Metode Avansate de Programare
JDBC. Elemente Java 8

Arthur Molnar
arthur.molnar@ubbcluj.ro

Universitatea Babeș-Bolyai

2023

Privire de ansamblu

1. JDBC
   - Conexiunea
   - Comenzi
   - `java.sql.ResultSet`
   - Tranzacții
   - Exemple
2. Java 8
   - Expresii lambda
   - Interfețe funcționale
   - Referințe la metode

JDBC I

- API-ul Java Database Connectivity (JDBC) definește un set de interfețe Java care încapsulează funcționalitățile importante lucrului cu baze de date:
  - Conectarea la o bază de date.
  - Executarea interogărilor și a actualizărilor bazei de date.
  - Returnarea și procesarea rezultatelor obținute ca urmare a interogărilor.
  - Utilizarea informațiilor de configurare.
- JDBC ne permite să scriem aplicații care trimit interogări/comenzi SQL către orice sursă de date SQL pentru care este implementat suport JDBC (ex. Microsoft SQL Server, Postgre SQL, Oracle, etc). Astfel, putem schimba furnizorul bazei de date fără a fi necesară actualizarea codului pe partea Java.

JDBC II

- Pachetele importante:
  - `java.sql` - conține clasele și interfețele care permit accesarea și procesarea datelor stocate într-o bază de date relațională (SQL).
  - `javax.sql` - suplimentează pachetul `java.sql` și furnizează partea de funcționalitate (API) pentru accesarea și procesarea datelor pe partea de server. Acest pachet este o extensie a `java.sql`, clasele acestuia fiind bazate pe implementările din pachetul `java.sql`.

Stabilirea unei conexiuni

- Aceasta se poate realiza în două moduri:
  - Prin utilizarea clasei `java.sql.DriverManager`: este necesară încărcarea unui driver specific tipului de bază de date iar conexiunea este creată prin utilizarea unui URL.

    `jdbc.subprotocol.<database_name>`

  - Prin utilizarea interfeței `javax.sql.DataSource`: aceasta este mai nouă și potrivită aplicațiilor web/enterprise. Ea permite ca detaliile legate de sursa de date utilizată la nivelul bazei de date să rămână complet transparente aplicației.

Conexiunea

- Interfața `java.sql.Connection` reprezintă o conexiune (sesiune) cu o bază de date specifică.
- Interogările și comenzile SQL sunt executate, iar rezultatele lor returnate în contextul conexiunii.
- Metode importante:
  - `DatabaseMetaData getMetaData()`: returnează un obiect de tipul `java.sql.DatabaseMetaData` care conține metadate despre baza de date.
  - `close(), isClosed(): boolean`
  - `Statement createStatement()`: creează un obiect de tipul `java.sql.Statement` pentru executarea interogărilor SQL.
  - `PreparedStatement prepareStatement()`: creează un obiect de tipul `java.sql.PreparedStatement` pentru executarea interogărilor SQL.
  - `rollback()`: anulează toate modificările efectuate de la ultima operațiune `commit()` sau `rollback`.
  - `commit()`: salvează modificările efectuate de la ultima operațiune `commit()` sau `rollback`.

Comenzi I

- Clasele care implementează interfața `java.sql.Statement` sunt utilizate pentru a executa comenzi SQL și a returna rezultatele obținute.
- Metode notabile:
  - `ResultSet executeQuery(String sql)`: pentru interogări de tip `SELECT`.
  - `int executeUpdate(String sql)`: pentru comenzile `CREATE`, `DROP`, `INSERT`, `DELETE`.
  - `boolean execute(String sql)`: pentru comenzile SQL care pot returna mai multe rezultate.
  - `int[] executeBatch()`: pentru loturi de comenzi.

Comenzi II

- Exemple:

```
// SELECT
Statement s = conn.createStatement();
ResultSet rs =
    s.executeQuery("select * from books");
// proceseaza rezultatul
rs.close();
s.close();

// stergere
String delString =
    "delete from books where title='Open'";
Statement s = conn.createStatement();
s.executeUpdate(delString);
s.close();
```

java.sql.PreparedStatement

- Spre deosebire de `java.sql.Statement`, interfața `java.sql.PreparedStatement` permite parametrizarea, astfel se poate lucra cu interogări și comenzi ce includ parametri.
- `java.sql.PreparedStatement` este mai rapid față de `java.sql.Statement`.
- Este de asemenea util în prevenția atacurilor de tip injectarea de SQL, deoarece introduce în mod automat caracterele ce previn interpretarea unui șir de caractere ca fiind parte a unei comenzi SQL.
- Exemplu de injectare SQL:

```
statement = "SELECT * FROM users
             WHERE name = '" + userName + "';"
```

java.sql.PreparedStatement II

- Dacă variabilei `userName` i se atribuie:

```
' OR '1'='1
```

  atunci interogarea devine:

```
SELECT * FROM users WHERE name = '' OR '1'='1';
```

- Aceasta va returna întreaga tabelă `users`.
- Însă, poate fi și mai grav de atât. Variabila `userName` poate fi setată astfel:

```
a';DROP TABLE users;
SELECT * FROM userinfo WHERE 't' = 't
```

- În acest caz, interogarea devine:

```
SELECT * FROM users WHERE name = 'a';
DROP TABLE users;
SELECT * FROM userinfo WHERE 't' = 't';
```

java.sql.PreparedStatement III

- În cadrul java.sql.PreparedStatement se pot utiliza mai multe metode set() cu scopul parametrizării valorilor:

```
PreparedStatement statement =
    conn.prepareStatement("INSERT INTO books
                           VALUES (?, ?, ?)");
statement.setString(1, "Andre Agassi");
statement.setString(2, "Open");
statement.setInt(3, 300);
statement.executeUpdate();
statement.close();
```

java.sql.ResultSet

- Această clasă modelează un tabel care reprezintă rezultatele unei interogări de tip `SELECT`.
- Clasa păstrează un cursor care indică rândul curent în cadrul datelor.
- Inițial, cursorul este poziționat înaintea primului rând. Cursorul poate fi mutat pe rândul următor prin apelarea metodei `next()`.
- Când nu mai sunt alte rânduri, acească metodă returnează `false`.
- Un obiect de tipul `java.sql.ResultSet` nu poate fi actualizat (nu se poate modifica baza de date prin intermediul său).

java.sql.ResultSet II

- Metode importante:
  - `boolean next()`, `previous()`, `first()`, `last()`: acestea mută cursorul.
  - `boolean absolute(int row)`, `relative(int row)`: acestea mută cursorul pe rândul specificat (absolut sau relativ).
  - `int getInt(int columnIndex)`, `int getInt(String columnName)`: returnează datele din coloana specificată (prin index sau nume); metode simiare există pentru celelalte tipuri de date (ex. `getLong()`, `getDouble()`, `getDate()`, `getTime()`, etc.).

Tranzacții I

- O tranzacție reprezintă o unitate de lucru (una sau mai multe instrucțiuni) care trebuie executate ca o singură operație. Ori toate instrucțiunile sunt executate, ori nici una nu e (ex. transferul de bani între conturi).
- În cazul în care o singură instrucțiune din cadrul tranzacției nu poate fi îndeplinită, atunci întreg efectul tranzacției trebuie anulat, ca și cum nu ar fi avut loc.
- Comportamentul implicit al tranzacțiilor poate fi modificat prin intermediul obiectelor de tipul `java.sql.Connection`, folosind metoda `setAutoCommit()`.

Tranzacții II

- În mod implicit, fiecare interogare/comandă SQL e tratată ca o tranzacție individuală, fiind executat `commit()` imediat după ce a fost rulată.
- Pentru a grupa două sau mai multe comenzi într-o singură tranzacție, modul `auto-commit` trebuie dezactivat. În acest caz, nici o comandă SQL nu este salvată până la apelarea explicită a metodei `commit()` pe obiectul `java.sql.Connection` corespunzător.
- Metoda `rollback()` a obiectului `java.sql.Connection` corespunzător anulează tranzacția și restaurează entitățile bazei de date la valorile lor de dinaintea operației anulate.

Exemple I

Pentru a putea rula acest exemplu, trebuie ca proiectul IntelliJ să aibă acces la driverul sqlite, care permite conectarea la o bază de date SQLite folosind JDBC.

Exemplu

`lecture.examples.lecture6.jdbc.JDBC.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbcluj-ro/MAP

Elemente noi în Java 8

- Java 8 a fost o versiune revoluționară, care a adus modificări importante limbajului Java precum și a mașinii virtuale și a bibliotecilor implicite.
- Cele mai importante elemente noi introduse în Java 8:
  - Expresii lambda (eng. *lambda expressions*).
  - Fluxuri și pipe-uri (eng. *streams and pipelines*).
  - API pentru lucrul cu date și timp.
  - Adnotări pentru tipuri de date (eng. *type annotations*).
  - Metode implicite în interfețe (eng. *default methods*).
  - Operații paralele (eng. *parallel operations*).

Expresii lambda

- Primul pas al platformei Java în programarea funcțională.
- O expresie lambda este o funcție care poate fi creată fără ca ea să aparțină unei clase și fără a fi conectată la un identificator (funcțiile lambda sunt anonime).
- Funcția poate fi transmisă ca un obiect și executată la cerere, ceea ce ajută la reducerea cantității de cod sursă necesar.

(<lambda_parameters>) -> lambda_body

Interfețe funcționale

- O interfață funcțională este o interfață care are o singură metodă (abstractă).
- Opțional, ea poate fi anotată cu `@FunctionalInterface`. Anotarea există pentru a se evita adăugarea accidentală de metode adiționale unei interfețe care se dorește a fi funcțională.
- Beneficiul este că putem utiliza expresii lambda pentru a le instanția. Interfețele funcționale pot fi implementate folosind mecanismul expresiilor lambda.

```
@FunctionalInterface
public interface InterfaceName
{
    public Type function(<params>);
}
```

Interfețe funcționale deja existente

- Limbajul Java include un număr de interfețe funcționale introduse pentru a sprijini cazurile de utilizare comune, toate fiind disponibile în pachetul `java.util.function`.
- Exemple:
  - `java.util.function.Function`
  - `java.util.function.Predicate`
  - `java.util.function.UnaryOperator`
  - `java.util.function.BinaryOperator`
  - `java.util.function.Supplier`
  - `java.util.function.Consumer`

java.util.function.Function

- Interfața java.util.function.Function reprezintă o funcție care acceptă un singur parametru și care returnează o singură valoare.

```
public interface Function<T,R> {
    public <R> apply(T parameter);
}
```

- Metoda apply() trebuie să fie implementată.
- Interfața include mai multe metode care sunt implicite sau statice, așadar nu este nevoie de implementarea lor.

java.util.function.Predicate

- Interfața `java.util.function.Predicate` reprezintă o funcție care acceptă un singur parametru și care returnează o valoare de tip `boolean`.

```
public interface Predicate<T> {
    boolean test(T t);
}
```

- Metoda *test()* trebuie să fie implementată.
- Interfața include mai multe metode care sunt implicite sau statice, așadar nu este nevoie de implementarea lor.

java.util.function.Predicate II

- Metode importante:
  - `and(Predicate<? super T> other))`: returnează un predicat compus care reprezintă o operație `AND` logică cu scurt-circuit între acest predicat și cel dat.
  - `or(Predicate<? super T> other)`: returnează un predicat compus care reprezintă o operație `OR` logică cu scurt-circuit între acest predicat și cel dat.
  - `negate()`: returnează un predicat compus care reprezintă operația `NOT` logică.

java.util.function.UnaryOperator și java.util.function.BinaryOperator

- Interfața `java.util.function.UnaryOperator` reprezintă o operație care acceptă un singur parametru și care returnează o valoare de același tip.
- Poate fi utilizată pentru a reprezenta o operație care acceptă un obiect ca parametru, modifică acel obiect și îl returnează.
- Interfața `java.util.function.BinaryOperator` reprezintă o operație care acceptă doi parametri și care returnează o singură valoare. Atât parametrii, cât și valoarea returnată trebuie să fie de același tip.

java.util.function.Supplier

- Interfața java.util.function.Supplier reprezintă o funcție care furnizează niște valori.
- Poate fi privită ca o interfață a șablonului de proiectare Factory (https://refactoring.guru/design-patterns/factory-method)

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

java.util.function.Consumer

- Interfața java.util.function.Consumer reprezintă o funcție care consumă o valoare, fără a returna nimic.
- Această implementare poate fi utilizată pentru afișarea unei valori, scrierea în fișier etc.

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

Referințe la metode

- Referințele la metode reprezintă un tip special de expresii lambda.
- Tipuri de referințe la metode:
  - Metode statice.
  - Metode de instanță ale unor obiecte particulare.
  - Metode de instanță ale unui obiect arbitrar de un anumit tip.
  - Constructor.

Exemplu

`lecture.examples.lecture6.Examples.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbcluj-ro/MAP

Metode adiționale pentru colecții

- `forEach` - utilizat pentru a executa aceeași operație pe fiecare element al unei colecții.

```
String[] stringArray = { "Barbara", "James", "Mary"};
List<String> names = Arrays.asList(stringArray);
names.forEach(System.out::println);
```

- `removeIf` - șterge toate elementele colecției care satisfac un predicat filtru dat ca și parametru al metodei.

```
names.removeIf(x -> x.endsWith("a"));
```