## ¿Qué es Java JDBC?

**JDBC** son las siglas de **Java Database Connectivity** (Conectividad de Base de Datos de Java). Es una **API (Application Programming Interface)** de Java que permite a las aplicaciones Java interactuar con bases de datos relacionales.

En términos más simples, JDBC es como el "puente" que conecta tu programa Java con una base de datos (como MySQL, PostgreSQL, Oracle, SQL Server, etc.). Proporciona un conjunto de clases e interfaces estándar que los desarrolladores de Java utilizan para:

1. **Establecer una conexión** con una base de datos.
2. **Enviar sentencias SQL** (Structured Query Language) a la base de datos.
3. **Procesar los resultados** devueltos por la base de datos.

### ¿Por qué es necesario JDBC?

Imagina que tienes una aplicación Java que necesita guardar y recuperar información de una base de datos. Sin JDBC, tendrías que escribir código específico para cada tipo de base de datos (uno para MySQL, otro para Oracle, etc.), lo cual sería un caos.

JDBC resuelve esto al proporcionar una **interfaz uniforme**. Esto significa que, una vez que aprendes a usar JDBC, puedes escribir código Java para interactuar con **cualquier base de datos relacional**, siempre y cuando haya un **driver JDBC** disponible para esa base de datos. El driver es el componente específico que traduce las llamadas JDBC estándar de Java a los protocolos y comandos que entiende cada base de datos particular.

## Componentes Clave de JDBC

La API JDBC se compone principalmente de las siguientes clases e interfaces, ubicadas en el paquete java.sql (y javax.sql para funcionalidades más avanzadas como DataSource):

1. **DriverManager**:
   * Una clase que gestiona un conjunto de **drivers JDBC**.
   * Es el punto de partida para establecer una conexión a la base de datos.
   * Históricamente se usaba mucho su método getConnection(). Hoy en día, con la llegada de **DataSource y los pools** de conexiones, su uso directo ha disminuido para la gestión de conexiones.
2. **Connection**:
   * Una **interfaz** que representa una sesión activa con una base de datos específica.
   * A través de un objeto Connection, puedes crear sentencias SQL, gestionar transacciones y obtener metadatos de la base de datos.
   * Es el primer objeto que obtienes después de establecer la conexión.
3. **Statement**:
   * Una **interfaz** que se utiliza para ejecutar sentencias SQL estáticas (sin parámetros) en la base de datos.
   * Es simple, pero **menos eficiente y más propenso a inyección SQL** si se concatenan cadenas.
4. **PreparedStatement**:
   * Una **interfaz** que extiende Statement. Es la forma **preferida** de ejecutar sentencias SQL.
   * Permite ejecutar sentencias SQL precompiladas con parámetros (usando ?), lo que mejora el rendimiento y, lo más importante, **previene ataques de inyección SQL**.
   * Ideal para sentencias que se ejecutan varias veces o con valores de entrada del usuario.
5. **CallableStatement**:
   * Una **interfaz** que extiende PreparedStatement. Se utiliza para ejecutar **procedimientos almacenados (stored procedures)** y funciones en la base de datos.
6. **ResultSet**:
   * Una **interfaz** que representa el conjunto de datos devuelto por una consulta SQL (SELECT).
   * Permite iterar sobre las filas del resultado y acceder a los valores de las columnas por índice o por nombre de columna.
7. **DataSource (del paquete javax.sql)**:
   * Una **interfaz** más moderna y flexible para obtener conexiones, especialmente útil con **pools de conexiones**.
   * Proporciona una forma estándar de obtener conexiones sin preocuparse por los detalles de DriverManager. Los pools de conexiones (como Apache Commons DBCP, HikariCP) implementan esta interfaz.
8. **JDBC Driver**:
   * No es parte de la API de Java, sino un **software de terceros** (suministrado por el proveedor de la base de datos o por proyectos de código abierto) que implementa las interfaces JDBC para una base de datos específica.
   * Es el "traductor" entre el código JDBC estándar de Java y el protocolo de comunicación de la base de datos.

## Ciclo de Vida Básico de una Aplicación JDBC

El flujo típico de una aplicación Java que usa JDBC para interactuar con una base de datos es el siguiente:

1. **Cargar el Driver:** Aunque con JDBC 4.0 y versiones posteriores, el DriverManager suele descubrir los drivers automáticamente (si están en el classpath), tradicionalmente se usaba Class.forName("com.mysql.cj.jdbc.Driver"); (o el driver correspondiente).
2. **Establecer la Conexión:**
   * Usando DriverManager.getConnection(url, user, password); (método antiguo).
   * **Preferentemente**, usando un DataSource configurado (como BasicDataSource de Apache Commons DBCP):  
     Java  
     DataSource ds = /\* ... obtener un DataSource configurado ... \*/;  
     Connection connection = ds.getConnection();
3. **Crear una Sentencia:**
   * Statement stmt = connection.createStatement();
   * **Preferiblemente**, PreparedStatement pstmt = connection.prepareStatement("SELECT \* FROM users WHERE id = ?");
4. **Ejecutar la Sentencia:**
   * Para consultas (SELECT): ResultSet rs = stmt.executeQuery("SELECT \* FROM users"); o ResultSet rs = pstmt.executeQuery();
   * Para actualizaciones (INSERT, UPDATE, DELETE): int rowsAffected = stmt.executeUpdate("INSERT INTO users VALUES (...)"); o int rowsAffected = pstmt.executeUpdate();
5. **Procesar los Resultados (si es una consulta):**
   * Iterar sobre el ResultSet usando rs.next() para moverse a la siguiente fila.
   * Obtener los valores de las columnas usando métodos como rs.getString("nombre\_columna"), rs.getInt("id\_columna"), etc.
6. **Cerrar los Recursos:**
   * **CRUCIAL:** Cerrar los recursos en el orden inverso en que se abrieron para liberar los recursos de la base de datos y evitar fugas de memoria. Esto se hace en bloques finally o, mejor aún, usando **try-with-resources** (disponible desde Java 7).
   * rs.close();
   * stmt.close(); (o pstmt.close();)
   * connection.close();

## Ejemplo Básico (Concepto sin gestión de errores completa)

Java

import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.PreparedStatement;  
import java.sql.ResultSet;  
import java.sql.SQLException;  
  
public class JdbcExample {  
  
 // URL de conexión a la base de datos (ejemplo para MySQL)  
 static final String DB\_URL = "jdbc:mysql://localhost:3306/mi\_base\_de\_datos";  
 static final String USER = "usuario";  
 static final String PASS = "contraseña";  
  
 public static void main(String[] args) {  
 // try-with-resources asegura que los recursos se cierren automáticamente  
 try (Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);  
 PreparedStatement pstmt = conn.prepareStatement("SELECT id, nombre, email FROM usuarios WHERE edad > ?")) {  
  
 // 1. Establecer el valor del parámetro  
 pstmt.setInt(1, 18); // Establece el valor para el primer '?'  
  
 // 2. Ejecutar la consulta  
 ResultSet rs = pstmt.executeQuery();  
  
 // 3. Procesar los resultados  
 while (rs.next()) {  
 // Recuperar los datos por nombre de columna  
 int id = rs.getInt("id");  
 String nombre = rs.getString("nombre");  
 String email = rs.getString("email");  
  
 // Imprimir los valores  
 System.out.println("ID: " + id + ", Nombre: " + nombre + ", Email: " + email);  
 }  
  
 // El ResultSet, PreparedStatement y Connection se cierran automáticamente  
 // debido al try-with-resources.  
  
 } catch (SQLException e) {  
 e.printStackTrace(); // Manejo de errores básico: imprimir la pila de llamadas  
 }  
 }  
}

**Nota:** Para ejecutar este código, necesitarías el driver JDBC de MySQL (o de la base de datos que uses) en tu classpath. Por ejemplo, mysql-connector-java.jar o mysql-connector-j.jar.