Projekt BD

DataSource

- Zalecany zamiennik dla DriverManager
- Posiada właściwości, które można zmieniać setterami i getterami
  - każdy sterownik może mieć inne nazwy właściwości!
- Może być wpisywany do JNDI
- Może wspomagać przydział połączeń (Connection Pooling)

Zmiana w programie

- Zamiast:

```
Class.forName("com.mysql.jdbc.Driver");
Connection con =
        DriverManager.getConnection("jdbc:mysql://localhost/baza",
                                    "lab","lab");
```

- Używamy:

```
com.mysql.jdbc.jdbc2.optional.MysqlDataSource ds =
        new com.mysql.jdbc.jdbc2.optional.MysqlDataSource();
ds.setUser("lab");
ds.setPassword("lab");
ds.setDatabaseName("baza");
Connection con = ds.getConnection();
```

Utrzymywanie połączenia

- Nawiązanie połączenia jest czasochłonne – lepiej więc raz nawiązane połączenie używać do kolejnych zapytań
- Problem: jak przechwać informację o połączeniu, jeśli jest ono używane w różnych obiektach
- Rozwiązanie: własny obiekt "Połączenie" dostępny globalnie (wszędzie w aplikacji)
- Pytanie: jak to zrobić?
- Rozwiązanie naiwne: przekazywanie referencji na obiekt do każdej klasy – w dużych aplikacjach kłopotliwe

Przechowywanie połączenia

- Pole statyczne w jakiejś klasie

```
class SqlConnection {
    static Connection con;
}
```

- Singleton
  - klasa która z założenia ma tylko jedną implementację
  - tworzy się tylko jeden obiekt i najczęściej nie da się stworzyć nowego

Singleton

```
public class MojSingleton {
   private static MojSingleton mojSingleton = null;
   public static MojSingleton getInstance() {
       if(mojSingleton==null) {
             mojSingleton = new MojSingleton();
       }
       return mojSingleton;
   }

   private MojSingleton(){
       //tu tworzenie obiektu
   }
}
```

Użycie singletona

- Tak nie można!

```
MojSingleton s = new MojSingleton();
```

- Tak można w każdym miejscu (i zawsze dostajemy ten sam obiekt!)

```
MojSingleton s = MojSingleton.getInstance();
```

Przechowanie Connection

```
import java.sql.*;
class DBCon {
   private static Connection con;
   static Connection getConnection() {
       if(con==null) {...}
       return con;
   }
}
```

W programie:

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

Inicjalizacja połączenia

```
import java.sql.*;
class DBCon {
   private static Connection con;
   static Connection getConnection() {
   if(con==null) {
    try{
     Class.forName("...");
     con = DriverManager.getConnection("<url>",
                                       "...","...");
    }catch(SQLException ec) {...}
     catch(ClassNotFoundException ex) {...}
   return con;
}
```

Użycie DBCon

```
...
Connection c = DBCon.getConnection();
try{
    Statement stmt = c.createStatement();
    ResultSet rs = stmt.executeQuery("...");
    while(rs.next())
          System.out.println(rs.getString(1));
}catch(SQLException ec) {...}
...
```

Użycie DBCon

```
...
try{
    Statement stmt =
          DBCon.getConnection().createStatement();
    ResultSet rs = stmt.executeQuery("...");
    while(rs.next())
         System.out.println(rs.getString(1));
}catch(SQLException ec) {...}
...
```

Użycie DBCon

```
...
try{
    ResultSet rs = DBCon.getConnection()
          .createStatement().executeQuery("...");
    while(rs.next())
          System.out.println(rs.getString(1));
}catch(SQLException ec) {...}
...
```

Zalety rozwiązania

- Połączenie nawiązywane jest tylko raz na początku
- Kolejne wywołania używają już stworzonego połączenia
- Kod się uprościł
- Zmiany parametrów połączenia dokonuje się tylko w jednym miejscu

Zadanie

- Przerobienie kodu – osobna klasa połączeniowa

Data Access Objects (DAO)

- Główna idea: uniezależnić aplikację od źródła danych
- Interfejs DAO zapewnia wszystkie operacje na danych (tzw. CRUD – Create Retrieve Update Delete)
- Składniki:
  - interfejs DAO
  - źródło danych (DataSource)
  - obiekt transferowy (JavaBean)

Przykładowe DAO

- Typowe metody interfejsu DAO
- Klasa (lub interfejs) PracownicyDAO
  - void add(Pracownik pracownik)
  - Collection findByNazwisko(String nazwisko)
  - Pracownik get(Integer nr_prac) // nr_prac jest kluczem!
  - void save(Pracownik pracownik)
  - void delete(Pracownik pracownik)
- Klasa Pracownik (JavaBean – obiekt transferowy):
  - pola nr_prac, nazwisko, ...
  - gettery i settery dla każdego z nich

Bean Pracownik

```
class Pracownik {
  int nrp;
  String nazw;
  Date dataur;

  public String getNrp() {return nrp;}
  public void setNrp(int nrp) {this.nrp = nrp;}

  public String getNazw() {return nazw;}
  public void setNazw(String nazw) {this.nazw=nazw;}

  public Date getDataur() {return dataur;}
  public void setDataur(Date dataur) {this.dataur=dataur;}
}
```

Warstwa DAO

- Implementacja metod wykonujących operacje na bazie
- Często operacje te różnią się dla różnych serwerów
- Dlatego najczęściej przygotowuje się:
  - interface PracownicyDAO
  - implementacje:
    - class MySQLPracownicyDAO implements PracownicyDAO
    - class OraclePracownicyDAO implements PracownicyDAO
    - ...
- Uwaga: większość operacji jest w SQL i jest taka sama dla każdej implementacji

DAOFactory

- Problem: jak podmieniać implementacje w aplikacji bez zmiany kodu aplikacji?
- Rozwiązanie: klasa Factory – produkuje obiekty
- Zamiast wywoływać
  - PracownicyDAO p = new MySQLPracownicyDAO()
- Wywołujemy
  - PracownicyDAO p = factory.getPracownicyDAO()
- Dlaczego jest to przydatne:
  - właściwości obiektów są ustawiane automatycznie
  - można używać różnych implementacji obiektów – zmiana implementacji to tylko zmiana w fabryce a nie w kodzie aplikacji

Bean Car

```
public class Car {
   int idc;
   String make;
   String model;
   String regNum;
   Double price;
   public int getIdc() {
       return idc;
   }
   //dalsze gettery i settery
}
```

Interfejs DAO

- Interfejs CarDAO
- Metody:
  - void add(Car car)
  - void delete(int numer)
  - Car get(int numer)
  - List<Car> find(String make)
- Implementacja: CarDAOImpl

Statyczne połączenie z bazą

```
public class CarDAOImpl implements CarDAO {
   static Connection con;
   static {
        com.mysql.jdbc.jdbc2.optional.MysqlDataSource ds =
                new com.mysql.jdbc.jdbc2.optional.MysqlDataSource();
        ds.setUser("root");
        ds.setPassword("");
        ds.setDatabaseName("nowa");
        try{
                con = ds.getConnection();
        }catch(SQLException ex) {ex.printStackTrace();}
   }
```

Metoda add

```
public void add(Car car) {
   try{
       con.createStatement().executeUpdate(
              "insert into car values("+
              car.getIdc()+","+
              "'"+car.getMake()+"',"+
              "'"+car.getModel()+"',"+
              "'"+car.getRegNum()+"',"+
              car.getPrice()+")"
       );
   }catch(SQLException ex) {ex.printStackTrace();}
}
```

Metoda find

```
public List<Car> find(String make) {
List<Car> carList = new ArrayList<Car>();
try{
ResultSet rs = con.createStatement().executeQuery(
   "select * from car where make like '"+make+"'");
while(rs.next()) {
   Car car = new Car();
   car.setIdc(rs.getInt("idc"));
   car.setMake(rs.getString("make"));
   car.setModel(rs.getString("model"));
   car.setRegNum(rs.getString("regnum"));
   car.setPrice(rs.getDouble("price"));
   carList.add(car);
}
}catch(SQLException ex) {ex.printStackTrace();}
return carList;
}
```

Zadanie

- Przygotowanie interfejsu DAO dla swojego rozwiązania
- Implementacja