JDBC - Preparazione Esame

**1. Cos'è JDBC?**

**JDBC (Java Database Connectivity)** **è un'API** *(Un'****API*** *(Application Programming Interface) è un insieme di regole e specifiche che permettono a diversi software di comunicare tra loro.)* **di Java** che consente di collegare un'applicazione Java a un database, permettendo di eseguire query, recuperare dati e elaborare i risultati. JDBC facilita la comunicazione tra il programma Java e il database.

**FAST RISPOSTA**

JDBC (Java Database Connectivity) è un'API di Java che definisce un insieme di interfacce e classi per consentire alle applicazioni Java di interagire con i database. Per connettersi a un database MySQL, è necessario un driver JDBC specifico, chiamato **MySQL Connector/J**, che agisce da intermediario tra l'applicazione Java e il database MySQL.

--------------------------------------------------------------------------------------------------------------------

**2. Cosa serve per collegare un database MySQL a un programma Java?**

Per collegare un database MySQL a un programma Java, è necessario il **MySQL Connector/J**, un driver JDBC specifico per MySQL. Questo driver consente al programma Java di comunicare con MySQL.

**Dove inserire il MySQL Connector?** Il connettore MySQL va incluso nel progetto come una libreria. Può essere scaricato dal sito ufficiale o incluso nel progetto tramite strumenti di gestione delle dipendenze come Maven o Gradle. Se non utilizzi un sistema di gestione delle dipendenze, devi aggiungere manualmente il .jar del connettore alla tua cartella di librerie.

**FAST RISPOSTA**

Il **MySQL Connector/J** può essere aggiunto al progetto Java in due modi principali:

1. **Manuale (tramite Build Path):** Puoi scaricare il file JAR del MySQL Connector e aggiungerlo manualmente al build path del progetto. In questo caso, il JAR viene inserito generalmente nella cartella lib del progetto, e successivamente dovrai configurare l'ambiente di sviluppo (ad esempio Eclipse, IntelliJ, etc.) affinché lo riconosca come una libreria del progetto.
2. **Automatica (tramite Maven):** Un modo più semplice ed efficiente è utilizzare **Maven**, uno strumento di gestione delle dipendenze. Maven ti permette di aggiungere la dipendenza nel file pom.xml, e automaticamente scarica la versione corretta del MySQL Connector dalla repository centrale di Maven, evitando di gestire manualmente i file JAR. Maven si occupa anche della gestione delle versioni e delle dipendenze transitive, semplificando così il processo di configurazione e riducendo il rischio di conflitti tra versioni.

--------------------------------------------------------------------------------------------------------------------

**3. Come si gestisce una connessione in JDBC?**

Per gestire una connessione in JDBC:

* Crei un'istanza di Connection usando una classe come ConnectionFactory.
* Se la connessione è null, la crei con il metodo DriverManager.getConnection(), fornendo i dettagli di connessione (come il percorso del database e le credenziali).
* È utile avere un metodo per chiudere la connessione quando non serve più, per liberare le risorse.

**FAST RISPOSTA**

Per creare una connessione con JDBC, è consigliabile utilizzare una classe Java dedicata, ad esempio ConnectionFactory. La procedura prevede:

1. Creare un'istanza di Connection.
2. Controllare se la connessione è null. In tal caso, utilizzare il metodo DriverManager.getConnection() per creare una nuova connessione, fornendo i dettagli necessari come URL del database, nome utente e password.
3. Una volta terminato l'uso della connessione, è fondamentale chiuderla utilizzando il metodo close(). Questo rilascia la connessione al *connection pool*, permettendo il suo riutilizzo senza doverne creare una nuova, migliorando così l'efficienza e riducendo il consumo di risorse.

**Ricorda però che:**Quando vuoi utilizzare JDBC, hai bisogno di un **driver JDBC specifico** per il database con cui desideri interagire (ad esempio MySQL, Oracle, PostgreSQL, ecc.). Questo driver è distribuito come file JAR e include:

1. **Implementazioni delle interfacce JDBC** (come Connection, Statement, PreparedStatement, ecc.) definite dalla piattaforma Java.
2. Metodi specifici per comunicare con il database scelto.

**Facciamo chiarezza con un esempio:**

1. **File JAR**:  
   Ad esempio:
   * **Per Oracle:** ojdbc8.jar (o ojdbc11.jar per Java più recente).
   * **Per MySQL:** mysql-connector-java-x.x.x.jar.

**4. Esempio di codice per gestire la connessione**

**public** **class** **ConnectionFactory** {

**private** **static** Connection connection = **null**;

**public** **static** Connection **getConnection**() {

**if** (connection == **null**) {

**try** {

connection = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/nomeDatabase",

"username",

"password"

);

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**return** connection;

}

**public** **static** **void** **closeConnection**() {

**if** (connection != **null**) {

**try** {

connection.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

--------------------------------------------------------------------------------------------------------------------

**5. Cos'è un PreparedStatement e perché è importante?**

Un **PreparedStatement** è un tipo di oggetto che permette di eseguire query SQL parametrizzate. È più sicuro rispetto a concatenare semplicemente stringhe per creare query, perché previene attacchi come la **SQL Injection**. I parametri della query vengono sostituiti con valori sicuri, rendendo la query meno vulnerabile.

**6. Esempio di inserimento con PreparedStatement**

String queryString = "INSERT INTO passeggero (nome, cognome, eta, numeroPassaporto) VALUES (?, ?, ?, ?)";

try (Connection conn = ConnectionFactory.getConnection();

PreparedStatement preparedStatement = conn.prepareStatement(queryString)) {

preparedStatement.setString(1, passeggero.getNome());

preparedStatement.setString(2, passeggero.getCognome());

preparedStatement.setInt(3, passeggero.getEta());

preparedStatement.setString(4, passeggero.getNumeroPassaporto());

int updateRows = preparedStatement.executeUpdate();

if (updateRows > 0) {

System.out.println("Inserimento avvenuto con successo!");

} else {

System.out.println("Nessun dato inserito.");

}

} catch (SQLException e) {

e.printStackTrace();

}

**FAST RISPOSTA**

Un PreparedStatement è un oggetto in Java ottenuto dall’interfaccia PreparedStatement fornita da JDBC, utilizzata per eseguire query SQL parametrizzate.

Ad oggi i preparedStatement sono in disuso poiché nel tempo sono nate tecnologie in grado di lavorare direttamente con oggetti Java anziché scrivere manualmente SQL (ORM)

*[JDBC è ancora una tecnologia di base che viene utilizzata* ***sotto il cofano*** *anche in framework come Hibernate, quindi le nuove tecnologie si basano su queste tecnologie più vecchie]*

--------------------------------------------------------------------------------------------------------------------

**7. Cosa succede se la query non riesce a inserire i dati?**

Se updateRows > 0, l'inserimento è avvenuto con successo. Altrimenti, il messaggio "Nessun dato inserito" viene visualizzato, indicando che l'inserimento non è riuscito.

**8. Cos'è il ResultSet e a cosa serve?**

Il **ResultSet** è un oggetto che rappresenta il risultato di una query SQL di tipo **SELECT**. Quando una query restituisce dati (come una lista di righe), il ResultSet permette di scorrere i risultati e leggerli.

**9. Come funziona il metodo next() del ResultSet?**

Il metodo next() del ResultSet sposta il cursore alla riga successiva nei risultati della query. È comune usare un ciclo while con resultSet.next() per leggere tutte le righe restituite dalla query:

**while** (**resultSet**.next()) {

String nome = **resultSet**.getString("nome");

String cognome = **resultSet**.getString("cognome");

// Altre colonne...

}

**10. Differenze tra Statement, PreparedStatement e CallableStatement**

**Statement**

* Esegue comandi SQL direttamente, senza parametri.
* Vulnerabile a SQL Injection.
* Adatto per query semplici e statiche.

**PreparedStatement**

* Supporta parametri tramite ?, utilizzando valori sicuri.
* Previene vulnerabilità come la SQL Injection.
* Più efficiente per query ripetitive.

**CallableStatement**

* Usato per chiamare stored procedure (funzioni definite nel database).
* Efficiente per operazioni complesse o frequenti che è più conveniente eseguire direttamente nel database.

**Differenze principali**

| **Tipo** | **Parametri** | **Sicurezza (SQL Injection)** | **Quando usarlo** |
| --- | --- | --- | --- |
| **Statement** | No | Bassa | Per query semplici, senza input esterno |
| **PreparedStatement** | Sì | Alta | Per query dinamiche e sicure |
| **CallableStatement** | Sì | Alta | Per eseguire stored procedure |

**Esempi di Statement, PreparedStatement e CallableStatement**

**Statement:**

java

Copia codice

Statement statement = connection.createStatement();

ResultSet resultSet = statement.executeQuery("SELECT \* FROM utenti");

while (resultSet.next()) {

String nome = resultSet.getString("nome");

String email = resultSet.getString("email");

System.out.println("Nome: " + nome + ", Email: " + email);

}

**PreparedStatement:**

java

Copia codice

PreparedStatement preparedStatement = connection.prepareStatement("INSERT INTO utenti (nome, email) VALUES (?, ?)");

preparedStatement.setString(1, "Mario");

preparedStatement.setString(2, "mario@example.com");

preparedStatement.executeUpdate();

**CallableStatement:**

java

Copia codice

CallableStatement callableStatement = connection.prepareCall("{call procedure\_name(?, ?)}");

callableStatement.setString(1, "parameter1");

callableStatement.setInt(2, 123);

callableStatement.execute();