Implementing Interfaces

When a class implements an interface, it must provide the implementations for all the methods declared in the interface (unless the class is declared as abstract). Implementing an interface is a way for a class to commit to performing certain behaviors.

Extending Interfaces

An interface can extend another interface, which allows it to inherit the method declarations from another interface. This means the new interface will have all the methods from the interface it extends, plus any additional methods it declares. Note that interfaces can extend multiple interfaces, allowing for a form of multiple inheritance that is not allowed with classes.

Implementing an Interface

```
interface Movable {
    void move();
}

class Car implements Movable {
    public void move() {
        System.out.println("Car moves");
    }
}
```

Extending an Interface

```
interface Movable {
    void move();
}
interface Flyable extends Movable {
    void fly();
}
class Airplane implements Flyable {
    public void move() {
        System.out.println("Airplane moves on the runway");
    }
    public void fly() {
        System.out.println("Airplane flies");
    }
}
```

Implementing Multiple Interfaces

```
interface Movable {
    void move();
}
interface Stoppable {
    void stop();
}
class Bicycle implements Movable, Stoppable {
    public void move() {
        System.out.println("Bicycle moves");
    }
    public void stop() {
        System.out.println("Bicycle stops");
    }
}
```

Extending Multiple Interfaces

```
interface Movable {
    void move();
}
interface Stoppable {
    void stop();
}
interface Operable extends Movable, Stoppable {
    void operate();
}
class Machine implements Operable {
    public void move() {
        System.out.println("Machine moves");
    }
    public void stop() {
        System.out.println("Machine stops");
    }
    public void operate() {
        System.out.println("Machine operates");
    }
}
```

Implementing interfaces - extending interfaces

Scenario: Online Store Product Catalog System

Imagine we are building an online store system that requires a product catalog. The catalog needs to manage different types of products such as electronics, books, and clothing. Each product category has some common attributes like ID, name, and price, but also has category-specific attributes like the author for books, size for clothing, and warranty for electronics.

We'll create interfaces for common operations such as listing the product details and calculating the final price after taxes and discounts, and we'll extend interfaces to accommodate category-specific attributes and behaviors.

Interfaces:

- Product: A general interface for all products that includes methods for getting basic information.
- Taxable: An interface for taxable items that can calculate tax-related methods.
- Discountable: An interface for items that can be discounted, providing methods to apply discounts.

Extending Interfaces:

The Taxable and Discountable interfaces could extend the Product interface to include common operations. Each product category will have its own interface that extends from Product and adds category-specific methods.

```
interface Product {
    String getId();
    String getName();
    double getPrice();
    String getDetails();
}
interface Taxable extends Product {
    double calculateTax();
}
interface Discountable extends Product {
    double applyDiscount(double discountRate);
}
class Electronics implements Taxable, Discountable {
    private String id;
    private String name;
    private double price;
    private double warrantyYears;
```

```
    // Constructor, getters and setters

    @Override
    public String getDetails() {
        // Return details specific to Electronics, including warranty information
    }

    @Override
    public double calculateTax() {
        // Implement tax calculation for electronics
    }

    @Override
    public double applyDiscount(double discountRate) {
        // Apply discount to the price
    }
}
```

```
// Similar classes can be created for Books and Clothing, implementing the appropriate interfaces

public class OnlineStore {
    public static void main(String[] args) {
        Electronics phone = new Electronics("001", "SuperPhone", 999.99, 2);

        System.out.println("Product Details: " + phone.getDetails());
        System.out.println("Tax Amount: $" + phone.calculateTax());
        System.out.println("Price after Discount: $" + phone.applyDiscount(0.1));
    }
}
```

JDBC

Java Database Connectivity (JDBC) is an Application Programming Interface (API) used to connect Java application with Database. JDBC is used to interact with the various type of Database such as Oracle, MS Access, My SQL and SQL Server and it can be stated as the platform-independent interface between a relational database and Java programming.

JDBC:

- JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

- The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage:
  - ✓ Making a connection to a database.
  - ✓ Creating SQL or MySQL statements.
  - ✓ Executing SQL or MySQL queries in the database.
  - ✓ Viewing & Modifying the resulting records.

- Components of JDBC:
  I. JDBC API
  II. JDBC Driver manager
  III. JDBC Test suite
  IV. JDBC-ODBC Bridge Drivers

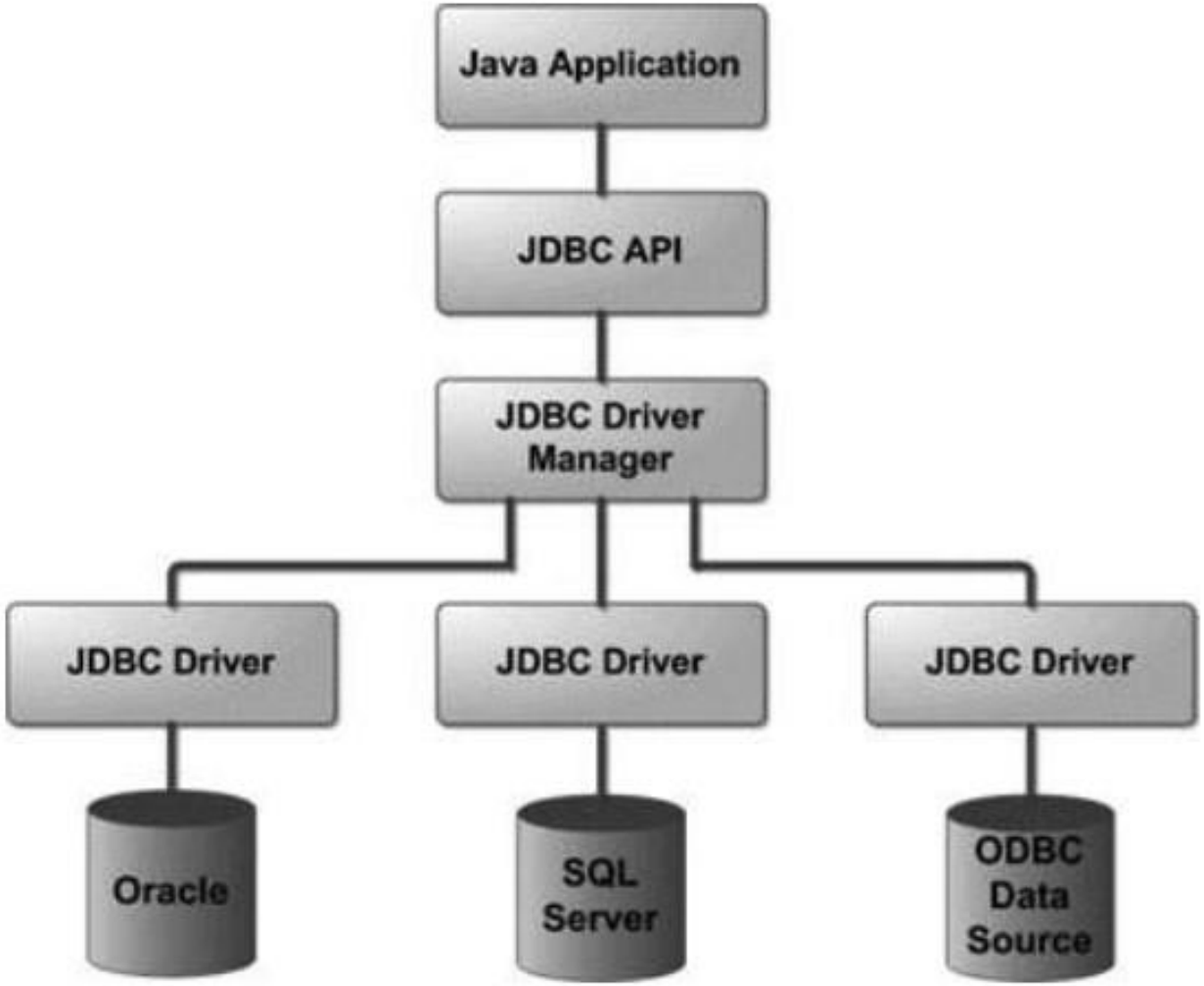
JDBC API: It provides various methods and interfaces for easy communication with the database. The java.sql package contains interfaces and classes of JDBC API.

JDBC Driver manager: It loads a database-specific driver in an application to establish a connection with a database. It is used to make a database-specific call to the database to process the user request.

JDBC Test suite: It is used to test the operation(such as insertion, deletion, updation) being performed by JDBC Drivers.

JDBC-ODBC Bridge Drivers: It connects database drivers to the database. This bridge translates the JDBC method call to the ODBC function call. It makes use of the sun.jdbc.odbc package which includes a native library to access ODBC characteristics.

Architecture of JDBC:



Why Should We Use JDBC:

- Before JDBC, ODBC API was the database API to connect and execute the query with the database.

- But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured).

- That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

What is API:

- API (Application programming interface) is a document that contains a description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc.
- Through an API one application can request information or perform a function from another application without having direct access to it's underlying code or the application data.

JDBC Drivers:

VIT-AP UNIVERSITY

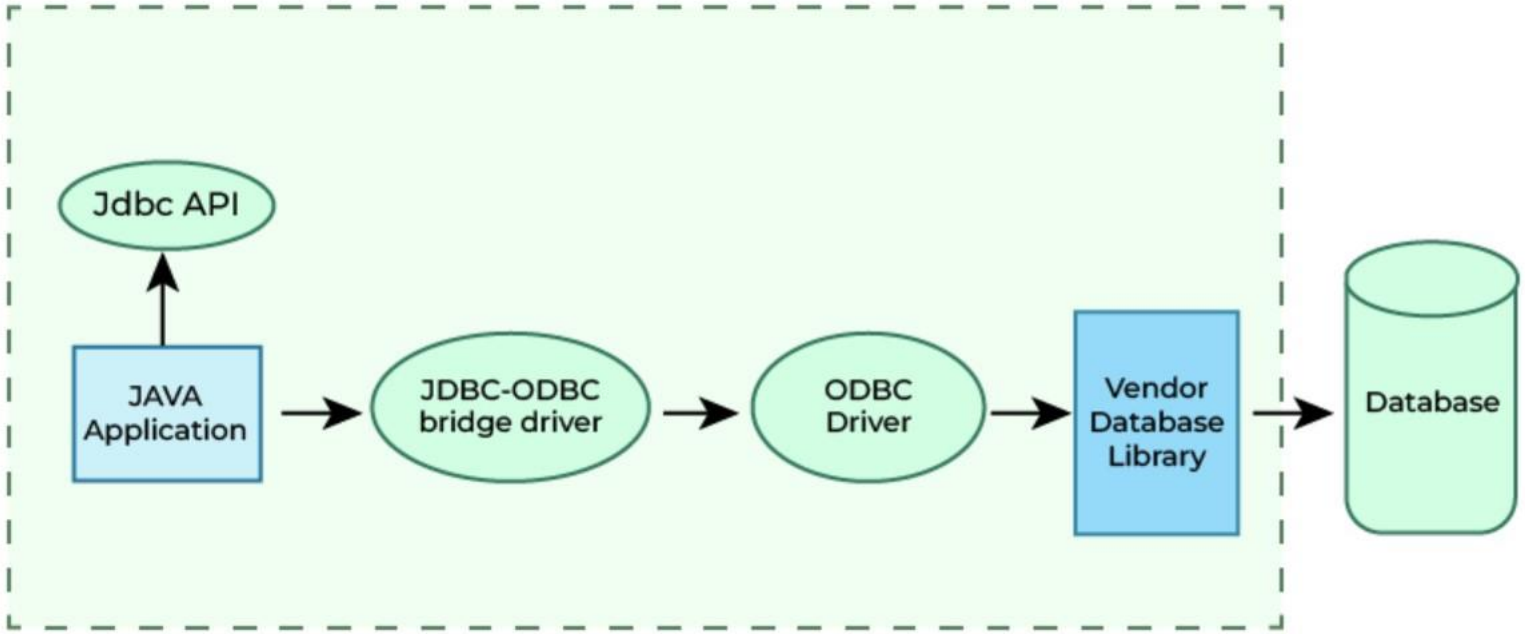
- JDBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand.

- There are 4 types of JDBC drivers:

  I. Type-1 driver [JDBC-ODBC bridge driver]
  II. Type-2 driver [Native-API driver (partially java driver)]
  III. Type-3 driver [Network Protocol driver (fully java driver)]
  IV. Type-4 driver [Thin driver (fully java driver)]

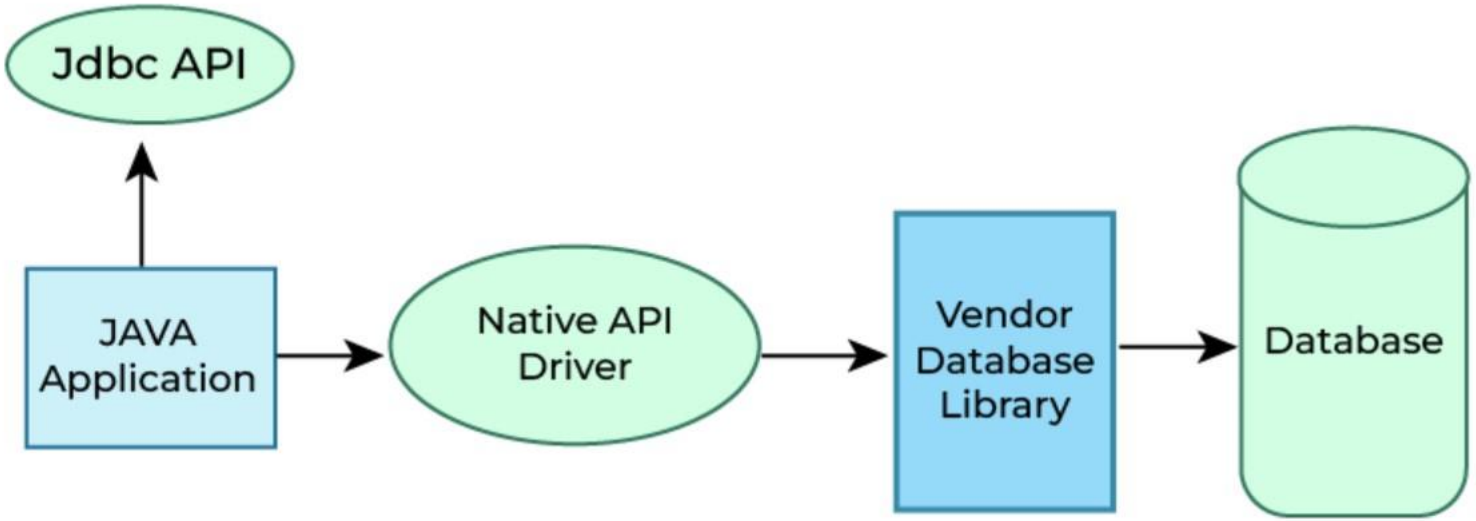
(1) JDBC-ODBC bridge driver:

- JDBC-ODBC bridge driver uses ODBC driver to connect to the database.
- The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls.
- Type-1 driver is also called Universal driver because it can be used to connect to any of the databases. It is a database independent driver.



(2) Native-API driver:

- The Native API driver uses the client -side libraries of the database.
- This driver converts JDBC method calls into native calls of the database API.
- In order to interact with different database, this driver needs their local API, that's why data transfer is much more secure as compared to type-1 driver.
- This driver is not fully written in Java that is why it is also called Partially Java driver. It is a database dependent driver.

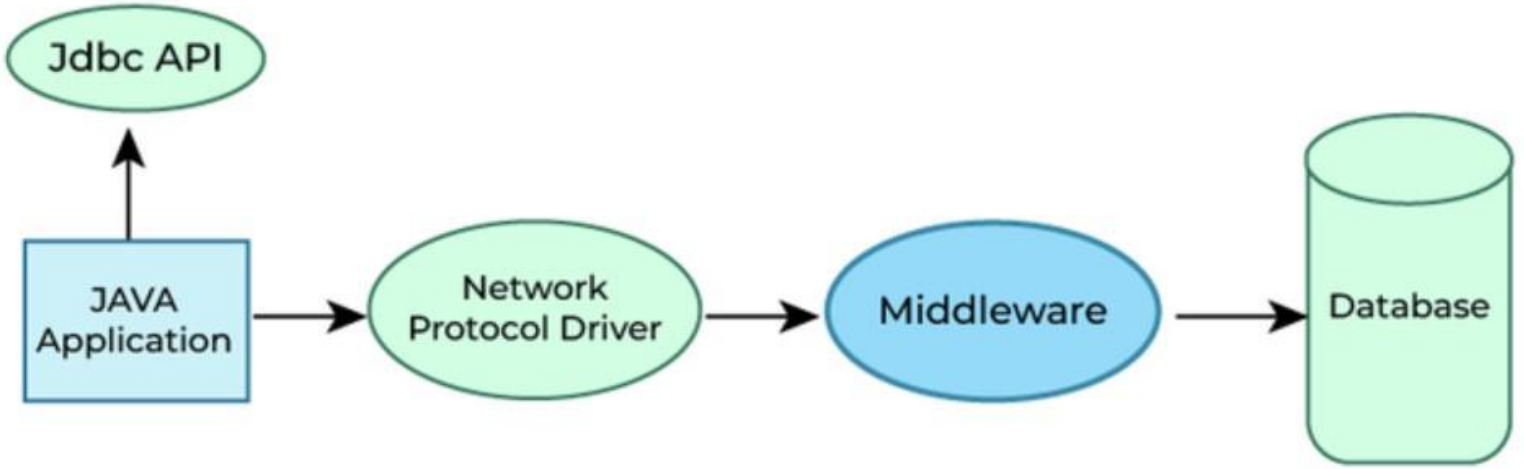


(3) Network Protocol driver:

- The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol.

- Here all the database connectivity drivers are present in a single server, hence no need of individual client-side installation.

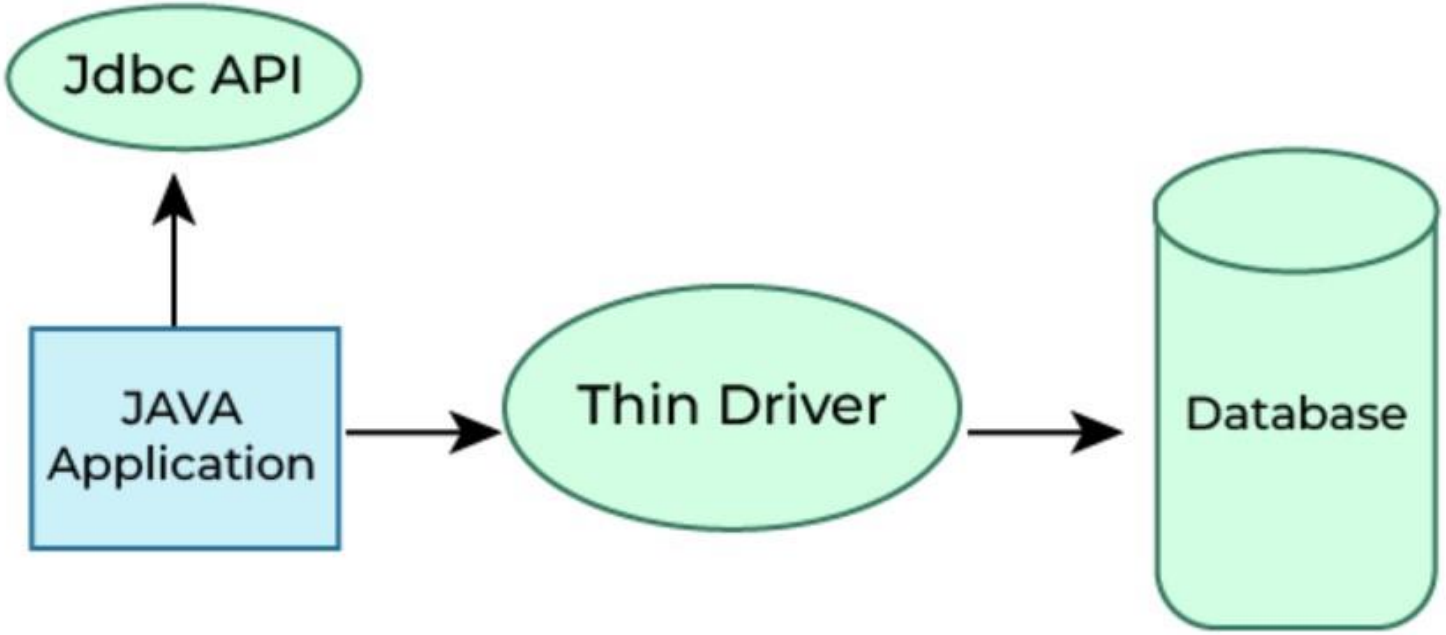
- It is fully written in java.



(3) Thin driver:

- The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver.

- It is fully written in Java language.



Steps to Connect Java Application with Database:

VIT-AP UNIVERSITY

I. Register the Driver class

II. Create connection

III. Create statement

IV. Execute queries

V. Close connection

Loading the drivers

- The forName() method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

- The following example uses Class.forName() to load the Oracle driver as shown below as follows:

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

Establish a connection using the Connection class object

- The getConnection() method of DriverManager class is used to establish connection with the database.

```
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","password");
```

Where oracle is the database used, thin is the driver used, @localhost is the IP Address where a database is stored, 1521 is the port number and xe is the service provider.

Create a statement

- The createStatement() method of Connection interface is used to create statement.

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

Here, con is a reference to Connection interface used in previous step .

Execute the query

- The executeQuery() method of Statement interface is used to execute queries to the database.

- This method returns the object of ResultSet that can be used to get all the records of a table.

```
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
{
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

Close the connection

- By closing the connection, objects of Statement and ResultSet will be closed automatically.

- The close() method of the Connection interface is used to close the connection. It is shown below as follows:

```
con.close();
```

```
import java.sql.*;
class MysqlCon{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");

Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/sonoo","root","root");

Statement stmt=con.createStatement();

ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));

con.close();
}
catch(Exception e){ System.out.println(e);}  } }
```