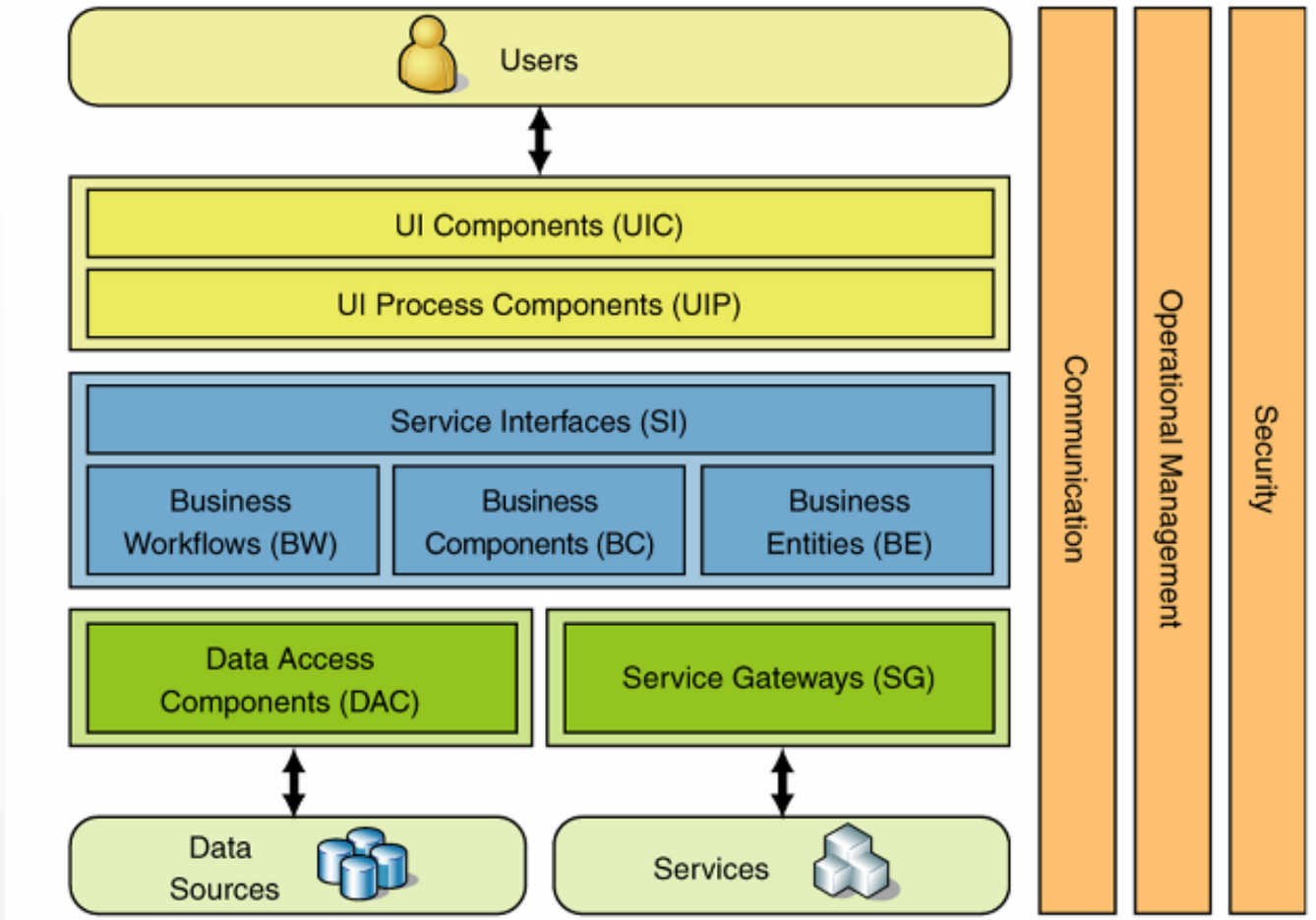
Perzisztencia alapok

Multitier architecture, Abstract DAO Factory, JDBC

Simon Károly
simon.karoly@codespring.ro

Multitier Architecture

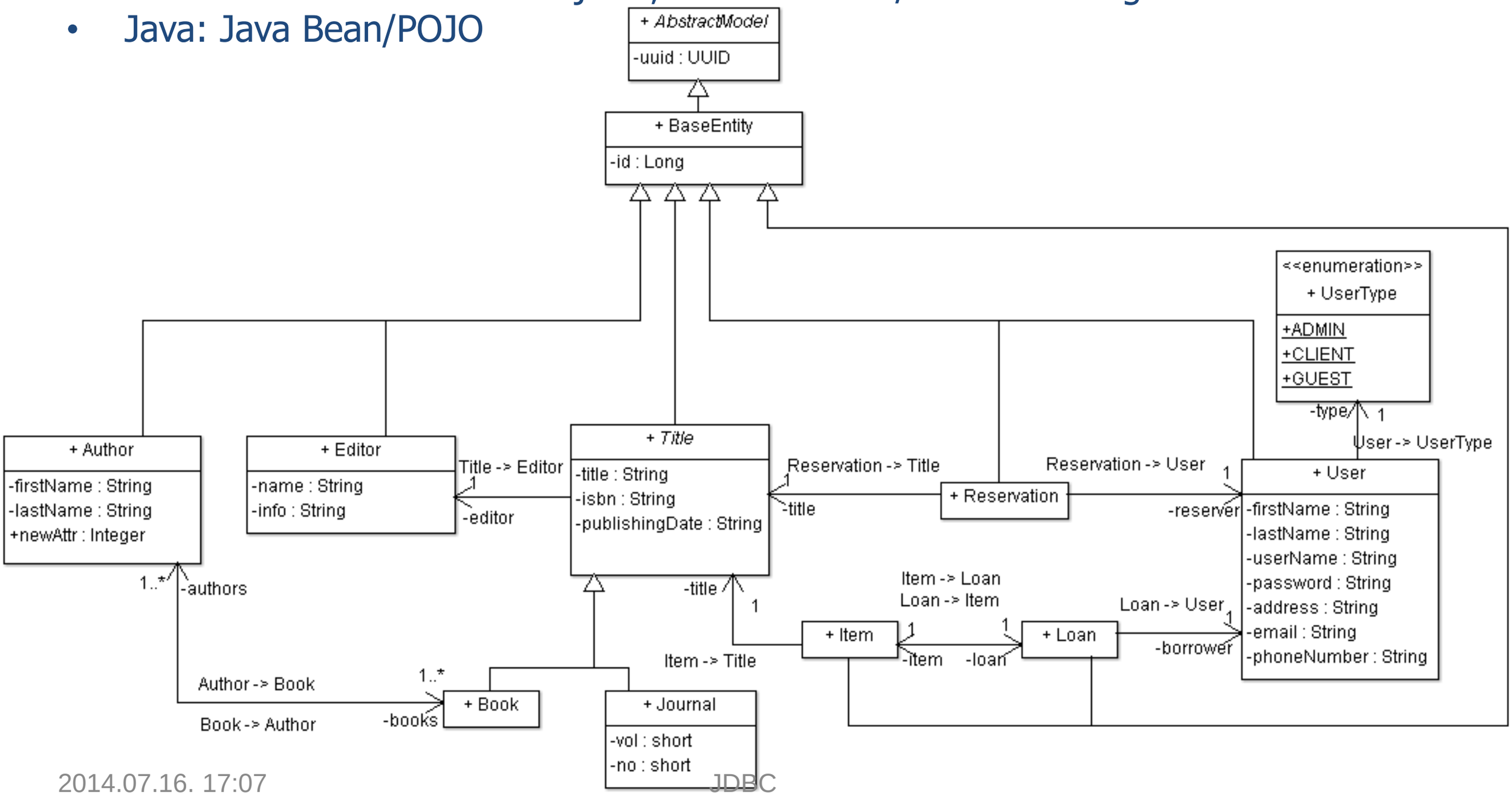
- Többrétegű architektúra (multitier/multilayer architecture).
- Általában három réteg: Presentation Layer, Application (Business) Layer, Data Acces Layer



Forrás: Hanselman's Computer Zen

Környezeti elemzés, modell

- Domain Analysis: a rendszeren belüli központi entitások beazonosítása
  domain classes → model objects, beans → core/model csomag
- Java: Java Bean/POJO



Abstract Model

```
package edu.codespring.bibliospring.backend.model;

import java.util.UUID;

public abstract class AbstractModel {

    private UUID uuid;

    @Override
    public int hashCode () {
      …
    }
    @Override
    public boolean equals (final Object obj) {
      …
    }
    public UUID getUuid () {
        if (uuid == null) {
            uuid = UUID.randomUUID ();
        }
        return uuid;
    }
```

Base Entity

```
package edu.codespring.bibliospring.backend.model;

import java.io.Serializable;

public class BaseEntity extends AbstractModel implements Serializable {

    private static final long serialVersionUID = 1L;
    private Long              id;

    public BaseEntity () {
        this (null);
    }

    public BaseEntity (final Long id) {
        super ();
        this.id = id;
    }

    public Long getId () {
        return id;
    }

    public void setId (final Long id) {
        this.id = id;
    }

}
```

Title

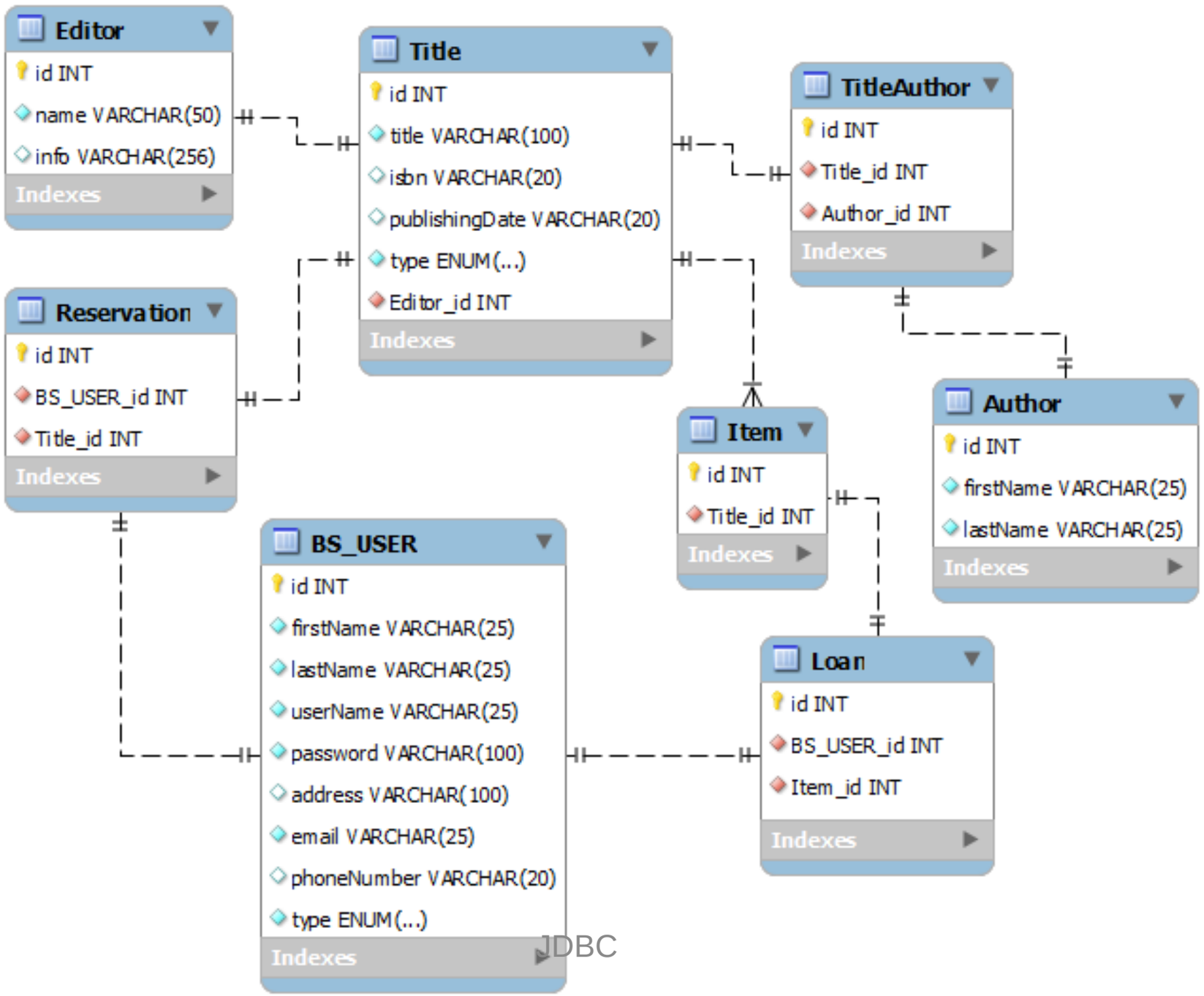
```
package edu.codespring.bibliospring.backend.model;

…               //imports

public abstract class Title extends BaseEntity implements Comparable<Title> {

    private static final long serialVersionUID = 1L;
    private String            title;
    private String            isbn;
    private Editor            editor;
    private String            publishingDate;
    private List<Item>        items;

    …           //constructors
    …           //getters and setters

    //return empty list instead null when there are no items
    public List<Item> getItems () {
        if (items == null) {
            items = Collections.emptyList ();
        }
        return items;
    }
    //initialize the list when the first element is inserted
    public void addItem (final Item i) {
        if (items == null) {
            items = new ArrayList<Item> ();
        }
        items.add (i);
    }

    …           //compareTo, toString etc.
}
```

Book

```
package edu.codespring.bibliospring.backend.model;

…           //imports

public class Book extends Title {

    private static final long serialVersionUID = 1L;
    private List<Author>      authors;

    …       //constructors

    …       //getters and setters

    …       //add/remove author

    …       //toString etc.


}
```

Database

- Relációs adatbázis esetén a megfelelő táblák:



Database

- Példa: Title

```
CREATE  TABLE IF NOT EXISTS `bibliospring`.`Title` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT ,
  `title` VARCHAR(100) NOT NULL ,
  `isbn` VARCHAR(20) NULL ,
  `publishingDate` VARCHAR(20) NULL ,
  `type` ENUM('BOOK', 'JOURNAL') NOT NULL ,
  `Editor_id` INT NOT NULL ,
  PRIMARY KEY (`id`) ,
  INDEX `fk_Title_Editor1_idx` (`Editor_id` ASC) ,
  UNIQUE INDEX `isbn_UNIQUE` (`isbn` ASC) ,
  CONSTRAINT `fk_Title_Editor1`
    FOREIGN KEY (`Editor_id` )
    REFERENCES `bibliospring`.`Editor` (`id` )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

Data Access Layer

- DAO interfaces:
- Példa: BookDAO.java

```
package edu.codespring.bibliospring.backend.repository;

import java.util.List;

import edu.codespring.bibliospring.backend.model.Book;

public interface BookDAO {

    List<Book> getAllBooks () throws RepositoryException;

    Book getBookById (Long id) throws RepositoryException;

    List<Book> getBooksByFilter (String pattern) throws RepositoryException;

    void insertBook (Book book) throws RepositoryException;

    void updateBook (Book book) throws RepositoryException;

    void deleteBook (Book book) throws RepositoryException;

}
```

Data Access Layer

```
package edu.codespring.bibliospring.backend.repository;

public class RepositoryException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    public RepositoryException () {
        super ();
    }

    public RepositoryException (final String message) {
        super (message);
    }

    public RepositoryException (final String message, final Throwable cause) {
        super (message, cause);

    }

}
```

Data Access Layer

- Abstract DAO Factory:
- Példa: DAOFactory.java

```
package edu.codespring.bibliospring.backend.repository;

import edu.codespring.bibliospring.backend.repository.jdbc.JdbcDAOFactory;

public abstract class DAOFactory {

    public static DAOFactory getInstance () {
        return new JdbcDAOFactory ();
    }

    public abstract UserDAO getUserDAO ();

    public abstract AuthorDAO getAuthorDAO ();

    public abstract EditorDAO getEditorDAO ();

    public abstract BookDAO getBookDAO ();

    public abstract ItemDAO getItemDAO ();

    public abstract ReservationDAO getReservationDAO ();

    public abstract LoanDAO getLoanDAO ();

}
```

Data Access Layer

- JDBC DAO Factory:
- Példa: JdbcDAOFactory.java

```
package edu.codespring.bibliospring.backend.repository.jdbc;

…     //imports

public class JdbcDAOFactory extends DAOFactory {

    @Override
    public UserDAO getUserDAO () {
        return new JdbcUserDAO ();
    }

    @Override
    public BookDAO getBookDAO () {
        return new JdbcBookDAO ();
    }

    … //getter methods for DAO instances

}
```

Data Access Layer

- DAO implementations:
- Példa: JdbcBookDAO.java

```
package edu.codespring.bibliospring.backend.repository.jdbc;
…    //imports

public class JdbcBookDAO implements BookDAO {

    private final ConnectionManager cm;
    private final DAOFactory        df;

    public JdbcBookDAO () {
        cm = ConnectionManager.getInstance ();
        df = DAOFactory.getInstance ();
    }

    @Override
    public List<Book> getAllBooks () throws RepositoryException {
        final List<Book> bookList = new ArrayList<Book> ();
        Connection con = null;
        try {
            con = cm.getConnection ();
            …         //db operations
        } catch (final SQLException e) {
             …        //log
             throw new RepositoryException ("Book selection failed");
        } finally {
            if (con != null) {
                cm.returnConnection (con);
            }
        }
        return bookList;
    }
```

Data Access Layer

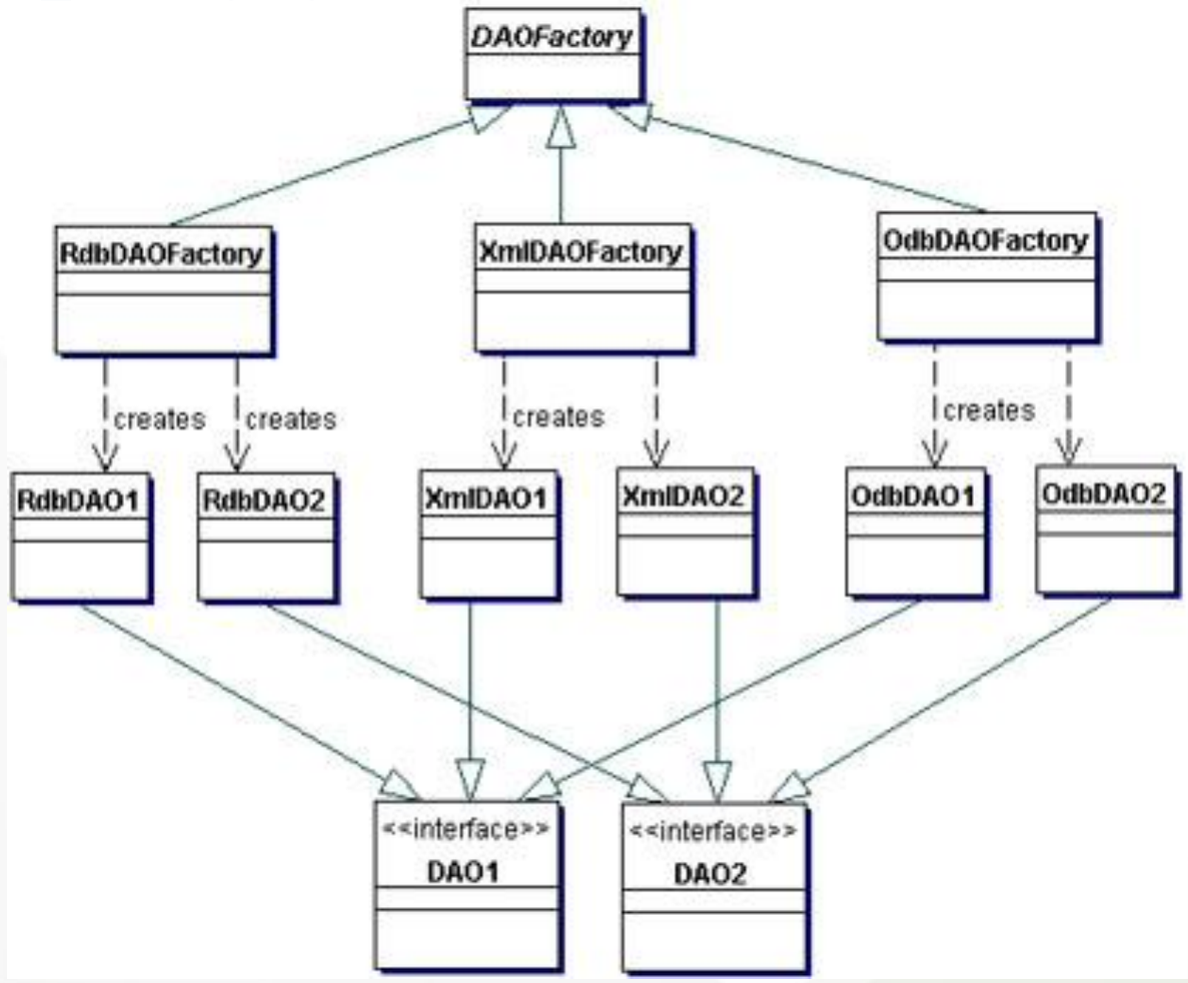
```
package edu.codespring.bibliospring.backend.repository.jdbc;
…       //imports

public final class ConnectionManager {

    …   //connection properties, pool size, logger
    private final List<Connection>   pool;
    private static ConnectionManager instance;

    private ConnectionManager () {
        pool = new LinkedList<Connection> ();
        initializePool ();
    }

    public synchronized static ConnectionManager getInstance () {
        if (instance == null) {
            instance = new ConnectionManager ();
        }
        return instance;
    }

    public synchronized Connection getConnection () throws RepositoryException {
        Connection con = null;
        if (pool.size () > 0) {
            con = pool.get (0);
            pool.remove (0);
        }
        if (con == null) {
            throw new RepositoryException ("No connections in pool");
        }
        return con;
    }

    public synchronized void returnConnection (final Connection con) {
        if (pool.size () < SIZE) {
            pool.add (con);
        }
    }

    private void initializePool () {
        …               //create connections, initialize pool
    }

}
```

Alkalmazás

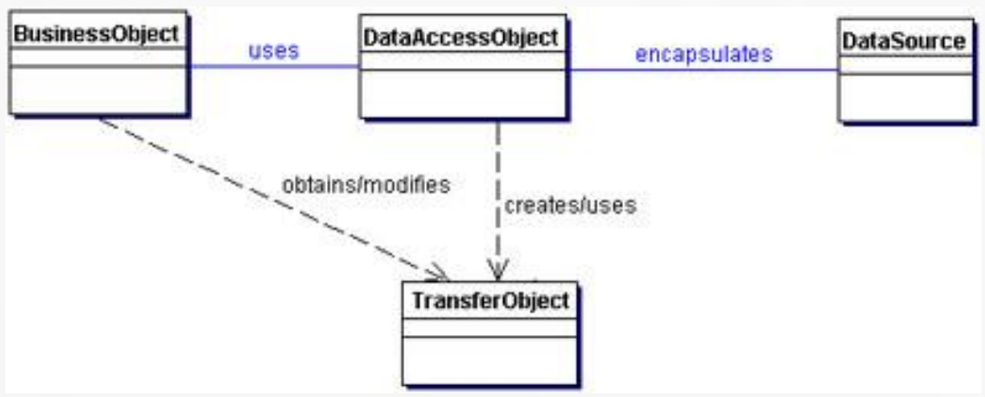
- …

```
DAOFactory df = DAOFactory.getInstance();
BookDAO bd = df.getBookDAO();
Book b = bd.getBookById(5);
```

…



- Service Layer
- Exception Handling
- Data Transfer Objects



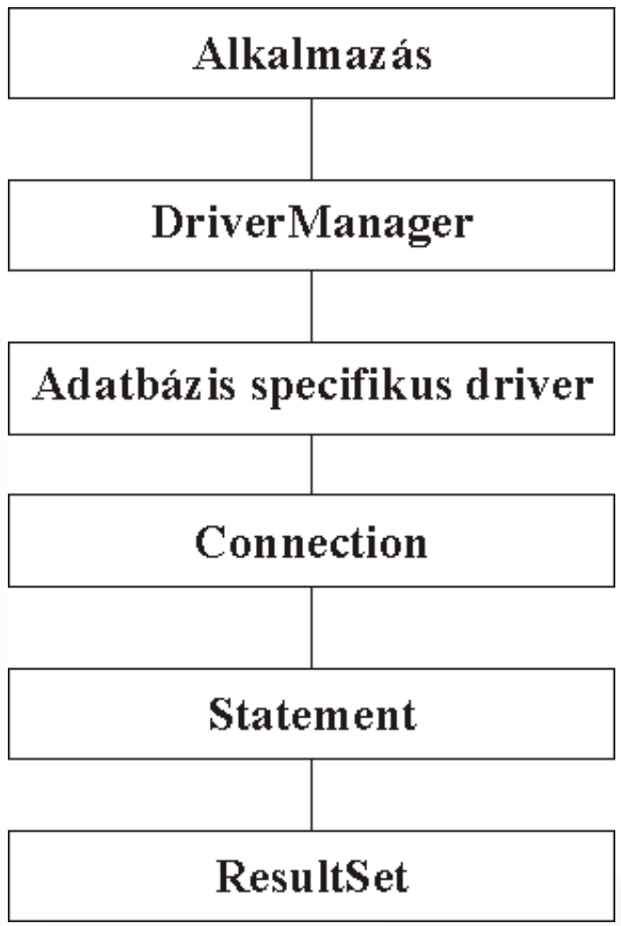
2. rész

Relációs adatbázisok elérése Java programokból

Java Database Connectivity (JDBC)

JDBC

- Call Level Interface (CLI): standard, ami definiálja, hogy egy program hogyan kommunikálhat egy relációs adatbázis management rendszerrel (RDBMS).
- A Microsoft ODBC implementálja és kiterjeszti (ODBC - Open Database Connectivity). C-ben implementált, Javaban közvetlen módon nem alkalmazható (JDBC-ODBC bridges).
- **JDBC:** API, amely definiálja, hogy egy Java programozási nyelvben megírt program hogyan férhet hozzá egy relációs adatbázishoz.
- A JDBC (a DriverManager osztály) egy vagy több adatbázis-specifikus driver-t alkalmaz (egy alkalmazáson belül több adatbázis típussal is dolgozhatunk).
- Minden nagyobb RDBMS rendszer biztosít natív JDBC drivert, és JDBC-ODBC bridge segítségével ODBC driver-ek is alkalmazhatóak.



A kommunikáció lépései:

- A DriverManager-től kérünk egy adatbázis-specifikus drivert.
- A driver létrehozza a kapcsolatot és visszafordít egy Connection objektumot.
- A Connection segítségével létrehozunk egy Statement-et, ami egy SQL parancsot tartalmaz.
- Lekérdezések esetében a Statement objektum egy ResultSet objektumban visszaadja a kérés eredményét.

JDBC – példa

Inventory.mdb (Access DB)

- setup data source

Név	Típus	Hossz
NAME	szöveg	40
QUANTITY	valós	20

```
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class SimpleJDBC {

    public static void main (String[] args) {
        try {
            Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
            String databaseName = "jdbc:odbc:Inventory";
            Connection con = DriverManager.getConnection (
                              databaseName, "username", "password");
            Statement stmt = con.createStatement ();
            ResultSet rs = stmt.executeQuery ("select * from Inventory");
            while (rs.next ()) {
                    System.out.println (rs.getString (1) + ":" + rs.getFloat (2));
            }
        } catch (final SQLException e) {
             e.printStackTrace ();
        } catch (final ClassNotFoundException e) {
             e.printStackTrace ();
        }
    }
}
```

Driver és kapcsolat

- A DB specifikus driver betöltése:

```
Class.forName("com.sybase.jdbc.SybDriver");
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

- A driver-nek megfelelő osztály tartalmaz egy statikus metódust, amelynek segítségével a DriverManager regisztrálja a driver-t.
- Az adatbázis elérése:
  - hol található az adatbázis (a gazda számítógép neve/IP címe)
  - hol hallgatja az RDBMS a kéréseket (melyik porton)
- JDBC - URL:

```
jdbc:<masodlagos_protokoll>:<masodlagos_nev>//gazda_neve:port//adatbazis_neve
```

- Példák:

```
jdbc:oracle:<drivertype>:<user>/<password>@<database>
jdbc:oracle:thin:myuser/mypassword@myserver:1521:mydb
jdbc:microsoft:sqlserver://<server name>[:port>][;<property>=<value>]
jdbc:microsoft:sqlserver://myserver;DatabaseName=mydb;User=myuser;Password=password
jdbc:mysql://[hostname][:port]/dbname[?param1=value1][&param2=value2]...
jdbc:mysql://localhost:3306/bibliospring
```

Driver és kapcsolat

- A Connection tulajdonképpen egy interfész, ami lehetővé teszi kérések küldését és válaszok fogadását
- A kapcsolat létrehozásához szükséges a felhasználó neve és jelszava. Ha több különböző adatbázissal dolgozunk, több driver van betöltve, a DriverManager dolga az aktuális Connection-nak megfelelő driver kiválasztása.
- Statement objektum létrehozása a **`Connection`** interfész **`createStatement`** metódusával:

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

DB hozzáférés, adatfeldolgozás

- A Statement osztály fontosabb metódusai:
  - **`executeQuery (String)`** – SELECT parancsok végrehajtására, az eredmény egy ResultSet objektumban fordítja vissza
  - **`executeUpdate (String)`** – INSERT/UPDATE/DELETE (és CREATE/DROP TABLE) parancsok végrehajtására, a módosított sorok számát fordítja vissza
  - **`execute (String)`** – az előzőek általánosítása, boolean értéket térít vissza, az eredmény(ek) (ResultSet(ek) vagy módosítások száma) metódusok segítségével téríthető vissza, több eredmény esetében is alkalmazható.
- További lehetőségek: automatikusan generált kulcs-értékek lekérdezése stb.
- Az eredményeket a ResultSet objektum tartalmazza:
  - Az ennek megfelelő táblázat mutatója az első sor „elé” mutat, így egyetlen while ciklussal bejárható:

```
while (rs.next()) {
  System.out.println (rs.getString (1) + ":" + rs.getFloat (2));
}
```

  - Megtörténhet, hogy a visszafordított sorok száma 0.
  - **`getXXX (int)`** és **`getXXX (String)`** – érték visszafordítása az aktuális sorból és a paraméter által meghatározott oszlopból.

SQL - Java típus-megfeleltetés

CODESPRING
OUR CODE YOUR SUCCESS

SQL típus	JAVA típus	Metódus
CHAR	String	getString()
VARCHAR	String	getString()
LONGVARCHAR	String	getString()
NUMERIC	java.math.BigDecimal	getBigDecimal()
DECIMAL	java.math.BigDecimal	getBigDecimal()
BIT	boolean/Boolean	getBoolean()
TINYINT	byte/Integer	getByte()
SMALLINT	short/Integer	getShort()
INTEGER	int/Integer	getInt()
BIGINT	long/Long	getLong()
REAL	float/Float	getFloat()
DOUBLE	double/Double	getDouble()
BINARY	byte[]	getBytes()
VARBINARY	byte[]	getBytes()
LONGVARBINARY	byte[]	getBytes()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.Timestamp	getTimestamp()

Scrollable Result Sets

CODESPRING
OUR CODE YOUR SUCCESS

- a Statement objektumra meghívott createStatement és prepareStatement metódusok paraméterei:

```
Statement stmt = con.createStatement (ResultSet.TYPE_SCROLL_SENSITIVE,
                                      ResultSet.CONCUR_READ_ONLY);
```

- A ResultSet objektumon belüli kurzor-pozicionálással kapcsolatos metódusok:

Metódus	Eredmény
`boolean first ()`	Pozicionálás az első sorra
`boolean previous ()`	Pozicionálás az előző sorra
`boolean next ()`	Pozicionálás a következő sorra
`boolean last ()`	Pozicionálás az utolsó sorra
`boolean absolute (int poz)`	Pozicionálás a megadott sorra
`boolean relative (int relPoz)`	Pozicionálás az aktuális pozíció függvényében

PreparedStatement

- PreparedStatement: „előkészített” utasítások
- Egy alkalmazáson belül többször akarjuk alkalmazni ugyanazt az SQL kódot, különböző paraméterekkel: az RDBMS lehetővé teszi előkészített utasítások használatát, melyek létrehozására, elemzésére csak egy alkalommal kerül sor az adatbázis oldalán, ezután többször használhatóak
- A PreparedStatement a Statement osztály leszármazottja:

```
PreparedStatement pstmt = con.prepareStatement (
        "SELECT quantity FROM Inventory WHERE ingredient = ?”);
```

- A ? helyére kerülhet a bemeneti paraméter:

```
public boolean checkInventory () {
    Enumeration e = ingredients.elements ();
    while (e.hasMoreElements ()) {
        InventoryItem i = (InventoryItem) e.nextElement ();
        pStmt.setString (1, i.item);
        ResultSet rs = pStmt.executeQuery ();
        rs.next();
        if (rs.getFloat (1) < i.amount) {
            return false;
        }
    }
    return true;
}
```

PreparedStatement

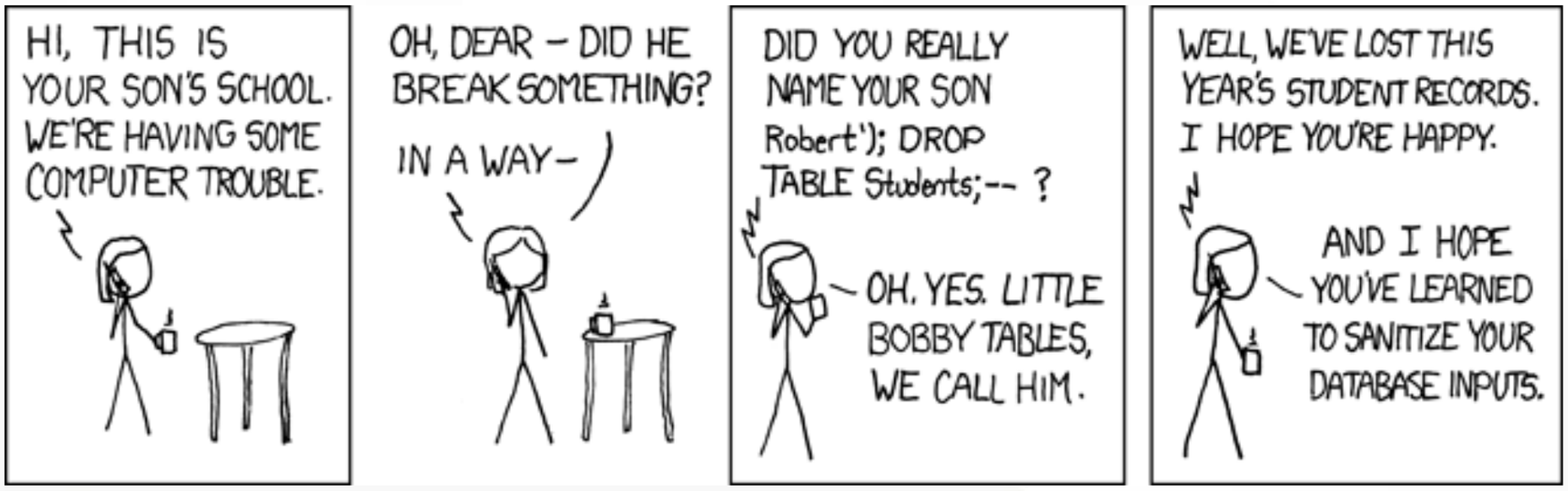
CODESPRING
OUR CODE YOUR SUCCESS

- Az SQL utasítás csak egyszer lesz létrehozva és elemezve – a lekérdezés gyorsabb. A PreparedStatement objektumra meghívott setXXX() metódusok első paramétere jelzi, hogy az SQL parancson belül hányadik paraméterről van szó, a második a paraméter értéke. A Java-SQL típusok közötti megfeleltetések:

Java típus	SQL típus	Metódus
java.math.BigDecimal	NUMERIC	setBigDecimal()
boolean	BIT	setBoolean()
byte	TINYINT	setByte()
short	SMALLINT	setShort()
int	INTEGER	setInt()
long	BIGINT	setLong()
float	REAL	setFloat()
double	DOUBLE	setDouble()
byte[]	VARBINARY sau LONGVARBINARY	setBytes()
java.sql.Date	DATE	setDate()
java.sql.Time	TIME	setTime()
java.sql.Timestamp	TIMESTAMP	setTIMESTAMP()
String	VARCHAR sau LONGVARCHAR	setString()

SQL injection elkerülése

- Biztonsággal kapcsolatos meggondolásból is javasolt lehet PreparedStatement alkalmazása (egyszerű Statement-ek helyett): kivédhető az SQL injection alapú támadási módszer.



Forrás: xkcd.com

Speciális mezők

- Speciális mezők: Binary Large Objects (BLOB)– dokumentumok, képek stb.
- Példa: .gif állományokban tárolt képek Access adatbázisba történő mentése, és kiolvasása/fájlba írása
- JDBCImages.java:

Mező neve	Mező típusa
ID	Numeric(10)
NAME	Text(30)
IMAGE	OLE Object

```
import java.io.*;
import java.sql.*;
public class JDBCImages {

String databaseName = "jdbc:odbc:Images";
Connection con = null;
String pictures[] = {„pelda1.gif",„pelda2.gif", „pelda3.gif",
                     „pelda4.gif", „pelda5.gif"};
PreparedStatement pStmt1 = null;
PreparedStatement pStmt2 = null;
```

Speciális mezők – példa

```
public JDBCImages() {
    try{
          Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
          con = DriverManager.getConnection (databaseName, "", "");
          pStmt1 = con.prepareStatement (
                    "INSERT INTO images (id, name, image) VALUES  (?, ?, ?)");
          pStmt2 = con.prepareStatement ("SELECT image FROM images WHERE id = ?");
    } catch (SQLException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
public void writeRecord (int id) {
    try {
          File inFile = new File (pictures[id]);
          int flength = (int) inFile.length();
          FileInputStream in = new FileInputStream (inFile);
          pStmt1.setInt (1, id);
          pStmt1.setString (2, pictures[id]);
          pStmt1.setBinaryStream (3, in, flength);
          pStmt1.executeUpdate();
    } catch (IOException e) {
          e.printStackTrace();
    } catch (SQLException e) {
          e.printStackTrace();
    }
}
```

Speciális mezők – példa

```
public void readRecord (int id, String fileName) {
    byte[] picture = new byte[1024];
    try {
          pStmt2.setInt (1, id);
          File outFile = new File (fileName);
          FileOutputStream out = new FileOutputStream (outFile);
          ResultSet rs = pStmt2.executeQuery ();
          rs.next ();
          InputStream ins = rs.getBinaryStream (1);
          int n;
          int s = 0;
          while ((n = ins.read (picture)) > 0) {
                    out.write (picture, 0, n);
                    s+=n;
          }
          System.out.println("Total bytes read: " + Integer.toString (s));
          out.close();
    } catch (IOException e) {
          e.printStackTrace();
    } catch (SQLException e) {
          e.printStackTrace();
    }
}
```

```
public static void main (String[] args) {
  int i;
  JDBCImages o = new JDBCImages ();
  o.writeRecord (1);
  o.readRecord (1, "picture1.gif");
 }
}
```

Batch parancsok

- Az összes aktualizálási parancs végrehajtható egyetlen műveleten belül.
- Példa:

```
Statement stmt = con.createStatement ();
//batch parancsok hozzáadása
stmt.addBatch ("INSERT INTO Katedra (Id, Nev) VALUES (5, 'Info')");
stmt.addBatch ("INSERT INTO Tanarok (Id, KatedraId, Nev) VALUES (5,2,'Valaki')");
//Parancsok végrehajtása
int[] updateCounts = stmt.executeBatch ();
```

- Egy tömböt fordít vissza az egyes parancsok által módosított sorok számával.
- PreparedStatement esetében:

```
PreparedStatement pstmt = con.prepareStatement ("DELETE FROM Katedra WHERE Nev = ?");
Enumeration e = v.elements ();
while (e.hasMoreElements()) {
    pstmt.setString (1, (String) e.nextElement());
    pstmt.addBatch ();
}
int [] updateCounts = pstmt.executeBatch();
```

CallableStatement

- A PreparedStatement kiterjesztettje, tárolt SQL utasítások végrehajtására alkalmas. Ezek az utasítások a bemenő paramétereken kívül kimenő paramétereket is használhatnak.
- ```
  {?= call <procedure-name>[<arg1>,<arg2>, ...]}
  {call <procedure-name>[<arg1>,<arg2>, ...]}
  ```
- arg1, arg2… - lehetnek egyaránt be- vagy kimeneti paraméterek (vagy egyszerre mindkettő)
- ```
  CallableStatement cstmt = con.prepareCall ( "{call getTestData(?, ?)}");
  cstmt.registerOutParameter (1, java.sql.Types.TINYINT);
  cstmt.registerOutParameter (2, java.sql.Types.DECIMAL, 3);
  cstmt.executeQuery ();
  byte x = cstmt.getByte (1);
  java.math.BigDecimal n = cstmt.getBigDecimal (2, 3);
  ```