Web Based Java Programming (4350708)

Semester – 5th, Diploma in Computer Engineering

COMPUTER ENGINEERING, A. V. PAREKH TECHNICAL INSTITUTE, RAJKOT

Examination Scheme

EXAMINATION SCHEME				
THEORY MARKS		PRACTICAL MARKS		TOTAL MARKS
ESE	CA	ESE	CA	
70	30	25	25	150

Course Outcomes

Web Based Java Programming (4350708)	
CO1	Implement basic database operations using JDBC
CO2	Develop database-driven Java applications using Hibernate ORM framework.
CO3	Develop server side programs using Servlets.
CO4	Develop Java Server Pages application using JSP tags.
CO5	Develop networked applications in java using network protocols, socket programming, and related technologies.
CO6	Develop of simple web service applications using Java technologies

Java Database Connectivity (JDBC)

Unit - 1

Topics to be Covered…

- Introduction & JDBC Architecture – 2 Tier,& 3 Tier
- JDBC Components
- JDBC API
  - Statements: Statement, PreparedStatement & CallableStatement
  - ResultSet
  - Transaction Processing: Commit, Rollback, Savepoint
  - Creating Simple JDBC Application
- JDBC Drivers
  - Types
  - Advantages
  - Disadvantages

Introduction to JDBC

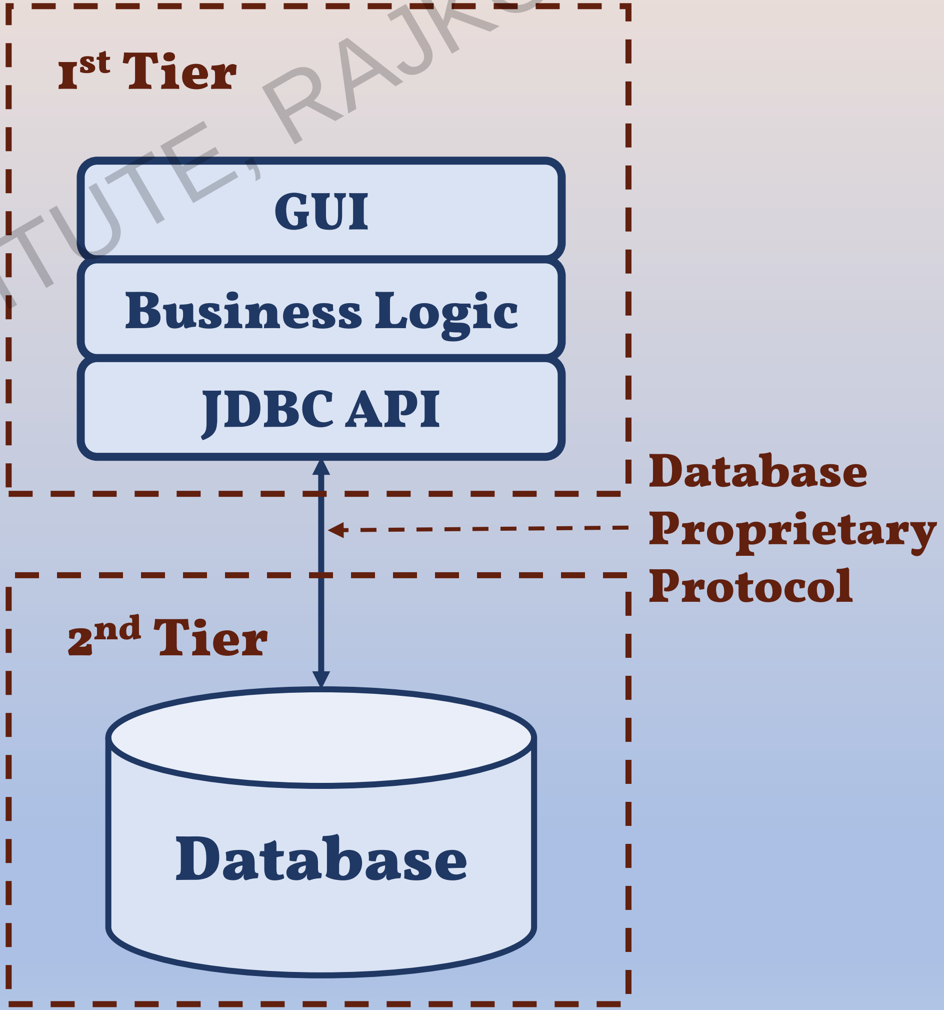
- Allows Java Applications to **communicate with Database**.
- Almost all RBDMS are supported like Microsoft SQL Server, MySQL, Oracle, Microsoft Access etc. and data connectivity is also reliable.
- Currently JDBC API 4.3 is introduced in Java SE 9, which is also backward compatible.
- Mainly 2 packages are to be studied:
  - java.sql: Contains set of classes and interfaces used to connect with database
  - javax.sql: Extended package containing set of classes and interfaces to **support Server-side database connectivity and data processing**.

Introduction to JDBC

- Various operations can be performed using JDBC API like,
  - **Access the database** from system securely.
  - **Performing various operations** on the database from the Java Program which includes operations like, Creating, Updating and Altering Table structures.
  - **Inserting, Updating, Deleting and Selecting rows** of the Data Table.
  - Then **representing the data on the User side** without users having to bother about its implementation structure.
  - Even **Transaction processing** is supported in JDBC like Commit, Rollback, Savepoints.
- Now, the data is contained in some other application (RDBMS). So these applications are no longer simplistic 1-Tier Applications, rather they are **2 or 3 Tier Applications**.

JDBC Architecture: 2-Tier Architecture

- Here, the Data is stored somewhere else than the executable file which is being used (.class / .java)
- So, **.java file (1st of 2 Tiers)** contains 2 things:
  - **GUI:** How to get and represent user Data.
  - **Business Logic:** Constraints on the data communications which are required by Application designer.
- Another is the **Database (2nd of 2 Tiers)** itself:
  - RDBMS software containing all data.
  - This will communicate with the JDBC API and processes the data as instructed by JDBC API from the 1st Tier.

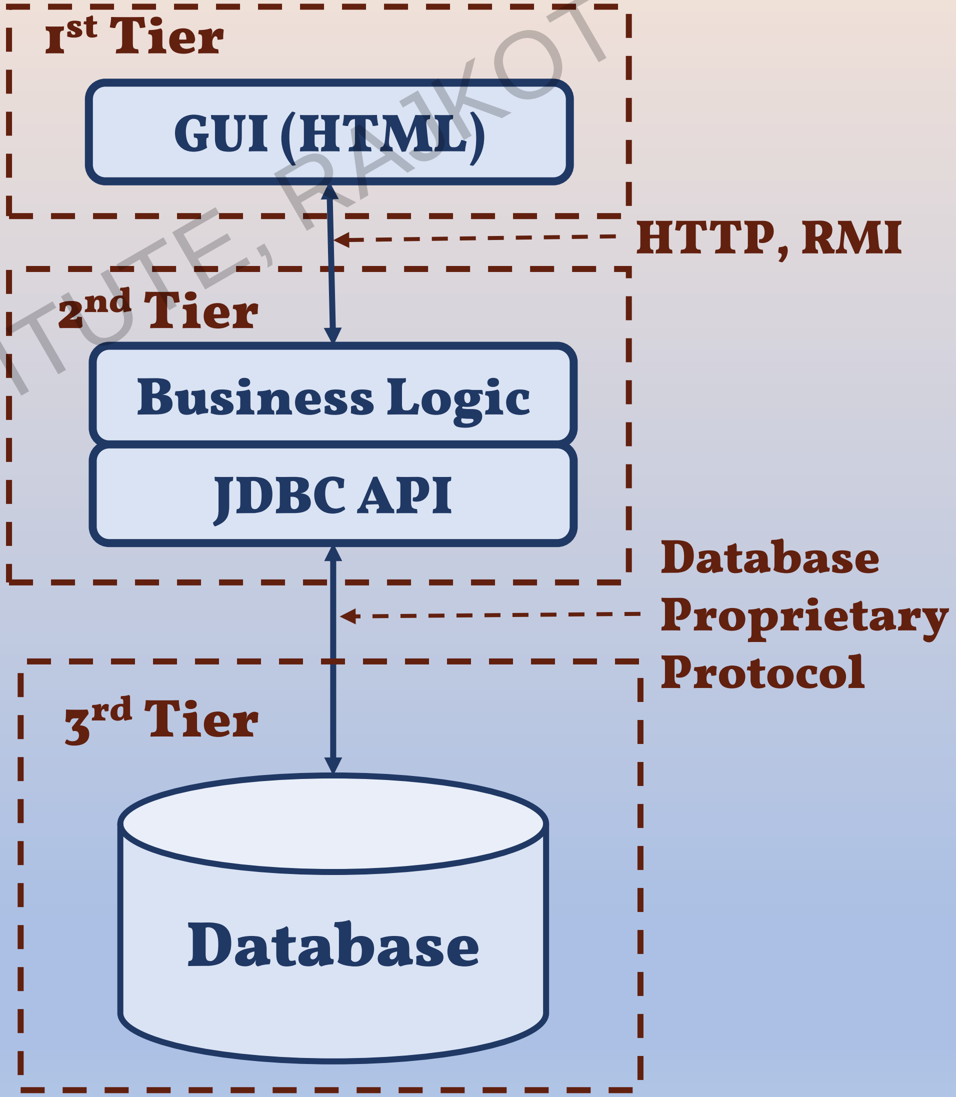


JDBC Architecture: 2-Tier Architecture

- Advantages:
  - Simplicity and cost-effectiveness.
  - Direct database control for fine-tuned queries.
- Disadvantages:
  - Limited scalability and difficulty with concurrent access.
  - Security concerns with embedded credentials.
  - Maintenance challenges as the application grows.
  - Platform dependence and lack of separation of concerns.

JDBC Architecture: 3-Tier Architecture

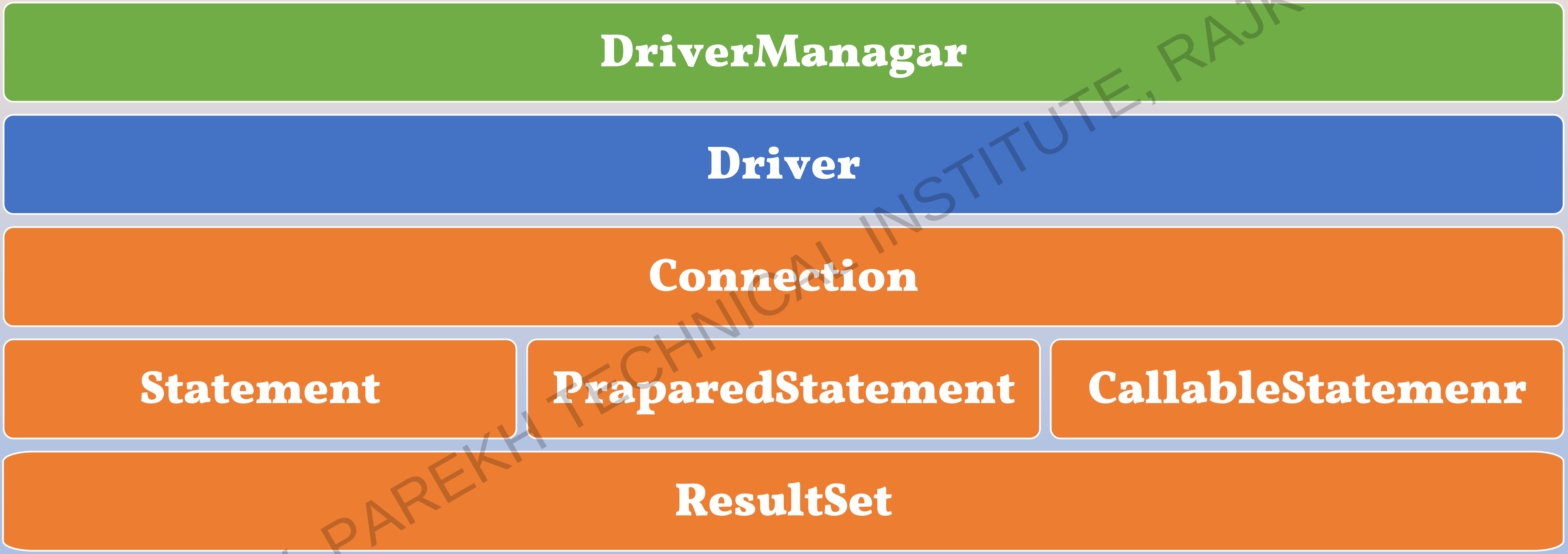
- Here, whole architecture is divided into 3-parts
- 1st **Tier** contains **GUI,** which is how to get and represent user Data, which is usually HTML or an RMI Enabled File
- 2nd **Tier** contains **Business Logic & JDBC API:** Constraints on the data communications which are required by Application designer & JDBC API classes to communicate with the database.
- **3rd of 3 Tiers is Database** itself:
  - RDBMS software containing all data.
  - This will communicate with the JDBC API and processes the data as instructed by JDBC API from the 2nd Tier.



JDBC Architecture: 3-Tier Architecture

- Advantages:
  - Improved scalability, security, and maintainability.
  - Easier platform independence and load balancing.
- Disadvantages:
  - Increased complexity and potential performance overhead.
  - Higher infrastructure and deployment complexity.
  - Learning curve for developers.
  - Overengineering for smaller applications.

JDBC API



DriverManagar
Driver
Connection
Statement
PraparedStatement
CallableStatemenr
ResultSet

JDBC API: DriverManager

- Class
- Used to keep track of the available driver.
- Creates a connection to the database using **Specified Driver**.
- Method:

```
public static Connection getConnection(String connURL, String
                                username, String password);
```

- Example:

```
Connection conn =
 DriverManager.getConnection("jdbc:mysql://localhost/StudentDB",
 "root","");
```

JDBC API: Driver

- Sometimes before creating a connection, we need to load the driver class from the specified drivers. (Usually needed in Old versions of JDK)

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

- This methods works in loading the class into current environment for instantiation whenever needed.

JDBC API: Connection

- Next Step is to use the Connection Object created in previous step and create a Statement Object, to execute SQL Statements over that connection.
- Method used to create statement object is from Connection, So it can be called using Connection type object:

```
public Statement createStatement();
```

- Example:

```
Statement stmt = conn.createStatement();
```

JDBC API: Statement

- Statements are of 3 types.

Statement

- Used to create a simple statement
- Ex.: `Statement stmt = conn.createStatement();`

PreparedStatement

- Used to create a Statement where same query is required to be executed on multiple data repeatedly.
- Ex.: `PreparedStatement pstmt = conn.prepareStatement("INSERT INTO StuResult ('Enr', 'Name', 'SPI') VALUES (?, ?, ?);");`
- `pstmt.setInt(1, 159);`
- `pstmt.setString(2, "Raj");     pstmt.setDouble(3, 8.0);`

CallableStatement

- Used to create a statement which is able to call Stored Procedures / Functions in SQL
- Ex.: `CallableStatement cstmt = conn.prepareCall("{call fetchResultProc(?);}");`
- `cstmt.setInt(1, 159);`

JDBC API: Execute Query

- After the Statement is generated, the query needs to be executed from the created statement object.
- 3 methods with 2 overloaded type for each is available to execute query.

```
public boolean execute();
public boolean execute(String query);
```

Suitable for all types of queries.
- true if query retuns a ResultSet, otherwise false.

```
public int executeUpdate();
public int executeUpdate(String query);
```

Suitable for Insert, Update types of queries.
- Retuns number of rows affected by the query.

```
public ResultSet executeQuery();
public ResultSet executeQuery(String query));
```

Suitable for Select type of query.
- Retuns the tabular data on the form of ResultSet object

JDBC API: Execute Query

- Ex.

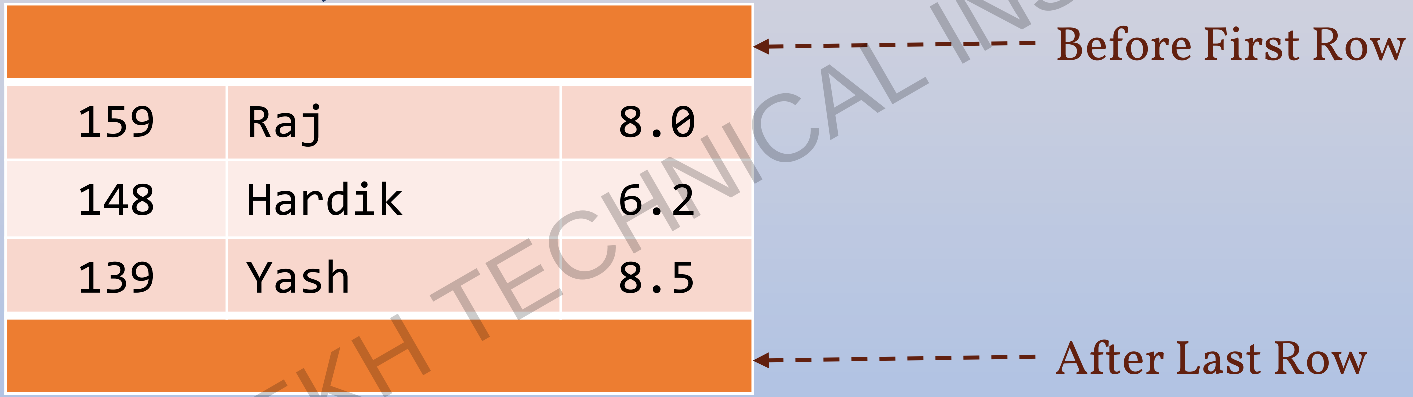
```
ResultSet table = stmt.executeQuery("SELECT * FROM StuResult");
```

- This returns a table in the form of a ResultSet object.
- Close the ResultSet and Connection, when no longer needed for security and performance.

```
table.close();
conn.close();
```

JDBC API: ResultSet

- ResultSet object is used to get Table Data in read-only mode.
- A cursor is maintained to identify the current row.
- One extra row “Before First” and another extra row “After Last” is added in the table in the object.



159	Raj	8.0
148	Hardik	6.2
139	Yash	8.5

- By Default the cursor stays in the “Before First” row.
- Cursor movement and Data Access can be done using various methods of ResultSet.

JDBC API: ResultSet

Note: *Replace Xxx with appropriate data types like Int, String, etc., depending on the data being retrieved or updated*

Methods of ResultSet

Method Signature	Description
`boolean next()`	Moves the cursor to the next row in the result set.
`boolean previous()`	Moves the cursor to the previous row in the result set.
`boolean first()`	Moves the cursor to the first row.
`boolean last()`	Moves the cursor to the last row.
`void beforeFirst()`	Moves the cursor to before the first row.
`void afterLast()`	Moves the cursor to after the last row.

`xxx getXxx(int columnIndex)`
`Ex.: String getString(int columnIndex)`
	Retrieves an integer value from the specified column.
	Retrieves a string value from the specified column.

`boolean absolute(int row)`	Moves the cursor to an absolute row number.
`boolean relative(int rows)`	Moves the cursor relative to the current position.
`void close()`	Closes the ResultSet, releasing associated resources.

Exception Handling in JDBC

- All the above functions from JDBC API throws an exception **SQLException (Checked Exception)**, which needs to be handled for every program.
- Class.forName() method throws **ClassNotFoundException**, if class is not found from the given path. This also needs to be handled.

Closing the Connection

- After all the operations are performed over the created connection, to reduce the overhead and improve data security, the connection must be closed.
- If it is not closed explicitly then the connection will be closed automatically when the program terminates but better practice is to do it beforehand explicitly.

```
table.close(); //required only if ResultSet is created.
conn.close();
```

JDBC Example

```
import java.sql.*;
public class DBConnection {
   public static void main(String[] args) {
        try {
                Class.forName("com.mysql.jdbc.Driver");
                Connection conn =
 DriverManager.getConnection("jdbc:mysql://localhost/StudentDB", "root", "");
                System.out.println("Connection established successfully.");
                Statement stmt = conn.createStatement();
                boolean result = stmt.execute("SELECT * FROM Students;");
                if(result)
                        System.out.println("Table Returned.");
                else
                        System.out.println("No Data.");
                conn.close();
        } catch (SQLException se) {
                System.out.println(se.getMessage());
                se.printStackTrace();
        }
   }
}
```

Transaction Management in JDBC

- Sequence of 1 or more steps treated as indivisible unit of work is known as transaction.
- ACID property.
  - Atomicity
  - Consistency
  - Isolation
  - Integrity
- Operations in transactions can be Reading, Modifying, inserting and deleting data in Database.
- Transaction involves 3 main operations: Commit, Rollback and Savepoint.

Transaction Management in JDBC: Autocommit

- While working with database, java provides auto commit option, which automatically reflects the updates done by the last executed query on the database directly.
- While working with transactions, auto-commit must be disabled to preserve atomicity.
- Connection object is responsible for that.
- Following method is used:

```
public void setAutoCommit(boolean c);
```

- Example:

```
conn.setAutoCommit(false);
```

Transaction Management in JDBC: Commit

- Commit command will write the changes done from previously executed queries from a temporary to the permanent one.
- Connection object is again responsible for commit.
- Method:

```
public void commit();
```

- Example:

```
conn.commit();
```

- This can be executed after the transaction is completed.

Transaction Management in JDBC: Rollback

- If there's an error occurred while executing transaction after some statements and needs to be reset, rollback should be done.
- It will simply reset the database to the state from before the transaction started.
- Connection object is again responsible for rollback.
- Method:

```
public void rollback();
```

- Example:

```
conn.rollback();
```

- This can be executed when some error occurs while executing the transaction.

Transaction Management in JDBC: Savepoint

- If a transaction is larger and sometimes its not needed to rollback to the start of the transaction but somewhere in between, the savepoint can be created on transaction.
- Connection object is again responsible for creating savepoint.
- Savepoint class is available in java.sql package to save the savepoint.
- Method:

```
public Savepoint setSavepoint();
```

- Example:

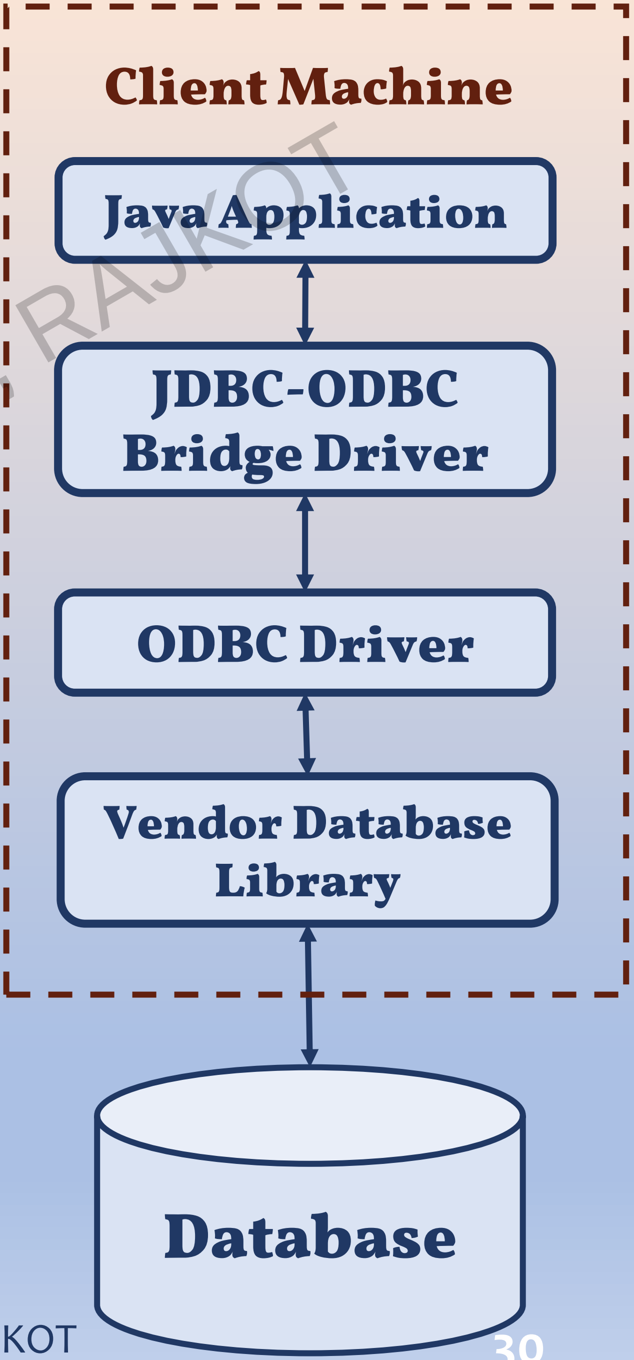
```
Savepoint sp = conn.setSavepoint();
// Other SQL Statements execution
// Error occurred after a while.
conn.rollback(sp);
```

JDBC Drivers

- Mainly 4 types of Drivers:
    - Type-1 Driver: JDBC-ODBC Bridge Driver
    - Type-2 Driver: Native API Driver
    - Type-3 Driver: Network Protocol Driver / Middleware Driver
    - Type-4 Driver: Pure Java Driver / Direct-to-Database Driver / Thin Driver

JDBC Drivers: Type-1 (Deprecated)

- Open Source database connectivity
- More number of entities for communication, hence slower.
- Supports almost all the Databases, but support is limited to the operations common to all.
- Client machine needs to be installed with DSN for ODBC Drivers.
- Ex.: ODBC Data source in Microsoft Windows.

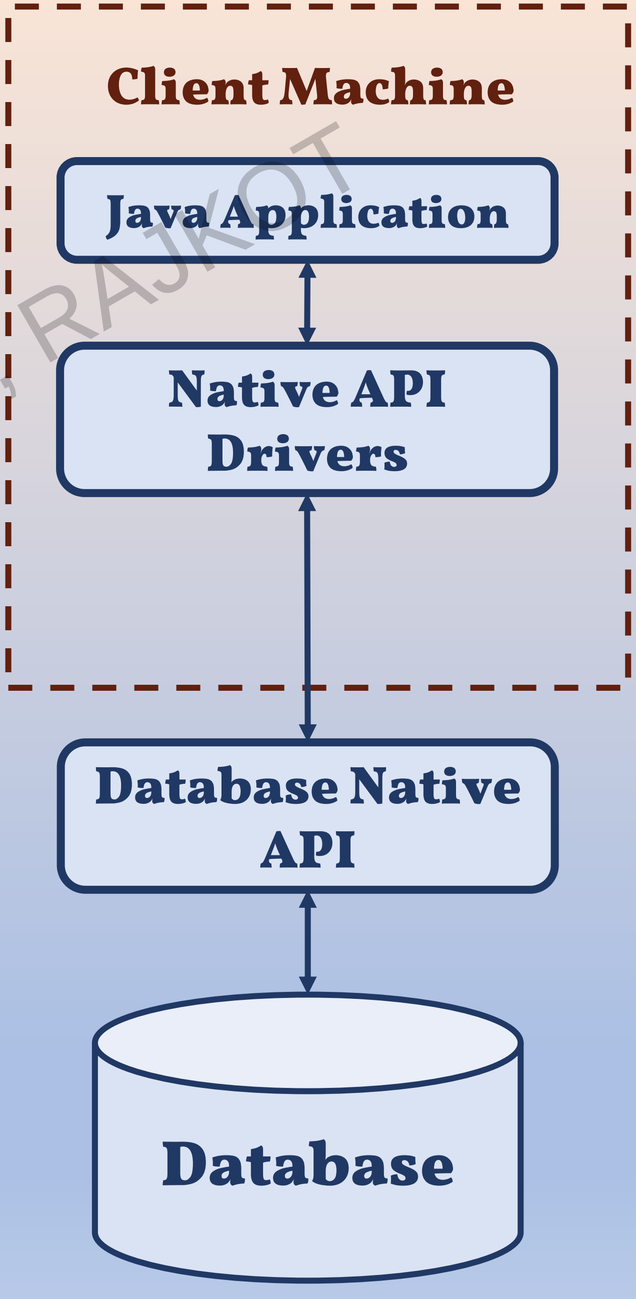


JDBC Drivers: Type-1 (Deprecated)

- Advantages:
  - Almost any database for which an ODBC driver is installed can be accessed, and data can be retrieved.
- Disadvantages:
  - Performance overhead since the calls have to go through the JDBC Overhead Bridge to the ODBC driver, then to the native database connectivity interface (thus may be slower than other types of drivers).
  - The ODBC driver needs to be installed on the client machine.
  - Not suitable for Applets, because the ODBC driver needs to be installed on the client and any client PC won't allow any installations on Client PC from browser. Also the browser can't access data of that driver.

JDBC Drivers: Type-2

- Also known as Native API Driver
- JDBC (Java) calls are directly converted into Native C/C++ (Database API) calls, which are unique to database.
- Less number of Calls
- Vendor Specific
- Ex.: Oracle Connection Interface (OCI) drivers for Java

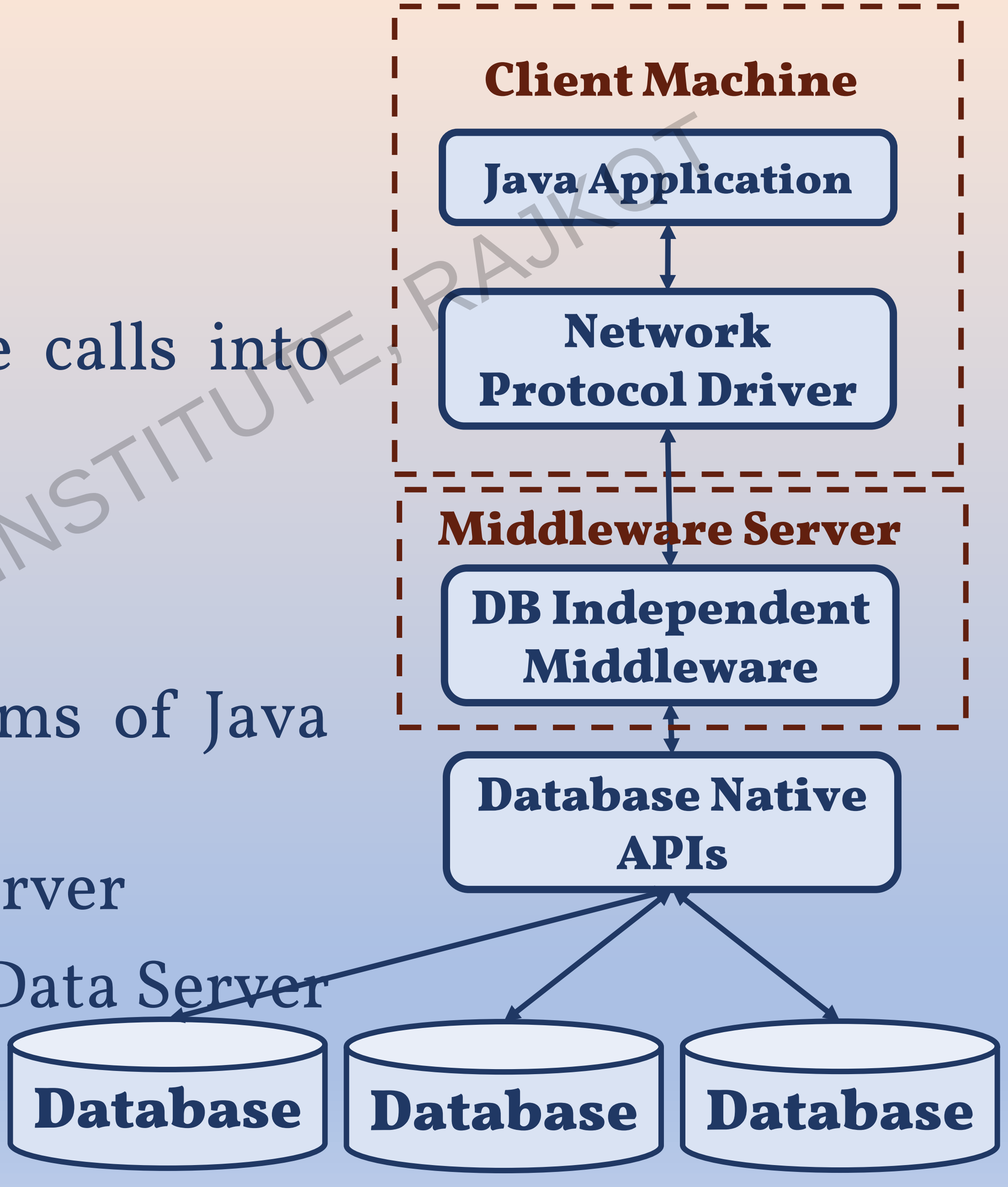


JDBC Drivers: Type-2

- Advantages:
  - Good performance due to native database communication and optimizations.
  - Access to database-specific features and optimizations.
  - Supports a variety of databases with available native libraries.
- Disadvantages:
  - Native libraries are platform-dependent, complicating cross-platform deployment.
  - Tied to a specific database vendor, limiting portability.
  - Requires client-side installation and configuration of native libraries.
  - Doesn't fully comply with JDBC standards, potentially impacting driver and database interchangeability.
  - Introduces security risks if native code isn't properly managed.
  - Complex maintenance with the need for updates or patches to native libraries.

JDBC Drivers: Type-3

- Middleware / Network Protocol Driver
- A server is used to translate Database calls into database specific calls.
- Network Communication is involved.
- Platform Independence
- Database Vendor Independence in terms of Java Code.
- Additional Security from Middleware server
- Examples: DataDirect SequeLink, IBM Data Server Driver for JDBC

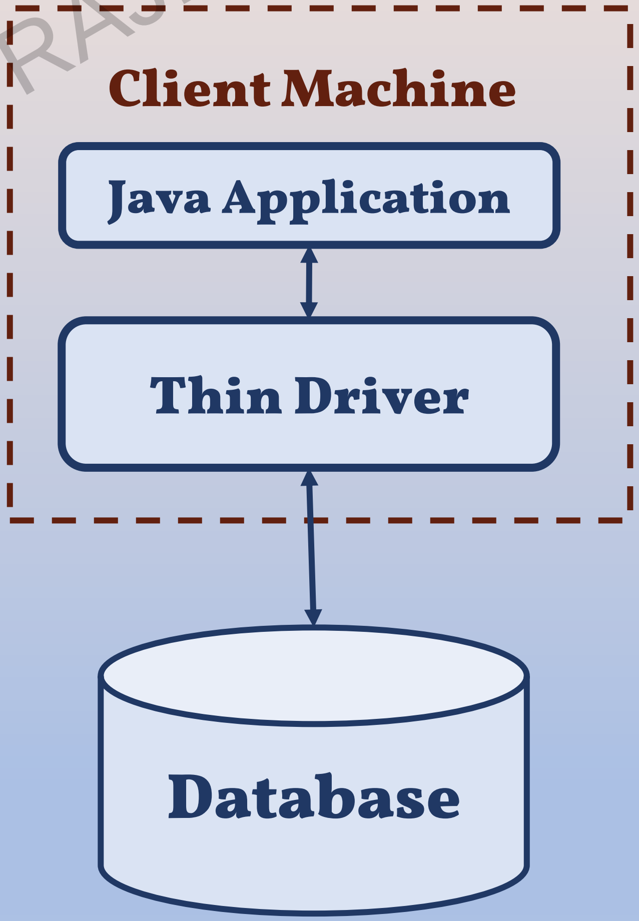


JDBC Drivers: Type-3

- Advantages:
  - Database vendor independence and platform independence through middleware.
  - Centralized security control and authentication.
  - Flexible deployment options for scalability.
- Disadvantages:
  - Performance overhead due to network communication and translation.
  - Installation and configuration complexity of middleware.
  - Limited support for database-specific features and optimizations.

JDBC Drivers: Type-4

- Thin Driver / Direct to Database Driver / Purely Java Driver.
- Written Purely in java, hence providing best performance.
- Vendor specific, hence full database support.
- Direct communication between database protocol, no middle layer involved.
- No Installation required in client machine.
- Platform Independent.
- Usually very small size driver.



JDBC Drivers: Type-4

- Advantages:
  - Excellent performance with direct database communication and support for database-specific features.
  - Platform and database vendor independence.
  - Easy deployment and scalability.
  - Robust security through database server authentication.
- Disadvantages:
  - Limited support for network-related features like load balancing and failover.
  - May require the database-specific driver to be included in the application, potentially increasing application size.
  - The need for specific database drivers can limit database portability.