using the setxxx() methods are Bound and the query is executed.

If not found, then the query is compiled and retrained by The DBMS for later use.

The JDBC driver passes two parameters to the DBMS.
1. One parameter is the query and

2. the other is an array of late binding variables.

Both binding and compiling is performed by the DBMS.

The late binding is not associated with the specific object Or code block where the preparedStatement() is Declared.

Callable Statement:

The CallableStatement object is used to call a stored Procedure from within a J2EE object.

The stored procedure is executed by invoking the name of The stored procedure.

The callableStatement object uses three types of Parameters when calling a stored procedure.

1. IN
2. OUT
3. INOUT

The IN parameter contains any data that needs to be Passed to the stored procedure and whose value is Assigned using the setxxx() method.

The OUT parameter contains the value returned by the Stored procedures, if any.

The OUT parameter must be registered using the registerOutParameter() method and then is later retrieved By the J2EE component using the getxxx() method.

The INOUT parameter is a single parameter that is used To both pass information to the stored procedure. And Retrieve information from a stored procedure.

Listing 6-12.java

ResultSet:

A query is used to update, delete, and retrieve information Stored in a database.

The executeQuery() method is used to send the query to The DBMS for processing and returns a ResultSet object That contains data that was requested by the query.

The ResultSet object contains methods that are used to Copy data from the ResultSet into a java collection object Or variable for further processing.

Data in a ResultSet object is logically organized into a Virtual table consisting of rows and columns.

In addition to data, the ResultSet object also contains Metadata such as column names, column size, and Column data type.

The ResultSet uses a virtual cursor to point to a row of The virtual table.

A J2EE component must move the virtual cursor to each Row and then use other methods of the ResultSet object To interact with the data stored in columns of that row.

A virtual cursor is positioned above the first row of data When the ResultSet is returned by the executeQuery() Method.

The virtual cursor is positioned above the first row of data when the ResultSet is returned by the executeQuery() method.

The virtual cursor is moved to the first row using the next() method.

The next() method returns a boolean true if the row contains data; otherwise, a boolean false is returned indicating that no more rows exist in the ResultSet.

Listing 6-13.java

Scrollable ResultSet:

Until the release of the JDBC 2.1 API, the virtual cursor could only be moved down the ResultSet object.

But, today, the virtual cursor can be moved backwards or even positioned at a specific row.

There are six methods of the ResultSet object that are used to position the virtual cursor in addition to the next() method. These are

1. first() 2. last() 3. previous() 4. absolute()
5. relative(), and 6. getRow().

The first() method moves the virtual cursor to the first row in the ResultSet.

The last() method moves the virtual cursor to the last row in the ResultSet.

The previous() method moves the virtual cursor to the previous row from the current position.

The absolute() method positions the virtual cursor at the row number specified by the integer passed as a parameter to the absolute() method.

The relative() method moves the virtual cursor the specified number of rows contained in the parameter.

The parameter is a positive or negative integer where the sign represents the direction the virtual cursor is moved.

For ex: -4 moves the virtual cursor back four rows from the current row.

And the getRow() method returns an integer that represents the number of the current row in the ResultSet.

The statement object that is created using the createStatement() of the Connection object must be set up to handle a scrollable ResultSet by passing the createStatement() method one of three constants.

St = con.createStatement(TYPE_FORWARD_ONLY)

1. TYPE_FORWARD_ONLY: constant restricts the virtual cursor to downward movement, which is the default setting.

2. TYPE_SCROLL_INSENSITIVE: constant makes the ResultSet insensitive to changes made by another J2EE component to data in the table whose rows are reflected in the ResultSet

3. TYPE_SCROLL_SENSITIVE: constant makes the ResultSet sensitive to those changes.

The above two constants permit the virtual cursor to move in both directions.

Note: Not all JDBC Drivers are Scrollable

Listing 6-14.java

UPDATABLE RESULTSET:

Rows contained in the ResultSet can be updatable similar to how rows in a table can be updated.

This is made possible by passing the createStatement() method of the Connection object the CONCUR_UPDATABLE.

```
Statement st =
con.createStatement(CONCUR_UPDATABLE)
```

Alternatively, the CONCUR_READ_ONLY constant can be passed to the createStatement() method to prevent the ResultSet from being updated.

1. Updating values in a row
2. Deleting a row
3. Inserting a new row.

All these changes are accomplished by using methods of the Statement object.

Update ResultSet:

String qry="select * from customers";
Statement st = con.createStatement();
ResultSet rs = st.executeQuery(qry);
C_fname="skt"

Once the executeQuery() method of the Statement object returns a ResultSet, the updatexxx() method is used to change the value of a column in the current row of the ResultSet.

The updatexxx() method requires two parameters.

1. Either the number or name of the column of the ResultSet that is being updated.
2. Second parameter is the value that will replace the value in the column of the ResultSet.

Example: updateString("c_fname","skt")

A value in a column of the ResultSet can be replaced with a NULL value by using the updateNull() method.

Ex: updateNull(2)

updateNull() method requires one parameter, which is the number of column in the current row of the ResultSet.

Note: The updateNull() doesn't accept the name of the column as a parameter.

The updateRow() method is called after all the updatexxx() m

C_fname	C_lastname	Balance	Paid status	Contact_no
shiva	T	2000	No	8888888888
Skt	R	0	Yes	4949494949

T ıs of the current row of the ResultSet based on the values of the updatexxx() methods.


Listing 6-17.java

Delete Row in the ResultSet:

The deleteRow() method is used to remove a row from a ResultSet.

The deleteRow() method is passed an integer that contains the number of the row to be deleted.

Results.deleteRow(0); - Deletes the current row

Insert Row in the ResultSet:

Inserting a row into the ResultSet is accomplished using basically the same technique as is used to update the ResultSet.

The insertRow() method is called after the updatexxx() methods, which causes a new row to be inserted into the ResultSet having values that reflect the parameters in the updatexxx() methods. This also updates the underlying database

Listing 6-18.java

Transaction Processing:

A database transaction isn't completed until the J2EE component calls the **commit()** method of the Connection object.

Note: **commit() method was automatically called in these examples because the DBMS has an AutoCommit feature that is by default set to true.**

Listing 6-19.java

J2EE component can control the number of tasks that are rolled back by using savepoints.

A savepoint, introduced in JDBC 3.0, is a virtual marker that defines the task at which the rollback stops.

setSavepoint(savepoint_name) is used to set the savepoint.

releaseSavepoint() method is called to remove the savepoint from the transaction. The name of the savepoint that is to be removed is passed to the releaseSavepoint() method.

Batch Transaction:

Another way to combine SQL statements into a transaction is to batch together these statements into a single transaction and then execute the entire transaction.

By using the addBatch() method of the Statement object.

addBatch() method receives a SQL statement as a parameter and places the SQL statement in the batch.

Once all the SQL statements that comprise the transaction are included in the Batch, the executeBatch() method returns an int array that contains the number of SQL statements that were executed successfully.

The batch can be cleared of SQL statements by using the clearBatch() method.

The transaction must be committed using the commit() method.

Note: Make sure that setAutoCommit() is set to false before executing the batch.


Listing 6-21.java

ResultSet Holdability:

Whenever the commit() method is called, all ResultSet objects that were created for the transaction are closed.

Sometimes, a J2EE component needs to keep the ResultSet open even after the commit() method is called.

We can control whether or not ResultSet objects are closed following the call to the commit() method by passing one of two constants to the createStatement() method.

These constants are
ex: Statement st =
con.createStatement(CLOSE_CURSORS_AT_COMMIT)

1. HOLD_CURSORS_OVER_COMMIT and
2. CLOSE_CURSORS_AT_COMMIT

The first one constant keeps ResultSet objects open following a call to the commit() method

Second one, closes ResultSet objects when the commit() method is called.

RowSets:

The JDBC RowSets object is used to encapsulate a ResultSet for use with Enterprise Java Beans.

A RowSet object contains rows of data from a table(s) that can be used in a disconnected operation.

Ie., an EJB can interact with a RowSet object without having to be connected to a DBMS, which is ideal for J2EE components that have PDA clients.

Auto-Generated Keys:

It is common for a DBMS to automatically generate unique keys for a table as rows are inserted into the table.

The getGeneratedKeys() method of the Statement object is called to return keys generated by the DBMS.

This method returns a ResultSet object. You can use the ResultSet.getMetaData() method to retrieve metadata relating to the automatically generated key, such as the type and properties of the automatically generated key.

Metadata:

Metadata is data about data. A J2EE component can access metadata by using the DatabaseMetaData interface.

This interface is used to retrieve information about databases, tables, columns, and indexes among other information about the DBMS.

A J2EE component retrieves metadata about the database by calling the getMetaData() method of the Connection object.

The getMetaData() method returns a DatabaseMetaData object that contains information about the database and its components.

Once the DatabaseMetaData object is obtained, an assortment of methods contained in the DatabaseMetaData object are called to retrieve specific metadata.

Some of the more commonly used DatabaseMetaData object methods.

- getDatabaseProductName() – Returns the product name of the database.
- getUserName() – Returns the username
- getURL () – Returns the URL of the database
- getSchemas() – Returns all the schema names available in this database.
- getPrimaryKeys() – Returns primary keys.
- getProcedures() – Returns stored procedure names.
- getTables() – Returns names of tables in the database.

ResultSet MetaData:

There are two types of metadata that can be retrieved from the DBMS.

These are

1. metadata that describes the database
2. Metadata that describes the ResultSet.

Metadata that describes the ResultSet is retrieved by calling the getMetaData() method of the ResultSet object.

This returns a ResultSetMetaData object,

```
ResultSetMetaData rm=Result.getMetaData()
```

Once the ResultSet metadata is retrieved, the J2EE can call methods of the ResultSetMetaData (rsmd) object to retrieve specific kinds of metadata.

Rsmd.getColumnCount() = 4

1. getColumnCount() – Returns the number of columns contained in the ResultSet.

Rsmd.getColumnName(4)

2. getColumnName(int number) – Returns the name of the column specified by the column number.

Rsmd.getColumnType(2)

3. getColumnType(int number) – Returns the data type of the column specified by the column number.

Data Types

The setxxx() amd getxxx() methods are used to set a value of a specific data type and to retrieve a value of a specific data type.

The xxx in the names of these methods is replaced with the name of the data type.

Data Types in Java	
SQL Type	**Java Type**
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	Java.math.BigDecimal
DECIMAL	Java.math.BigDecimal
BIT	Boolean
TINYINT	Byte
SMALLINT	Short
INTEGER	Integer
BIGINT	Long
REAL	float
FLOAT	float
DOUBLE	double

Data Types in Java	
SQL Type	**Java Type**
BINARY	Byte[ ]
VARBINARY	Byte[ ]
LONGVARBINARY	Byte[ ]
BLOB	java.sql.Blob
CLOB	java.sql.Clob
ARRAY	java.sql.Array
STRUCT	java.sql.Struct
REF	java.sql.Ref
DATALINK	java.sql.Types
DATE	java.sql.date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Exceptions:

There are three kinds of exceptions that are thrown by JDBC methods.

1. SQLExceptions – commonly reflects a SQL syntax error in the query & are thrown by many of the methods contained in the java.sql.package.

The getNextException() method of the SQLExceptions object is used to return details about the SQL error or null if the last exception was retrieved.

The getErrorCode() method of the SQLException object is used to retrieve vendor-specific error codes.

2. SQLWarnings – throws warnings received by the Connection from the DBMS.

getWarnings() method of the Connection object retrieves the warning and the getNextWarning() method of the connection object retrieves subsequent warnings.

3. Data Truncation – Whenever data is lost due to transaction of the data value, a DataTruncation exception is thrown.

SUMMARY:

- JDBC SPECIFICATION
- JDBC DRIVER TYPES
- JDBC PACKAGES
- JDBC PROCESS
- DATABASE CONNECTION
- CONNECTION POOL
- STATEMENT OBJECTS
- RESULT SET
- UPDATABLE RESULTSET
- TRANSACTION PROCESSING
- METADATA
- DATA TYPES
- EXCEPTIONS

References

- Java Complete Reference text book, Herbert Schedilt
- Java Programming, Black Book
- Enterprise Java Beans 3.1