Medii de proiectare și programare

2023-2024
Curs 2

Conținut curs 2

- Gradle (cont. curs 1)
- Jurnalizare
- SQLite, SQLiteStudio, MySQL
- Accesul la baze de date relaționale
    - Java: JDBC
    - C#: ADO.NET - curs 3
- Configurarea (Java properties, C# app.config - curs 3)
- Ierarhia repository - curs 3

Gradle Build Tool

- Proiecte multiple:
  - Fiecare proiect (subproiect) are aceeași structură - corespunzătoare proiectelor Gradle Java.
  - Fiecare proiect (subproiect) va conține fișierul build.gradle propriu, cu configurările specifice.
  - Proiectul rădăcină (root) conține obligatoriu și fișierul settings.gradle:

```
include 'A'
include 'B'
```

Gradle Build Tool

- Dependențe între (sub)proiecte: Subproiectul B depinde de subproiectul A:
  - build.gradle corespunzător subproiectului B:

```
dependencies {
    implementation project(':A')
}
```

Gradle Build Tool

- Proiectul A: build.gradle

```
plugins{
    id 'java'
}
repositories {
    mavenCentral()
}
dependencies {
   implementation 'com.google.guava:guava:20.0'
    testImplementation 'junit:junit:4.11'
}
```

Gradle Build Tool

- Proiectul B: build.gradle

```
plugins{
    id ‘java'
    id ‘application'
}
repositories {
    mavenCentral()
}
application{
    mainClass=‘StartApp’
}
dependencies {
    testImplementation ‘junit:junit:4.11'
    implementation project(‘:A')
}
```

Gradle Build Tool

- Proiectul Root: build.gradle

```
allprojects {
        plugins{
            id ‘java’
        }
        repositories {
            mavenCentral()
        }
        dependencies {
            testImplementation ‘junit:junit:4.11’
        }
}
```

Gradle Build Tool

- Proiectul A: build.gradle modificat

```
dependencies {
    implementation 'com.google.guava:guava:20.0'
    }
```

- Proiectul B: build.gradle modificat

```
plugins{
    id ‘application’
}
application{
    mainClass=‘StartApp’
}
dependencies {
      implementation project(‘:A')
}
```

Gradle Build Tool

- Proiectul Root: build.gradle

```
subprojects {
    //Configurări comune tuturor subproiectelor
}
project(':A') {
    //Configurări specifice proiectului A
}

project(':B') {
    //Configurări specifice proiectului B
}
```

Instrumente pentru jurnalizare

- Un ***instrument pentru jurnalizare*** permite programatorilor să înregistreze diferite tipuri de mesaje din codul sursă cu diverse scopuri: depanare, analiza ulterioară, etc.

- Majoritatea instrumentelor definesc diferite nivele pentru mesaje: *debug*, *warning*, *error*, *information*, *sever*, etc.

- Configurarea instrumentelor se face folosind fișiere text de configurare și pot fi oprite sau pornite la rulare.

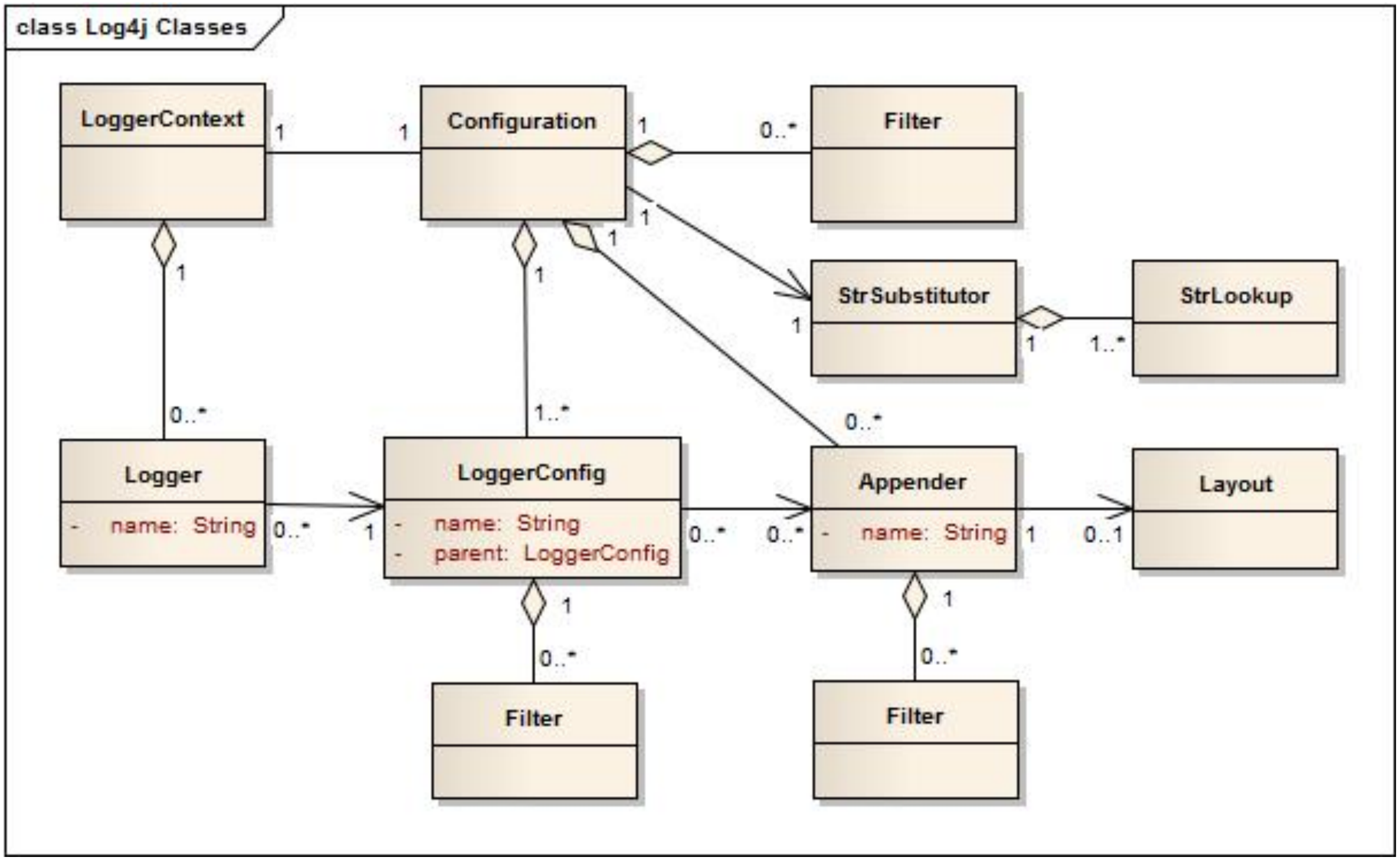
- Apache Log4j, Logging SDK, slf4j, etc.

- Proiect open source dezvoltat de Apache Foundation.

  http://logging.apache.org/log4j/2.0/

- Log4j 2 are 3 componente principale:
  - *loggers*,
  - *appenders* (pentru stocare),
  - *layouts* (pentru formatare).

Arhitectura



- Aplicațiile care folosesc Log4j 2 cer o referință către un obiect de tip *Logger* cu un anumit nume de la *LogManager*.
- LogManager va localiza obiectul LoggerContext corespunzător numelui și va obține referința către obiectul Logger de la el.
- Dacă obiectul de tip Logger corespunzător încă nu a fost creat, se va crea unul nou și va fi asociat cu un obiect de tip LoggerConfig care fie:
  - are același nume ca și Logger,
  - are același nume ca și pachetul părinte,
  - este rădăcina LoggerConfig.
- Obiectele de tip LoggerConfig sunt create folosind declarațiile din fișierul de configurare.
- Fiecărui LoggerConfig îi sunt asociate unul sau mai multe obiecte de tip Appender.

- Fișierul de configurare (XML, JSON, proprietăți Java, yaml)
- Dacă nu este configurat, log4j 2 afișează doar mesajele de tip *error* la consolă

Exemplu fișier de configurare în format XML: *log4j2.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="TRACE">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="TRACE">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

- Log level - fiecare mesaj are asociat un anumit nivel
  - TRACE, DEBUG, INFO, WARN, ERROR și FATAL

Event Level	LoggerConfig Level						
	TRACE	DEBUG	INFO	WARN	ERROR	FATAL	OFF
ALL	YES	YES	YES	YES	YES	YES	NO
TRACE	YES	NO	NO	NO	NO	NO	NO
DEBUG	YES	YES	NO	NO	NO	NO	NO
INFO	YES	YES	YES	NO	NO	NO	NO
WARN	YES	YES	YES	YES	NO	NO	NO
ERROR	YES	YES	YES	YES	YES	NO	NO
FATAL	YES	YES	YES	YES	YES	YES	NO
OFF	NO	NO	NO	NO	NO	NO	NO

- Numele asociat unui logger: structura ierarhică asemănătoare structurii pachetelor Java.

```
public class LogTest {
//a
private static final Logger logger = LogManager.getLogger(LogTest.class);
 //b
  private static final Logger logger =
LogManager.getLogger(LogTest.class.getName());
//c
private static final Logger logger = LogManager.getLogger();
}
```

- Clasa Logger conține metode ce permit urmărirea fluxului execuției unei aplicații.
  - entry(…) - 0 ..4 parametrii
  - traceEntry(String, …) - String și o lista variabilă de parametri
  - exit(…), traceExit(String, …)
  - throwing (…) - când se aruncă o excepție
  - catching(…) când se prinde o excepție
  - trace(…)
  - error(…)
  - log(…)
  - etc.
- Exemplu

- Salvarea mesajelor: fișier, consolă, baze de date, etc.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="TRACE">
  <Appenders>
    <File name="FisierLog" fileName="logs/app.log">
      <PatternLayout pattern="%d{DATE} [%t] %class{36} %L %M - %msg%xEx%n"/>
    </File>
  </Appenders>
  <Loggers>
    <Root level="TRACE">
      <AppenderRef ref="FisierLog"/>
    </Root>
  </Loggers>
</Configuration>
```

SGBD

- Sqlite
    - SQLiteStudio
- MySQL/MariaDB

- https://www.sqlite.org/
- Bază de date relațională
- Nu necesită configurări adiționale
- Nu necesită pornirea unui proces separat
- Toate informațiile sunt păstrate într-un singur fișier
- Formatul fișierului este independent de platformă
- Open source, gratuit.

```
sqllite_dir> sqlite3
```

- https://sqlitestudio.pl/
- Sistem de gestiune a unei baze de date Sqlite
- Interfață grafică ușor de folosit
- Independent de platformă
- Gratuit
- Open source

- https://www.mysql.com/ https://mariadb.com/

- Sistem de gestiune a bazelor de date relaționale

- Rapid, scalabil, ușor de folosit

- Sistem de tip client-server/ embedded

- Gratuit

- Open source (MariaDb)

JDBC

- Java Database Connectivity (JDBC) API conține o mulțime de clase ce asigură accesul la date.
- Se pot accesa orice surse de date: baze de date relaționale, foi de calcul (*spreadsheets*) sau fișiere.
- JDBC oferă și o serie de interfețe ce pot fi folosite pentru construirea instrumentelor specializate.
- Pachete:
  - **`java.sql`** conține clase și interfețe pentru accesarea și procesarea datelor stocate într-o sursă de date (de obicei bază de date relațională).
  - **`javax.sql`** - adaugă noi funcționalități pentru partea de server.

Stabilirea unei conexiuni

- Conectarea se poate face în două moduri:
  - Clasa **`DriverManager`**: Conexiunea se creează folosind un URL specific.
    - Necesita încărcarea unui driver specific bazei de date (JDBC<4).
    - Incepând cu JDBC 4.0 nu mai este necesară încărcarea driverului.
  - Interfața **`DataSource`**: Este recomandată folosirea interfeței pentru aplicații complexe, deoarece permite configurarea sursei de date într-un mod transparent.
- Stabilirea unei conexiuni se realizează astfel:
  - Încărcarea driverului (versiuni JDBC <4.0)

```
Class.forName(<DriverClassName>);
```

    - **`Class.forName`** creează automat o instanță a driverului și o înregistrează la **`DriverManager.`**
    - Nu este necesară crearea unei instanțe a clasei.
  - Crearea conexiunii.

Crearea unei conexiuni

- Folosind clasa **`DriverManager`** :
  - Colaborează cu interfața Driver pentru gestiunea driverelor disponibile unui client JDBC.
  - Când clientul cere o conexiune și furnizează un URL, clasa DriverManager este responsabilă cu găsirea driverului care recunoaște URL și cu folosirea lui pentru a se conecta la sursa de date.
  - Sintaxa URL-ului corespunzător unei conexiuni este:

```
jdbc:subprotocol:<numeBazaDate>[listaProprietati]
```

```
Connection conn = DriverManager.getConnection("jdbc:sqlite:users.db");

String url = "jdbc:mysql:Test";

String url = "jdbc:mariadb://localhost:3306/Test"

Connection conn = DriverManager.getConnection(url, <user>, <passwd>);
```

Crearea unei conexiuni

- Folosind interfața **`DataSource`**:

```
InitialContext ic = new InitialContext()
 //a)
 DataSource ds = ic.lookup("java:comp/env/jdbc/myDB");
 Connection con = ds.getConnection();
//b)
 DataSource ds =
    (DataSource)org.apache.derby.jdbc.ClientDataSource()
 ds.setPort(1527);
 ds.setHost("localhost");
 ds.setUser("APP")
 ds.setPassword("APP");

Connection con = ds.getConnection();
```

Clasa Connection

- Reprezintă o sesiune cu o bază de date specifică.
- Orice instrucțiune SQL este executată și rezultatele sunt transmise folosind contextul unei conexiuni.
- Metode:
  - `close(), isClosed():boolean`
  - `createStatement(…):Statement  //overloaded`
  - `prepareCall(…):CallableStatement //overloaded`
  - `prepareStatement(…):PreparedStatement //overloaded`
  - `rollback()`
  - `setAutoCommit(boolean)   //tranzactii`
  - `getAutoCommit():boolean`
  - `commit()`

Clasa Statement

- Se folosește pentru executarea unei instrucțiuni SQL și pentru transmiterea rezultatului.
- Metode:
  - `execute(sql:String, …):boolean  //pentru orice instructiune SQL`
    - `getResultSet():ResultSet`
    - `getUpdateCount():int`
  - `executeQuery(sql:String, …):ResultSet  //pentru SELECT`
  - `executeUpdate(sql:String, …):int  //INSERT,UPDATE,DELETE`
  - `cancel()`
  - `close()`

Exemplu Statement – Structura bazei de date

	Field Name	Data Type
🔑	ID	AutoNumber
	title	Text
	authors	Text
	isbn	Text
	year	Number
▶		

Statement exemplu

```
//Conectarea la o baza de date SQLite
//Class.forName("org.sqlite.JDBC");
Connection conn=DriverManager.getConnection("jdbc:sqlite:/Users/teste/database/
    books.db");
//select
try(Statement stmt=conn.createStatement()){
  try(ResultSet rs=stmt.executeQuery("select * from books")){
    ...
  }
}catch(SQLException ex){
  System.err.println(ex.getSQLState());
  System.err.println(ex.getErrorCode())
  System.err.println(ex.getMessage());

}
//update
String upString="update books set isbn='tj234' where isbn='tj237'"
try(Statement stmt=conn.createStatement()){
  stmt.executeUpdate(upString);
}catch(SQLException ex){…}
```

Statement exemplu

```
//insert
String insert="insert into books (title, authors, isbn, year) values
   ('Nuvele', 'Mihai Eminescu', '4567567', 2008)";
 try(Statement  stmt=conn.createStatement()) {
       stmt.executeUpdate(insert);
} catch (SQLException e) {
     System.out.println("Insert error "+e);
}
```

```
//delete
String delString="delete from books where isbn='tj234'"
try(Statement stmt=conn.createStatement()){
     stmt.executeUpdate(delString);
}catch(SQLException ex){
  //…
}
```

ResultSet

- Conține o tabelă ce reprezintă rezultatul unei instrucțiuni SELECT.
- Un obiect de tip ResultSet conține un cursor care indică linia curentă din tabelă.
- La început cursorul este poziționat înaintea primei linii din tabelă.
- Metoda **`next`** mută cursorul pe următoarea linie din tabelă. Rezultatul returnat este **`false`**, dacă nu mai există linii neparcurse în obiectul ResultSet.
- Metoda **`next`** se folosește pentru a parcurge toate liniile din tabelă.
- Se pot configura anumite proprietăți (daca tabela poate fi modificata, modul de parcurgere, etc.) .
- Configurarea se face în momentul apelului metodei de tip **`createStatement(…)`**:

```
 Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT name, address FROM users");
    // rs poate fi iterat, nu va fi notificat de modificari facute
    //de alti utilizatori ai BD, si poate fi actualizat.
```

ResultSet

- Metode:
  - **absolute(row:int)**
  - **relative(n:int)**
  - **afterLast(), beforeFirst() , first(), last(), next():boolean**
  - **getRow():int**
  - **getInt(columnIndex|columnLabel):int**
  - **getFloat(…), getString(…), getObject(…), etc**
  - **updateInt(columnIndex|columnLabel, newValue)**
  - **updateFloat(…), updateString(…), etc**
  - **updateRow()**
  - **refreshRow()**
  - **rowDeleted(), rowInserted(), rowUpdated()**

ResultSet

- Implicit un obiect de tip **`ResultSet`** este unidirecțional, cu parcurgerea înainte și nu poate fi modificat (actualizat).

```
try(Statement stmt=conn.createStatement()){
    try(ResultSet rs=stmt.executeQuery("select * from books")){
        while(rs.next()){
            System.out.println("Book "+rs.getString("title")
              +' '+rs.getString("author")+' '+rs.getInt("year"));
        }
    }
}catch(SQLException ex){
// …
}
```

Clasa PreparedStatement

- Unui obiect de tip **`PreparedStatement`** i se transmite instrucțiunea SQL în momentul creării.
- Instrucțiunea SQL este transmisă sistemului de gestiune a bazei de date (SGBD), unde este compilată.
- Când se execută instrucțiunea asociată unui **`PreparedStatement`**, SGBD execută direct instrucțiunea SQL fără a o reverifica în prealabil.
- Este mai eficientă decât Statement.
- Poate să aibă parametrii. Aceștia sunt marcați folosind '?'.

```
PreparedStatement preStmt = con.prepareStatement(
    "select * from books WHERE year=?");
```

- Valoarea unui parametru este transmisă folosind metodele de tip **`setXYZ`**, unde **`XYZ`** reprezinta tipul parametrului.
- Pozițiile parametrilor încep de la 1.

```
preStmt.setInt(1, 2008);
ResultSet rs=preStmt.executeQuery();
```

Tranzacții

- Implicit, fiecare instrucțiune SQL este tratată ca și o tranzacție și este înregistrată/operată imediat după execuție.
- Comportamentul implicit poate fi modificat folosind metoda `setAutoCommit(false)` din clasa `Connection`.
- Metode:
  - `commit`
  - `rollback`
  - `setSavePoint`

Tranzacții - exemplu

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, “Black");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
    "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME
    LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, “Black");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

Proceduri stocate

- O procedură stocată este un grup de instrucțiuni SQL care formează o unitate logica și îndeplinesc o anumită sarcină.
- Ele sunt folosite pentru a îngloba o serie de operaţiuni sau interogări ce trebuie executate pe un server de baze de date.
- De exemplu, operaţiunile de pe o bază de date angajat (angajarea, concedierea, promovarea, cautarea) ar putea fi codificate ca proceduri stocate executate în funcţie de codul cerere.
- Procedurile stocate pot fi compilate şi executate cu diferiți parametrii și pot avea orice combinație de intrare, ieșire sau intrare/ieșire.

CallableStatement

- Este folosită pentru executarea procedurilor stocate.
- Tehnologia JDBC API furnizează o sintaxă de apelare a procedurilor stocate independentă de SGBD folosit.
- Sintaxa folosită are două variante:
  - conține un parametru de tip rezultat
  - nu conține un parametru de tip rezultat.
- Dacă se folosește prima varianta, parametrul de tip rezultat trebuie să fie înregistrat ca și parametru de tip OUT. Ceilalți parametrii pot fi folosiți pentru intrare, ieșire sau ambele.
- Parametrii sunt referiți secvențial, folosind numere, primul parametru fiind pe poziția 1.

```
{?= call <procedure-name>[(<arg1>,<arg2>, ...)]}
{call <procedure-name>[(<arg1>,<arg2>, ...)]}
```

```
CallableStatement cs = con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

Properties

- Clasa **`Properties (pachetul java.util)`** se folosește pentru a păstra perechi cheie-valoare. Perechile pot fi citite sau salvate dintr-un/într-un flux de date (ex. fișier). Cheia și valoarea sunt de tip String, cheia fiind unică.

```
//exemplu.properties
tasksFile=tasks.txt
inputDir=input
outputDir=output

//Citirea fișierului cu proprietăți
Properties props=new Properties();
try {
    props.load(new FileInputStream(“exemplu.properties"));
} catch (IOException e) {
    System.out.println("Eroare: "+e);
}
```

Properties

- Metode:
  - `getProperty(cheie:String):String`
  - `setProperty(c:String, v:String):Object`
  - `list(PrintWriter)`
  - `load(Reader)`
  - `store(w:Writer, comentarii:String)`

```
Properties props=new Properties();
try {
    props.load(new FileInputStream(“exemplu.properties"));
} catch (IOException e) {
    System.out.println("Eroare: "+e);
}
String tasksFile=props.getProperty(“tasksFile”);
if (tasksFile==null) //proprietatea nu a fost gasita in fisier
   System.out.println(“fisier incorect”);
...
```

System +Properties

- Metode din clasa System:
  - **setProperties(Properties)**
  - **setProperty(c:String, v:String):String**
  - **getProperty(String):String**
  - **...**

```
Properties serverProps=new Properties(System.getProperties());
try {
     serverProps.load(new FileReader("exemplu.properties"));
     System.setProperties(serverProps);
     System.getProperties().list(System.out);
} catch (IOException e) {
      System.out.println("Eroare "+e);
}
String tasksFile=System.getProperty("tasksFile");
```

Exemplu configurare BD

```
//Fisierul bd.properties sau bd.config
jdbc.url=jdbc:mysql://localhost/mpp
jdbc.user=test
jdbc.pass=test


//cod
Connection getNewConnection(){
        String url=System.getProperty("jdbc.url");
        String user=System.getProperty("jdbc.user");
        String pass=System.getProperty("jdbc.pass");
        Connection con=null;
        try {
             con= DriverManager.getConnection(url,user,pass);
        } catch (SQLException e) {
            System.out.println("Eroare stabilire conexiune "+e);
        }
        return con;
}
```

Dependențe driver JDBC - Gradle

```
//Fisierul build.gradle
dependencies {
  testImplementation group: 'org.junit.jupiter', name: 'junit-jupiter-api', version:
   '5.8.2'
  testImplementation group: 'org.junit.jupiter', name: 'junit-jupiter-engine',
   version:'5.8.2'
   //jurnalizare
    implementation group: 'org.apache.logging.log4j', name: 'log4j-core', version:
    '2.18.0'
    implementation group: 'org.apache.logging.log4j', name: 'log4j-api', version:
    '2.18.0'
   //drivere conectare la baza de date
    runtimeOnly group: 'org.xerial', name: 'sqlite-jdbc', version: '3.36.0.3'

    runtimeOnly 'org.mariadb.jdbc:mariadb-java-client:2.1.2'
    runtimeOnly 'mysql:mysql-connector-java:5.1.20'

}
//Maven repository
//https://mvnrepository.com/
```