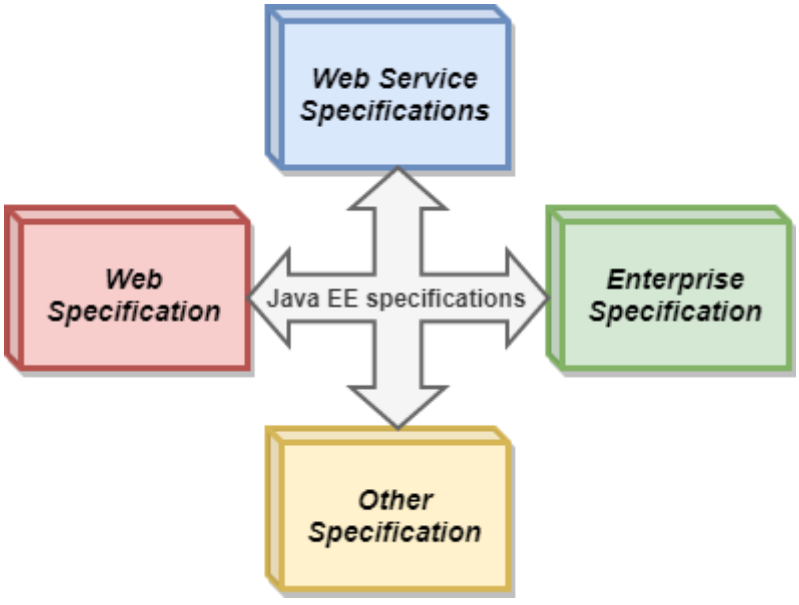
Java EE

The Java EE stands for Java Enterprise Edition, which was earlier known as J2EE and is currently known as Jakarta EE. It is a set of specifications wrapping around Java SE (Standard Edition). The Java EE provides a platform for developers with enterprise features such as distributed computing and web services. Java EE applications are usually run on reference run times such as microservers or application servers. Examples of some contexts where Java EE is used are e-commerce, accounting, banking information systems.

Specifications of Java EE

Java EE has several specifications which are useful in making web pages, reading and writing from database in a transactional way, managing distributed queues. The Java EE contains several APIs which have the functionalities of base Java SE APIs such as Enterprise JavaBeans, connectors, Servlets, Java Server Pages and several web service technologies.



1. Web Specifications of Java EE

- *Servlet*- This specification defines how you can manage HTTP requests either in a synchronous or asynchronous way. It is low level, and other specifications depend on it
- *WebSocket*- WebSocket is a computer communication protocol, and this API provides a set of APIs to facilitate WebSocket connections.

- *Java Server Faces*- It is a service which helps in building GUI out of components.
- *Unified Expression Language*- It is a simple language which was designed to facilitate web application developers.

2. Web Service Specifications of Java EE

- *Java API for RESTful Web Services*- It helps in providing services having Representational State Transfer schema.
- *Java API for JSON Processing*- It is a set of specifications to manage the information provided in JSON format.
- *Java API for JSON Binding*- It is a set of specifications provide for binding or parsing a JSON file into Java classes.
- *Java Architecture for XML Binding*- It allows binding of xml into Java objects.
- *Java API for XML Web Services*- SOAP is an xml-based protocol to access web services over http. This API allows you to create SOAP web services.

3. Enterprise Specifications of Java EE

- *Contexts and Dependency Injection*- It provides a container to inject dependencies as in Swing.
- *Enterprise JavaBean*- It is a set of lightweight APIs that an object container possesses in order to provide transactions, remote procedure calls, and concurrency control.
- *Java Persistence API*- These are the specifications of object-relational mapping between relational database tables and Java classes.
- *Java Transaction API*- It contains the interfaces and annotations to establish interaction between transaction support offered by Java EE. The APIs in this abstract from low-level details and the interfaces are also considered low-level.
- *Java Message Service*- It provides a common way to Java program to create, send and read enterprise messaging system's messages.

4. Other Specifications of Java EE

- *Validation*- This package contains various interfaces and annotations for declarative validation support offered by Bean Validation API.

- *Batch applications*- It provides the means to run long running background tasks which involve a large volume of data and which need to be periodically executed.
- *Java EE Connector Architecture*- This is a Java-based technological solution for connecting Java servers to Enterprise Information System.

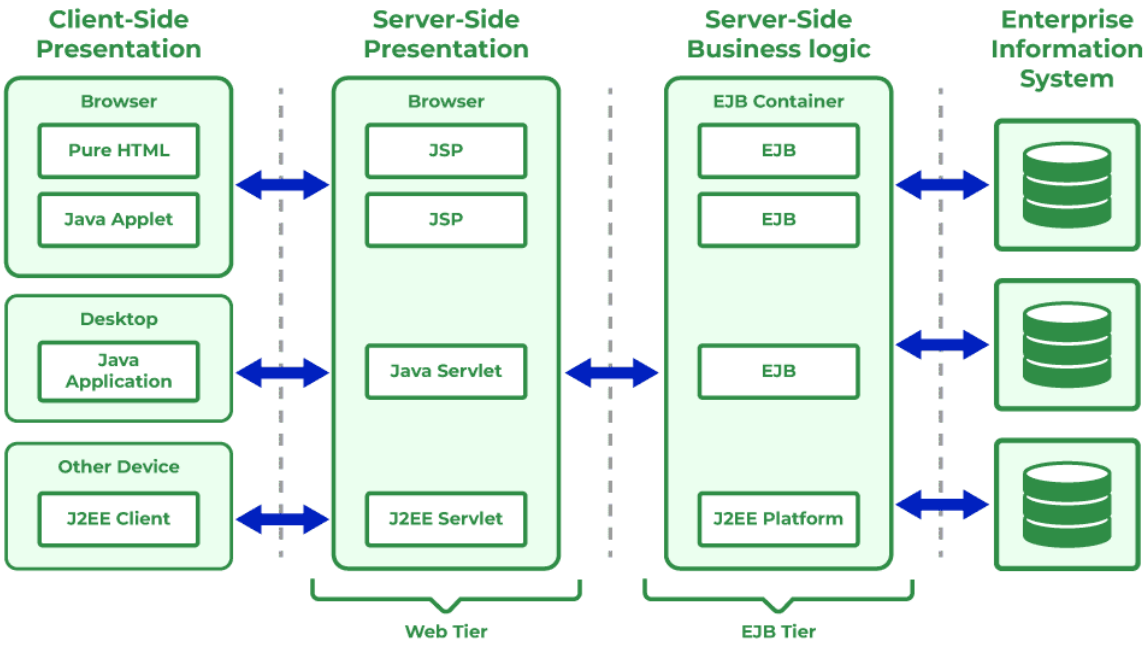
Java SE vs Java EE

Java SE refers to standard edition and contains basic functionalities and packages required by a beginner or intermediate-level programmer. Java EE is an enhanced platform and a wrapper around Java SE. It has the edge over Java SE an also has a variety of aspects in which it outshines other features.

Java SE	Java EE
Java SE provide basic functionalities such as defining types and objects.	Java EE facilitates development of large-scale applications.
SE is a normal Java specification	EE is built upon Java SE. It provides functionalities like web applications, and Servlets.
It has features like class libraries, deployment environments, etc.	Java EE is a structured application with a separate client, business, and Enterprise layers.
It is mostly used to develop APIs for Desktop Applications like antivirus software, game, etc.	It is mainly used for developing web applications.
Suitable for beginning Java developers.	Suitable for experienced Java developers who build enterprise-wide applications.
It does not provide user authentication.	It provides user authentication.

J2EE Multitier Architecture

A tier is an abstract concept that defines a group of technologies that provide one or more services to its clients. In multi-tier architecture, each tier contains services that include software objects, DBMS, or connectivity to legacy systems. IT departments of corporations employ multi-tier architecture because it's a cost-effective way to build an application that is flexible, scalable, and responsive to the expectation of clients.



The functionality of the application is divided into logical components that are associated with a tier. Each component is a service that is built and maintained independently of the other services. Services are bound together by a communication protocol that enables service and sends information from and to other services.

Multi-Tier Architecture

Multi-tier architecture is composed of:

Clients: It Refers to a program that requests service from a component.

Resource: A resource is anything a component needs to provide a service.

Component: It is part of a tier that consists of a collection of classes or a program that performs a function to provide a service.

Containers: It is software that manages a component and provides system services to the component. A container handles persistence, resource management, security, threading, and other system-level services from components that are associated with the container. Example: APIs.

J2EE Architecture

J2EE is a four-tier architecture that consists of:

1. Client Tier/ Presentation Tier
2. Web Tier
3. Enterprise JavaBeans Tier/ Business Tier
4. Enterprise Information Systems Tier

1. Client Tier/ Presentation Tier

It consists of programs that interact with the users. These components prompt the user for the input conversion as a request to forward to software on the component that processes the request and returns the result to the client program. Components under Client tier in J2EE Specification:

- *Applet Client*- It is a component used by Web clients that operate within Applet Container. It is a Java-enabled web browser.
- *Application Client* –It is a Java application that operates within the application container. It is the Java Runtime Environment (JRE).
- *Rich client*- This does not come under J2EE Specification. This is not considered a component of the client because it can be written in a language other than Java. J2EE does not define containers for Rich clients but these clients access any tier in the environment using HTTP, SOAP, and ebXML protocols.
- Clients are classified by the technology used to access the components and resources.

Categories of Client Tier: There are 5 categories of clients

- Web Client,
- Java Beans Client
- EIS Client
- Web-service peers and
- Multi-tier client.

2. Web Tier

It provides the Internet functionality to the J2EE application. The components on the web tier use HTTP to receive requests and send responses to the client that resides on the tier. Web-Tier Provides services to Client-tier using HTTP. The responsibilities act as an intermediary between components working on the web tier and other tiers and the client tier. The intermediary activities are:

- Accepting info from other s/w sent using POST, GET, and PUT using HTTP.
- Transmit data such as images and dynamic content.

Components of Web Tier: Two components that work on the web tier are mentioned below:

- Servlets
- Java server pages (JSP)

3. Enterprise Java Beans Tier Implementation

It contains business logic for J2EE applications. In this tier two or more EJB reside. It is a keystone for every J2EE Application. EJB beans are contained in the EJB server- which is a distributed object server that works on the EJB tier and manages transactions, and security, and ensures multi-threading and persistence when EJB are accessed. EJB tier provides concurrency, scalability, life cycle management, and fault-tolerant (Back-Up) to the J2EE applications. EJB-Tier manages (Creating/Destroying, also moving components in/out of memory) the instances of the components. Collectively EJB-Server and EJB-Container are responsible for providing low-level system services required to implement the business logic of Enterprise Java Beans.

Some of the system services include Resource Pooling, Distributed object protocols, thread management, State Management, Process Management, Object persistence, Security, and Deployment Configuration.

4. Enterprise Information System Tier

EIS tier links the J2EE application to resources and legacy systems that are available on the corporate backbone network. It is the tier where the J2EE applications directly or indirectly interface with a variety of technologies including DBMS and Mainframes. Components on the EIS communicate to resources using CORBA or Java Connectors.

Access Control List (ACL)

- It controls communication between tiers.
- It acts as a bridge on different virtual networks because it adds security to web Applications.
- ACL prevents hackers to access DBMS and similar resources.

EIS-Tier provides connectivity to the resources that are not part of J2EE- which includes resources such as Legacy Systems, DBMS, and systems provided by third-party. Provides flexibility to developers to connect resources that are not part of J2EE using CORBA and Java Connectors or through proprietary protocols.

API Full Form in Java

In the context of Java and software development in general, the term "API" stands for "Application Programming Interface."

An API in Java, or any other programming language, defines a set of rules and protocols that allow different software entities (like applications or libraries) to communicate with each other. It provides a set of methods and tools for building software applications, enabling them to interact with external systems or components.

In simpler terms, an API specifies how software components should interact, allowing developers to use pre-defined functions to perform specific tasks without needing to understand the complexities of how those tasks are implemented.

For instance, when you use Java's built-in ArrayList class, you're actually leveraging the API provided by Java to perform operations like adding elements, removing elements, or retrieving elements from the list without needing to know the intricate details of how the list operations are implemented behind the scenes.

What is API in Java with Example?

An API (Application Programming Interface) in Java refers to a collection of pre-written classes, methods, functions, and variables that developers can use to build software applications. It provides a set of rules and protocols that allow different software entities to communicate with each other. Essentially, APIs define how software components should interact.

Types of Java APIs

Java APIs (Application Programming Interfaces) play a crucial role in software development, enabling communication between various software components. Understanding the different types of Java APIs is essential for developers to harness the power of Java effectively. Let's delve deeper into the various types:

1) Public or Open Java API:
Description: These are APIs provided by Java itself as part of the Java Development Kit (JDK). They are freely available for developers to use without any restrictions.

Usage: Developers utilize these APIs to build applications by leveraging the functionalities provided directly by Java. Examples include the Java Standard Edition (SE) API, which offers classes for tasks like file handling, networking, and data structures.

2) Private or Internal Java API:
Description: This category refers to APIs developed by individual organizations or developers for specific purposes within their applications. They are not meant for public use and are designed for internal consumption.

Usage: Companies or developers create these APIs to maintain proprietary functionalities, ensuring that only authorized personnel can access or modify them.

3) Partner Java API:

Description: Partner APIs are third-party APIs that organizations offer to their partners for particular use cases or collaborations. These APIs facilitate integration between different entities, fostering seamless interactions between systems.

Usage: Organizations use partner APIs to share specific functionalities or data securely with their business partners, enhancing collaboration and interoperability.

4) Composite Java API:

Description: A composite API comprises multiple microservices developed using various types of Java APIs. Instead of relying on a single API type, developers integrate functionalities from different Java APIs to create comprehensive solutions.

Usage: By combining multiple Java APIs, developers can build complex applications or systems that leverage diverse functionalities, ensuring flexibility and scalability.

5) Web Java API (Java REST API):

Description: Web Java APIs, particularly RESTful APIs, facilitate communication between browser-based applications and external resources over

the HTTP protocol. REST (Representational State Transfer) is an architectural style that uses standard HTTP methods to access and manipulate resources.

Usage: Developers utilize Java REST APIs to create client-server architectures where Java applications provide services accessible to client applications via HTTP requests. These APIs enable interactions with various services like data storage, authentication, and more, making them essential for modern web development.

J2EE components

The J2EE Specification describes four types of components that can be created by a developer. Each component is a modular software unit that is deployed within the application server and interacts with its host environment and other components through the J2EE APIs that are available to it.

The APIs available to the components determine the facilities that the components have access to.

Applet component
These are client-side GUI components that are hosted by an applet container, which is typically a web browser. They have the ability to provide a feature-rich user interface to an enterprise application.

Application client component
These are Java-based programs that can execute within a supported JVM. They offer a user interface similar to what exists on the native client while accessing J2EE Business Tier facilities.

Web component
These are components that have the ability to respond to HTTP requests. They comprise servlets, JSP pages, filters, and web event listeners:

Servlets
These extend aweb server's functionality to support the dynamic processing of application logic. Servlets can be used in any request/response programming models but are most often used with HTTP. In this method, they utilize data embedded within HTTP requests for input and provide resulting output using the HTTP response, making the processing facilities of the J2EE Platform available to the information input on the source web page.

JavaServer Pages
Similar to servlets, JSP pages provide dynamic web application capability to the J2EE Platform. JSP pages achieve this goal through the introduction of templating functionality. A JSP page begins with HTML elements that remain static throughout the lifetime of the page; this forms the base template. Dynamic elements are then added through the embedding of Java code and/or special tags

within the page. It should be noted that JSP pages are indeed converted to servlets before execution, but the intricacies of this process are managed by the web container.

Filters

These components provide the ability to intercept requests by clients for resources on a server and also responses provided to these requests by the web container with the goal of transforming the content of either the request or response communication. Filters provide a reusable mechanism to handle recurring tasks such as authentication, logging, etc., that are applicable to items within the web container.

Web event listeners

These are components that are created to perform a particular function when a specific event occurs within the web container. These components provide developers with the flexibility to respond in a certain way when different types of web application–related events such as the creation or invalidation of a session occurs.

Enterprise JavaBeans components

These are server-side components that singularly or collectively encapsulate the application logic of an enterprise application.

EJB technology enables rapid and simplified development of distributed, transactional, secure, and portable applications based on Java technology. There are three types of EJBs defined by the specification:

Session beans

A session bean is a server-side extension of a client that exists to service requests made by the client. As the name implies, it simulates an interactive session between a client and the server-based component. Session beans exist in two forms. There are stateful session beans, which are said to maintain a "conversational state" with a client by virtue of retaining instance variable data on multiple method invocations. Because of this, they are wedded to a unique client throughout the instance existence. Stateless session beans, on the other hand, exist to service requests from multiple clients. They perform transient

services for their clients, fulfilling whatever request is made of them and returning to their original initialized states.

Entity beans

Entity beans are an encapsulated representation of business objects made available with a persistence mechanism. They represent an object to relational mapping of the data stored within these persistence layers, thereby facilitating the modification of the underlying business objects while preserving referential and data integrity constraints demanded by the specific business process.

Message-driven beans

These are stateless, server-side components invoked by the EJB container on the receipt of a message in an associated JMS Queue or Topic. This component allows the EJB container to provide support for asynchronous message processing.

JDBC

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a specification from Sun Microsystems that provides a standard abstraction (API or Protocol) for Java applications to communicate with various databases. It provides the language with Java database connectivity standards. It is used to write programs required to access databases. JDBC, along with the database driver, can access databases and spreadsheets. The enterprise data stored in a relational database (RDB) can be accessed with the help of JDBC APIs.

Definition of JDBC (Java Database Connectivity)
JDBC is an API (Application programming interface) used in Java programming to interact with databases. The classes and interfaces of JDBC allow the application to send requests made by users to the specified database.

Purpose of JDBC
Enterprise applications created using the JAVA EE technology need to interact with databases to store application-specific information. So, interacting with a database requires efficient database connectivity, which can be achieved by using the ODBC (Open database connectivity) driver. This driver is used with JDBC to interact or communicate with various kinds of databases such as Oracle, MS Access, Mysql, and SQL server database.

Components of JDBC
There are generally four main components of JDBC through which it can interact with a database. They are as mentioned below:

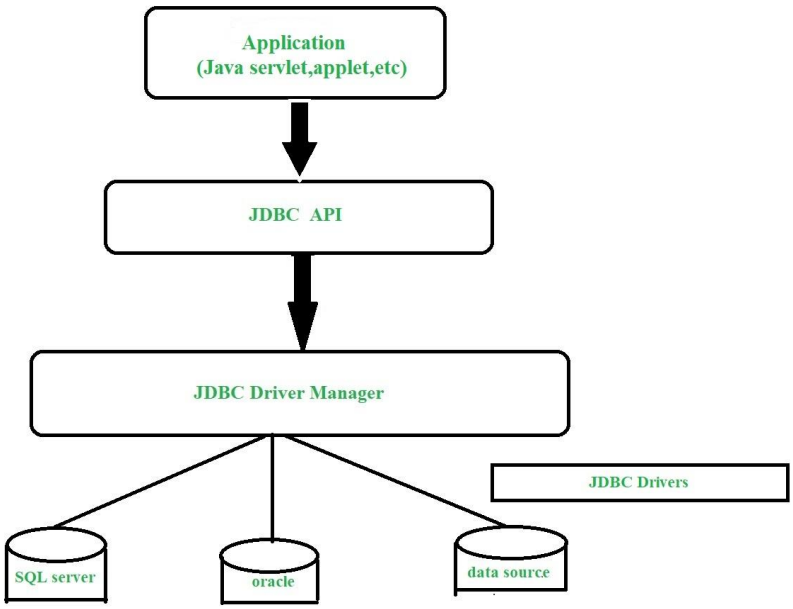
1. JDBC API: It provides various methods and interfaces for easy communication with the database. It provides two packages as follows, which contain the java SE and Java EE platforms to exhibit WORA (write once run anywhere) capabilities. The java.sql package contains interfaces and classes of JDBC API.

2. 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.

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

4. 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



Description:

Application: It is a java applet or a servlet that communicates with a data source.

The JDBC API: The JDBC API allows Java programs to execute SQL statements and retrieve results. Some of the important interfaces defined in JDBC API are as

follows: Driver interface, ResultSet Interface, RowSet Interface, PreparedStatement interface, Connection inteface, and cClasses defined in JDBC

API are as follows: DriverManager class, Types class, Blob class, clob class.

DriverManager: It plays an important role in the JDBC architecture. It uses some database-specific drivers to effectively connect enterprise applications to databases.

JDBC drivers: To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.

Types of JDBC Architecture (2-tier and 3-tier)

The JDBC architecture consists of two-tier and three-tier processing models to access a database. They are as described below:

Two-tier model: A java application communicates directly to the data source. The JDBC driver enables the communication between the application and the data source. When a user sends a query to the data source, the answers for those queries are sent back to the user in the form of results.

The data source can be located on a different machine on a network to which a user is connected. This is known as a client/server configuration, where the user's machine acts as a client, and the machine has the data source running acts as the server.

Three-tier model: In this, the user's queries are sent to middle-tier services, from which the commands are again sent to the data source. The results are sent back to the middle tier, and from there to the user.

This type of model is found very useful by management information system directors.

What is API?

Before jumping into JDBC Drivers, let us know more about API.

API stands for Application Programming Interface. It is essentially a set of rules and protocols which transfers data between different software applications and allow different software applications to communicate with each other. 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 API uses JDBC Drivers to connect with the database.

JDBC Drivers

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:

1. Type-1 driver or JDBC-ODBC bridge driver
2. Type-2 driver or Native-API driver (partially java driver)
3. Type-3 driver or Network Protocol driver (fully java driver)
4. Type-4 driver or Thin driver (fully java driver)

Interfaces of JDBC API

A list of popular interfaces of JDBC API is given below:

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface
- Classes of JDBC API

A list of popular classes of JDBC API is given below:

- DriverManager class
- Blob class

- Clob class
- Types class

Working of JDBC

Java application that needs to communicate with the database has to be programmed using JDBC API. JDBC Driver supporting data sources such as Oracle and SQL server has to be added in java application for JDBC support which can be done dynamically at run time. This JDBC driver intelligently communicates the respective data source.

Creating a simple JDBC application:

```
//Java program to implement a simple JDBC application
package com.vinayak.jdbc;

import java.sql.*;

public class JDBCDemo {

    public static void main(String args[])
        throws SQLException, ClassNotFoundException
    {
        String driverClassName
            = "sun.jdbc.odbc.JdbcOdbcDriver";
        String url = "jdbc:odbc:XE";
        String username = "scott";
        String password = "tiger";
        String query
```

```
            =   "insert   into   students   values(109,
'bhatt')";

        // Load driver class
        Class.forName(driverClassName);

        // Obtain a connection
        Connection con = DriverManager.getConnection(
            url, username, password);

        // Obtain a statement
        Statement st = con.createStatement();

        // Execute the query
        int count = st.executeUpdate(query);
        System.out.println(
            "number of rows affected by this query= "
            + count);

        // Closing the connection as per the
        // requirement with connection is completed
        con.close();
    }
} // class
```

The above example demonstrates the basic steps to access a database using JDBC. The application uses the JDBC-ODBC bridge driver to connect to the database.

You must import java.sql package to provide basic SQL functionality and use the classes of the package.

What is the need of JDBC?

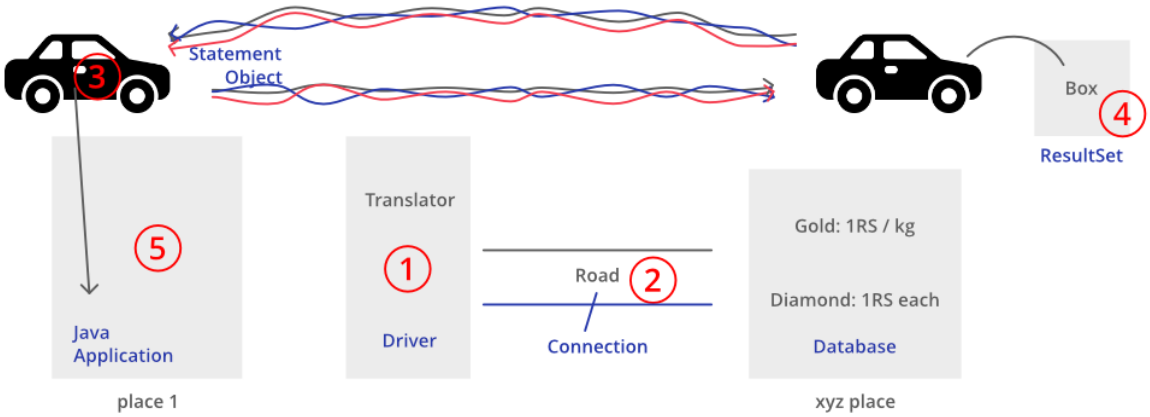
JDBC is a Java database API used for making connection between java applications with various databases. Basically, JDBC used for establishing stable database connection with the application API. To execute and process relational database queries (SQL or Oracle queries), multiple application can connect to different types of databases which supports both standard (SE) and enterprise (EE) edition of java.

JDBC Connection in Java

What is JDBC?

JDBC is an acronym for Java Database Connectivity. It's an advancement for ODBC (Open Database Connectivity). JDBC is a standard API specification developed in order to move data from the front end to the back end. This API consists of classes and interfaces written in Java. It basically acts as an interface (not the one we use in Java) or channel between your Java program and databases i.e it establishes a link between the two so that a programmer can send data from Java code and store it in the database for future use.

Illustration: Working of JDBC co-relating with real-time



Why JDBC Come into Existence?

As previously told JDBC is an advancement for ODBC, ODBC being platform-dependent had a lot of drawbacks. ODBC API was written in C, C++, Python, and Core Java and as we know above languages (except Java and some part of Python) are platform-dependent. Therefore, to remove dependence, JDBC was developed by a database vendor which consisted of classes and interfaces written in Java.

Steps to Connect Java Application with Database

Below are the steps that explains how to connect to Database in Java:

Step 1 – Import the Packages

Step 2 – Load the drivers using the forName() method

Step 3 – Register the drivers using DriverManager

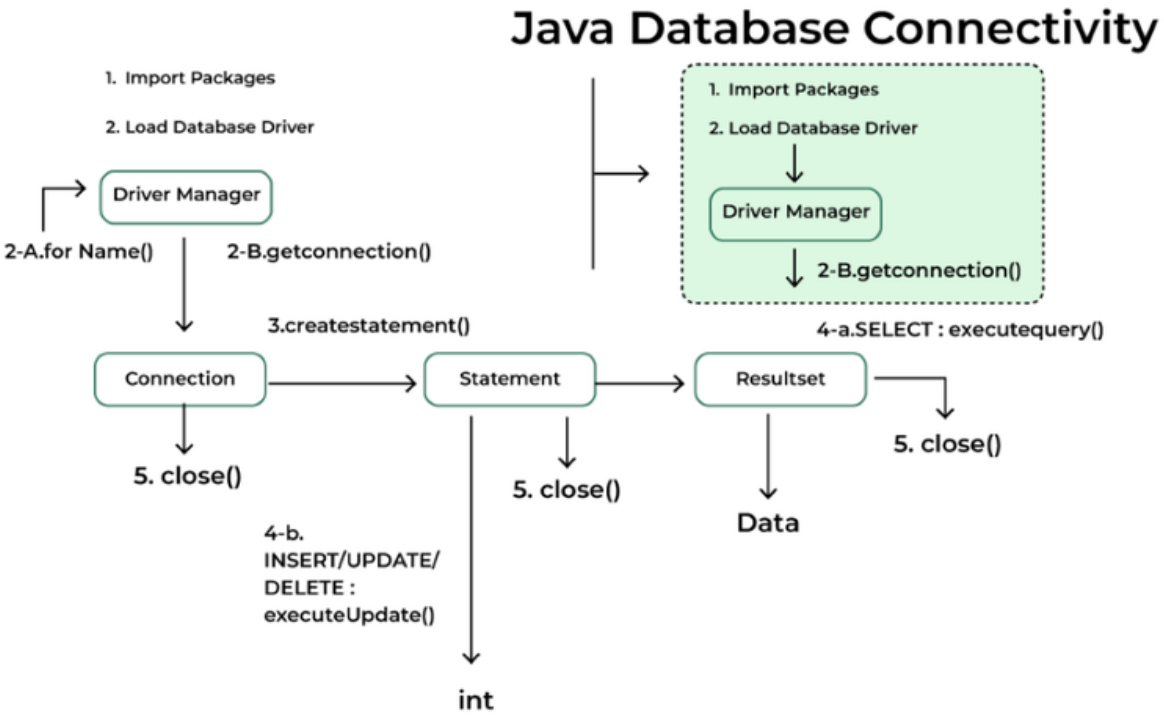
Step 4 – Establish a connection using the Connection class object

Step 5 – Create a statement

Step 6 – Execute the query

Step 7 – Close the connections

Java Database Connectivity



Let us discuss these steps in brief before implementing by writing suitable code to illustrate connectivity steps for JDBC.

Step 1: Import the Packages

Step 2: Loading the drivers

In order to begin with, you first need to load the driver or register it before using it in the program. Registration is to be done once in your program. You can register a driver in one of two ways mentioned below as follows:

2-A Class.forName()

Here we load the driver's class file into memory at the runtime. No need of using new or create objects. The following example uses Class.forName() to load the Oracle driver as shown below as follows:

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

2-B DriverManager.registerDriver()

DriverManager is a Java inbuilt class with a static member register. Here we call the constructor of the driver class at compile time. The following example uses DriverManager.registerDriver()to register the Oracle driver as shown below:

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

Step 3: Establish a connection using the Connection class object

After loading the driver, establish connections as shown below as follows:

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

user: Username from which your SQL command prompt can be accessed.

password: password from which the SQL command prompt can be accessed.

con: It is a reference to the Connection interface.

Url: Uniform Resource Locator which is created as shown below:

```
String url = " jdbc:oracle:thin:@localhost:1521:xe"
```

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. All 3 parameters above are of String type and are to be declared by the programmer before calling the function. Use of this can be referred to form the final code.

Step 4: Create a statement

Once a connection is established you can interact with the database. The JDBCStatement, CallableStatement, and PreparedStatement interfaces define the methods that enable you to send SQL commands and receive data from your database.

Use of JDBC Statement is as follows:

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

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

Step 5: Execute the query

Now comes the most important part i.e executing the query. The query here is an SQL Query. Now we know we can have multiple types of queries. Some of them are as follows:

The query for updating/inserting a table in a database.

The query for retrieving data.

The executeQuery() method of the Statement interface is used to execute queries of retrieving values from the database. This method returns the object of ResultSet that can be used to get all the records of a table.

The executeUpdate(sql query) method of the Statement interface is used to execute queries of updating/inserting.

Pseudo Code:

```
int m = st.executeUpdate(sql);
if (m==1)
    System.out.println("inserted successfully: "+sql);
else
    System.out.println("insertion failed");
```

Step 5: Execute the query

Now comes the most important part i.e executing the query. The query here is an SQL Query. Now we know we can have multiple types of queries. Some of them are as follows:

The query for updating/inserting a table in a database.

The query for retrieving data.

The executeQuery() method of the Statement interface is used to execute queries of retrieving values from the database. This method returns the object of ResultSet that can be used to get all the records of a table.

The executeUpdate(sql query) method of the Statement interface is used to execute queries of updating/inserting.

Pseudo Code:

```
int m = st.executeUpdate(sql);
if (m==1)
    System.out.println("inserted successfully: "+sql);
else
    System.out.println("insertion failed");
```

Statements in JDBC

The Statement interface in JDBC is used to create SQL statements in Java and execute queries with the database. There are different types of statements used in JDBC:

1. Create Statement
2. Prepared Statement
3. Callable Statement.

1. Create a Statement:

A Statement object is used for general-purpose access to databases and is useful for executing static SQL statements at runtime.

Syntax:

```
Statement statement = connection.createStatement();
```

Implementation: Once the Statement object is created, there are three ways to execute it.

boolean execute(String SQL): If the ResultSet object is retrieved, then it returns true else false is returned. Is used to execute SQL DDL statements or for dynamic SQL.

int executeUpdate(String SQL): Returns number of rows that are affected by the execution of the statement, used when you need a number for INSERT, DELETE or UPDATE statements.

ResultSet executeQuery(String SQL): Returns a ResultSet object. Used similarly as SELECT is used in SQL.

2. Prepared Statement:

A PreparedStatement represents a precompiled SQL statement that can be executed multiple times. It accepts parameterized SQL queries, with ? as placeholders for parameters, which can be set dynamically.

Illustration:

Considering in the people database if there is a need to INSERT some values, SQL statements such as these are used:

INSERT INTO people VALUES ("Ayan",25);

INSERT INTO people VALUES("Kriya",32);

To do the same in Java, one may use Prepared Statements and set the values in the ? holders, setXXX() of a prepared statement is used as shown:

```
String query = "INSERT INTO people(name, age)VALUES(?, ?)";

PreparedStatement pstmt = con.prepareStatement(query);

pstmt.setString(1,"Ayan");

ptstmt.setInt(2,25);

// where pstmt is an object name
```

Implementation: Once the PreparedStatement object is created, there are three ways to execute it:

execute(): This returns a boolean value and executes a static SQL statement that is present in the prepared statement object.

executeQuery(): Returns a ResultSet from the current prepared statement.

executeUpdate(): Returns the number of rows affected by the DML statements such as INSERT, DELETE, and more that is present in the current Prepared Statement.

3. Callable Statement:

A CallableStatement is used to execute stored procedures in the database. Stored procedures are precompiled SQL statements that can be called with parameters. They are useful for executing complex operations that involve multiple SQL statements.

Syntax: To create a CallableStatement,

```
CallableStatement cstmt = con.prepareCall("{call ProcedureName(?, ?)}");
```

{call ProcedureName(?, ?)}: Calls a stored procedure named ProcedureName with placeholders ? for input parameters.

Methods to Execute:

execute(): Executes the stored procedure and returns a boolean indicating whether the result is a ResultSet (true) or an update count (false).

executeQuery(): Executes a stored procedure that returns a ResultSet.

executeUpdate(): Executes a stored procedure that performs an update and returns the number of rows affected.

JDBC Result Set

Java Database Connectivity is Java-based technology and that provides a standard API for accessing databases in Java applications. The Key Component of Java Database Connectivity is the ResultSet. JDBC driver allows developers to read and manipulate data from the database.

The JDBC ResultSet is Object which represents the result of a SQL Query executed on a database. It acts as a cursor to navigate through the retrieved data, manipulate data, Fetching Specific columns and others. In this article, we will learn more about JDBC Result Set.

JDBC Result Set in Java

The ResultSet is essentially a table of data where each row represents a record and each column represents a field in the database. The ResultSet has a cursor that points to the current row in the ResultSet and we can able to navigate in ResultSet by using the next(), previous(), first(), and last() methods. We can retrieve data by using different methods like getString(), getInt(), getDouble() and other methods.

In Java, the ResultSet is the Object which is used for holding the result of a database query typically the SQL select statement. And It is the part of JDBC API which is used for interacting with relational databases. The ResultSet allows us over the rows of tables returned by the SQL query and extract a specific column from the SQL query result.

Syntax:

```
try {
      Connection                    conn                  =
DriverManager.getConnection(url, username, password);
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery("SELECT * FROM
your_table");
```

```
        while (rs.next()) {
             // Process the result set
         }

        rs.close();
        stmt.close();
        conn.close();
}
catch (SQLException e) {
        e.printStackTrace(); // Handle the exception
}
```

Common Operations with ResultSet:

Fetching data from database: We can fetch data from the database based on the requirements by using conditional statements.

Navigating the ResultSet: We can able navigating the ResultSet by using methods like next(), previous(), first(), and last().

Getting column values: We can fetch column values with specific conditions or without conditions.

Closing the result: Once database operations are completed we need close the connections related to database here we close the ResultSet connection. By using close method.

Types of ResultSet

There are three different characteristics by which ResultSet types are differentiated

Scrollability: Determines whether you can move back and forth in the ResultSet

TYPE_FORWARD_ONLY: Can only move forward through the rows

TYPE_SCROLL_INSENSITIVE: Can move forward and backward but changes are not reflect ResultSet

TYPE_SCROLL_SENSITIVE: Can move forward and backward but changes are affect the ResultSet

Concurrency: Determines whether you can update the ResultSet

CONCUR_READ_ONLY: Can only read data

CONCUR_UPDATABLE: Allows updates to the ResultSet

Holdability: Determines what happens to the ResultSet when a Transaction is committed.

HOLD_CURSORS_OVER_COMMIT: The ResultSet remains open after a commit

CLOSE_CURSORS_AT_COMMIT: The ResultSet closes after a commit