przygotował: pawel@kasprowski.pl

Aplikacje Internetowe

Łączenie z bazą danych

Podstawy

- Klient
- Serwer
- Sterownik
  - Własne API (Application Programmer Interface)

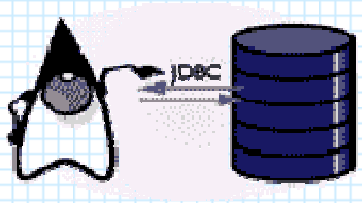
Łączenie z bazą danych

- Sterownik
- Protokół komunikacyjny
- Adres serwera
- Port nasłuchowy
- Przesłanie zapytania SQL
- Odebranie rezultatów

Język Java - JDBC

JDBC API pozwala na:

- Ustalenie połączenia z bazą
- Wysyłanie poleceń SQL
- Przetwarzanie rezultatów



Sterowniki JDBC

1) JDBC-ODBC Bridge
2) Native-API partly-Java driver
3) Net-protocol all-Java driver
4) Native-protocol all-Java driver

Główne klasy i interfejsy JDBC

- Driver
- DriverManager
- Connection
- Statement, PreparedStatement, CallableStatement
- ResultSet
- DatabaseMetaData
- ResultSetMetaData
- Types
- SQLException
- SQLWarning

Sposób działania JDBC

Załaduj sterownik

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

Połącz się ze źródłem danych

```
Connection con = DriverManager.getConnection( URL,
Username, Password );
```

Stwórz polecenie

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

Zapytaj bazę danych

```
ResultSet result = stmt.executeQuery("SELECT * FROM
table1");
```

Ładowanie sterownika

Statyczne:

```
import drivername;
```

Dynamiczne:

```
Class.forName(”drivername”);
```

Na przykład:

```
com.mysql.jdbc.Driver
```

lub

```
com.microsoft.sqlserver.jdbc.SQLServerDriver
```

Połączenie ze źródłem danych

```
Connection con = DriverManager.getConnection
   ( URL, Username, Password );
```

URL = jdbc:<subprotocol>:<subname>

<subname> = //host:port/dbname;params

Na przykład:

jdbc:mysql://127.0.0.1/baza

lub

jdbc:~~microsoft:~~sqlserver://db.xyz.pl:1433

Pierwszy przykład JDBC

```
import java.sql.*;
class DBExample1{
public static void main(String[] args){
  try{
    Class.forName("com.mysql.jdbc.Driver");
    Connection con = DriverManager.getConnection("jdbc:mysql://127.0.0.1/baza",
                                                 "lab","lab");
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select * from pracownicy");
    while(rs.next())
      System.out.println(rs.getString(1)+"|"+rs.getString(4));
    con.close();
    }
  catch(SQLException ec) {ec.printStackTrace(); }
  catch(ClassNotFoundException ex) {ex.printStackTrace();}
  }
}
```

DBExample1.java

Ładowanie sterownika

- Zwykle sterownik typu 4
- Ściągnięcie pliku jar ze sterownikiem
- Umieszczenie go na ścieżce
- Zmiana nazwy sterownika i url'a w programie

Przykłady sterowników

- SQL Server (http://msdn.microsoft.com/data/ref/jdbc)

```
java -classpath ".;sqljdbc4.jar" %1
```

- MySQL (http://www.mysql.com/products/connector/j/)

```
java -classpath ".;mysql-connector-java.jar" %1
```

- PostgreSQL (http://jdbc.postgresql.org)

```
java -classpath ".;postgres-8.2dev-503.jdbc2ee.jar" %1
```

run.bat

Zmiana sterownika

```
import java.sql.*;
class DBExample1{
public static void main(String[] args){
  try{
    Class.forName("com.mysql.jdbc.Driver");
    Connection con =
        DriverManager.getConnection("jdbc:mysql://127.0.0.1/baza","lab", "lab");
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select * from pracownicy");
    while(rs.next())
      System.out.print(rs.getString(1)+"|"+rs.getString(4)+"\n");
    con.close();
    }
  catch(SQLException ec) { System.err.println(ec.getMessage()); }
  catch(ClassNotFoundException ex) {System.err.println("Cannot find driver.");}
  }
}
```

DBExample1.java

Zmiana sterownika

```
import java.sql.*;
class DBExample1{
public static void main(String[] args){
  try{
    Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
    Connection con =
        DriverManager.getConnection("jdbc:sqlserver://localhost", "lab", "lab");
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select * from pracownicy");
    while(rs.next())
      System.out.print(rs.getString(1)+"|"+rs.getString(4)+"\n");
    con.close();
    }
  catch(SQLException ec) { System.err.println(ec.getMessage()); }
  catch(ClassNotFoundException ex) {System.err.println("Cannot find driver.");}
  }
}
```

DBExample1.java

ResultSet

```
ResultSet rs=stmt.executeQuery("select * from table");
```

- dostęp do jednego wiersza danych jednocześnie
- *next()* – przejście do następnego wiersza
- *getXXX(i)* – wartość i-tej kolumny
- Najczęściej: *getInt(), getString(), getDate(), getDouble()*...
- Informacje o strukturze danych (nazwy i typy kolumn) - MetaData

Tworzenie polecenia (Statement)

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

- *executeQuery()* uruchamia polecenia SQL zwracające listę wierszy jako *ResultSet*.
- *executeUpdate()* uruchamia polecenia SQL modyfikujące dane lub strukturę bazy
- *execute()* uruchamia polecenie SQL dowolnego typu

executeUpdate

```
int rows=stmt.executeUpdate(
   "update table set field=3 where id=5");
```

- Dla poleceń DML i DDL
- Zwraca ilość przetworzonych wierszy
- Zwraca zero dla poleceń DDL (CREATE TABLE itp.)

PreparedStatement

```
PreparedStatement pstmt = con.prepareStatement(
   "select * from table1
   where id=? and field like ?")
```

- Pre-kompilacja zapytań po stronie serwera
- Przyspiesza działanie podobnych zapytań
- Czytelniejszy kod
- Pozwala na użycie parametrów(marker ‘?’)
- setXXX() – ustawienie wartości parametru

PreparedStatement

```
   // definicja polecenia
PreparedStatement pstmt = con.prepareStatement(
   "select * from table1 where id<? and field like ?");

   // ustawienie parametrów
pstmt.setInt(1,20);
pstmt.setString(2,"value");

   // uruchomienie
Rs = pstmt.executeQuery();
```

SQLException

```
import java.sql.*;
class DBExample1{
public static void main(String[] args){
  try{
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection con = DriverManager.getConnection("jdbc:odbc:uran","lab","lab");
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select * from pracownicy");
    while(rs.next())
      System.out.print(rs.getString(1)+"|"+rs.getString(4)+"\n");
    con.close();
    }
  catch(SQLException ec) { System.err.println(ec.getMessage()); }
  catch(ClassNotFoundException ex) {System.err.println("Cannot find driver.");}
  }
}
```

SQLException

```
import java.sql.*;
class DBExample1{
public static void main(String[] args){
  try{
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  }catch(ClassNotFoundException ex) {System.err.println("Cannot find driver.");}
  try{
    Connection con = DriverManager.getConnection("jdbc:odbc:uran","lab","lab");
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select * from pracownicy");
    while(rs.next())
      System.out.print(rs.getString(1)+"|"+rs.getString(4)+"\n");
    con.close();
  }catch(SQLException ec) { System.err.println(ec.getMessage()); }
}
}
```

Transakcje

- Niepodzielny blok poleceń
- Utrzymanie spójności danych
- Domyślnie – AutoCommit – każde polecenie to osobna transakcja

```
con.setAutoCommit(false)
con.commit()
con.rollback()
```

Przykład transakcji (przelew bankowy)

```
...
con.setAutoCommit(false);
try{
int n1 = stmt.executeUpdate("update accounts set
   amount=amount-100 where custid=75");
int n2 = stmt.executeUpdate("update accounts set
   amount=amount+100 where custid=43");
if(n1==1 && n2==1)
  con.commit();
else
  con.rollback();
}catch(SQLException ec)
  {con.rollback();
   System.err.println(ec.getMessage());}
...
```

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 niemożliwe

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) {...stwórz połączenie...}
      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("com.mysql.jdbc.Driver");
     con = DriverManager.getConnection("<url>",
                                       "user","pass");
    }catch(Exception ec) {ec.printStackTrace();}
   }
   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) {ec.printStackTrace();}
...
```

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) {...}
...
```

Przykład aplikacji

- Wyświetlenie listy pracowników
- Plik: index.jsp
- Połączenie: DBCon.java
- Generacja listy: DBHandler.java

Index.jsp

Lista pracowników:

```
<%
pl.test.DBHandler dbh = new pl.test.DBHandler();
out.write(dbh.getList());
%>
```

Można też to samo za pomocą JSTL/EL:

```
<jsp:useBean id="dbh" class="pl.kurs.db.DBHandler"/>
${dbh.list}
```

DBHandler.java

```
public String getList() {
String txt = "";
try{
    ResultSet rs = DBCon.getConnection()
        .createStatement()
            .executeQuery("select * from pracownicy");
    while(rs.next())
        txt+= rs.getString("nr_prac") +" "+rs.getString("nazwisko");
}catch(SQLException ec) {ec.printStackTrace();}
return txt;
}
```

Problem z kodem (1)

- Co jeśli wiele jednoczesnych połączeń z bazą danych?
- Lepiej zamiast jednego połączenia przechowywać fabrykę połączeń!
- Gotowa klasa: DataSource
- DataSource.getConnection() zwraca połączenie
- Możliwe dzięki temu zaimplementowanie ConnectionPooling

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 linii:

```
Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager
        .getConnection(" jdbc:mysql://127.0.0.1/baza ","lab","lab");
```

- Wprowadzamy (dla MySQL):

```
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();
```

Connection Pooling

- Zalecane przy aplikacjach wielowątkowych używanych jednocześnie przez wielu użytkowników (np. aplikacje internetowe)
- Singleton (fabryka) tworzy kilka połączeń z bazą i udostępnia je innym obiektom
- Gdy obiekt skończy pracę nie zwraca połączenia lecz zwalnia je – "oddaje" do singletona
- Apache Tomcat ma gotowy do użycia moduł: DBCP
  - wystarczy wpisać jego konfigurację do server.xml

Użycie DBCP

- Stosuje connection pooling
- Używa sterownika, który podamy
  - jest niezależny od serwera bazy danych
- Konfiguracja połączenia w Context
  - conf/server.xml
  - conf/Catalina/localhost/aplikacja.xml
  - aplikacja/META-INF/context.xml
- W aplikacji używamy JNDI lookup

Konfiguracja połączenia

```
<Context path="/web1" reloadable="true">
    <Resource name="jdbc/mydb" auth="Container"
    type="javax.sql.DataSource"
    maxActive="100" maxIdle="30" maxWait="10000"
    username="myuser" password="mypass"
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost:3306/mydb?
                                        autoReconnect=true"/>
</Context>
```

Użycie DBCP w aplikacji

```
public Connection getConnection() {
  try{
     Context initContext = new InitialContext();
     DataSource ds =(DataSource)initContext.lookup(
                                        "java:/comp/env/jdbc/mydb");
     return ds.getConnection();
  }catch(SQLException ec) { ... }
   catch(NamingException ne) { ...}
  return null;
  }
```

Używanie połączenia

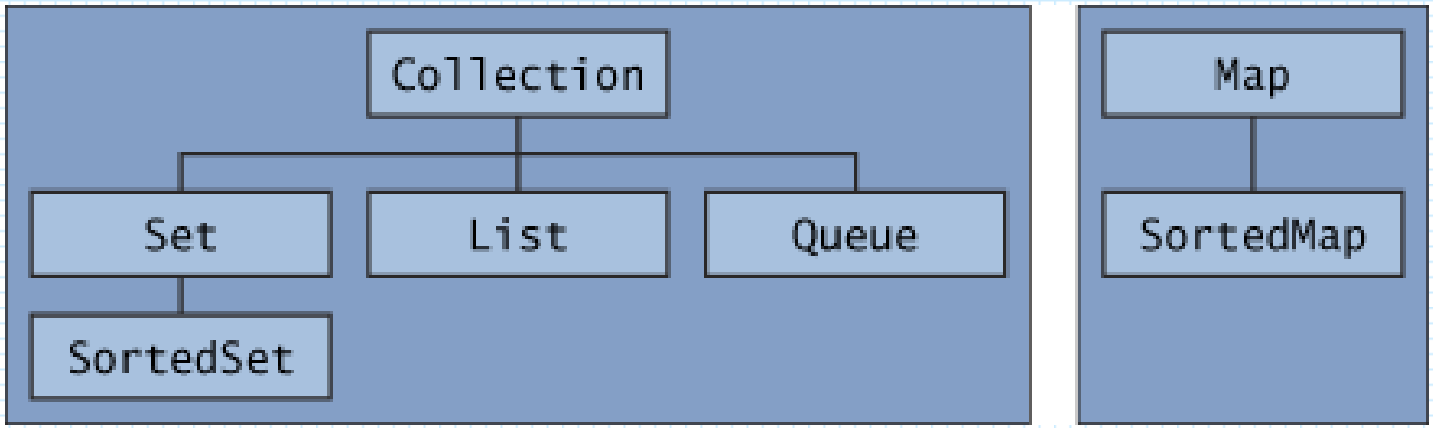
- Otwarcie połączenia:
  - Connection con = ds.getConnection();
- Zamykanie połączenia:
  - con.close()
- W rzeczywistości nie jest ono zamykane ale wraca do puli połączeń!
- Niezamknięcie połączenia jest potencjalnie niebezpieczne bo może wyczerpać pulę!

Problem z kodem (2)

- Formatowanie listy zaszyte w kodzie metody w Javie
- Ograniczona funkcjonalność pliku JSP
- Zmiana formatowania (np. zrobienie tablicy) to ingerencja w kod Javy
- Lepiej przekazywać do JSP dane niesformatowane!
- Jak przekazać listę danych?
  - Odpowiedź: kolekcja!

Kolekcje

- Gotowe do użycia struktury o wbudowanej przydatnej funkcjonalności
- Element *Java Collections Framework*
- Oparte na grupie interfejsów



Metody interfejsu Collection

- add(Object o)
- addAll(Collection c)
- contains(Object o)
- containsAll(Collection c)
- remove(Object o)
- removeAll(Collection c)
- size()
- iterator()

Metody interfejsu Iterator

- next()
  - zwraca kolejny obiekt kolekcji
- hasNext()
  - true, gdy jest kolejny obiekt w kolekcji
- remove()
  - usuwa obiekt i przechodzi na następny

```
Iterator it = vkl.iterator();
while(it.hasNext()) {
    System.out.println(it.next());
}
```

Implementacje interfejsu List

- ArrayList
  - tablica rozszerzalna
  - dodatkowe metody:
    - set(index, element) – zastępuje obiekt o podanym indeksie
    - add(index, element) – dodaje element w podany indeks
    - get(index) – zwraca obiekt z podanego indeksu
- LinkedList
  - metoda iterator() zwraca obiekt implementujący ListIterator
  - dodatkowe metody ListIterator
    - previous()
    - hasPrevoius()
    - add(element)

Implementacje interfejsu Set

- Nie może być powtórzeń (elementów takich samych)
- Metoda contains() działa znacznie efektywniej
- Kolekcje nie mogą zawierać duplikatów
- HashSet – elementy nieuporządkowane
- TreeSet – elementy uporządkowane
  - elementy muszą implementować interfejs Comparable (np. klasa String implementuje)

Pętla foreach

- Działa od wersji 5.0
- Upraszcza użycie iteracji

```
for(Object o: kolekcja) {
    System.out.println(o);
}
```

Generics (od 5.0)

- Kolekcja może zawierać obiekty dowolnego typu
- Można ograniczyć kolekcję do jednego typu
- Zamiast
  - List lista = **new** ArrayList();
- Piszemy
  - List<Klient> lista = **new** ArrayList<Klient>();
- Teraz do listy można dodać tylko dane zdefiniowanego typu
- Ten typ to może być także interfejs!
- Dzięki temu więcej błędów wykryjemy już na etapie kompilacji

Prosty przykład 1

```
// stwórz kolekcję
Collection<String> c = new ArrayList<String>()
// dodaj elementy
for(int i=0;i<100;i++)
        c.add("aa"+i);
// wyświetl elementy
for(String s:c)
        System.out.println(s);
```

- Dla ArrayList i LinkedList elementy są w kolejności dodawania
- Dla HashSet elementy są nieposortowane
- Dla TreeSet elementy są posortowane tekstowo

Prosty przykład 2

```
// stwórz kolekcję
Collection<String> c = new ArrayList<String>()
// dodaj elementy (teraz cały czas taki sam!)
for(int i=0;i<100;i++)
        c.add("aa");
// wyświetl elementy
for(String s:c)
        System.out.println(s);
```

- Dla ArrayList i LinkedList elementy są w kolejności dodawania
- Dla HashSet i TreeSet jest tylko jeden element

Użycie ArrayList

```
// stwórz kolekcję
Collection<String> c = new ArrayList<String>()
// dodaj elementy
for(int i=0;i<10;i++)
   ((ArrayList)c).add(0,"aa"+i);
// wyświetl elementy
for(String s:c)
        System.out.println(s);
```

- Nowe elementy są teraz dodawane zawsze na początek

Po co kolekcje JFC

- Mniejszy wysiłek przy programowaniu
- Zwiększa jakość i szybkość programu
- Ułatwia wymianę informacji pomiędzy różnymi fragmentami kodu (bibliotekami)
- Ujednolica pracę w różnych miejscach kodu
- Pozwala na tworzenie własnych implementacji

Rozwiązanie z kolekcją

- DBHandler.getList() zwraca jeden tekst ze wszystkimi danymi
- DBHandler.getList2() będzie zwracało kolekcję wierszy
- W pliku index.jsp wyświetlone będą w pętli

DBHandler.getList()

```
public String getList() {
String txt = "";
try{
    ResultSet rs = DBCon.getConnection()
        .createStatement()
            .executeQuery("select * from pracownicy");
    while(rs.next())
        txt+= rs.getString("nr_prac") +" "+rs.getString("nazwisko");
}catch(SQLException ec) {ec.printStackTrace();}
return txt;
}
```

DBHandler.getList2()

```
public List<String> getList2() {
List<String> lista = new ArrayList<String>();
try{
    ResultSet rs = DBCon.getConnection()
        .createStatement()
            .executeQuery("select * from pracownicy");
    while(rs.next())
        String txt = rs.getInt("nr_prac")+""+rs.getString("nazwisko");
        lista.add(txt);
}catch(SQLException ec) {ec.printStackTrace();}
return lista;
}
```

Stary index.jsp

Lista pracowników:

```
<%
pl.kurs.db.DBHandler dbh = new pl.kurs.db.DBHandler();
out.write(dbh.getList())
%>
```

Nowy index.jsp

```
Lista pracowników:
<%
pl.kurs.db.DBHandler dbh = new pl.kurs.db.DBHandler();
java.util.List<String> lista = dbh.getList2();
for(String linia:lista)
    out.write(linia+"<br/>");
%>
```

JSTL – użyjmy tagów!

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

Lista pracowników:

<jsp:useBean id="dbh" class="pl.kurs.db.DBHandler"/>

<c:forEach var="linia" items="${dbh.list2}">
    ${linia}<br/>
</c:forEach>
```

Co uzyskaliśmy?

- Całe formatowanie tylko w HTML
- Kod w Javie tylko generuje dane i przekazuje je w postaci listy
- Wciąż jednak brak dostępu do poszczególnych danych pracownika
  - Nie da się formatować w zależności od wartości kolumny
    - np. pogrubić nazwiska
  - Nie da się zrobić tabelki z kolumnami: numer, nazwisko

Bean zamiast tekstu

- DBHandler.getList3() będzie zwracało kolekcję
  - Kolekcję obiektów klasy Pracownik!
- Klasa Pracownik: JavaBean z getterami i setterami
- Pola
  - Numer
  - Nazwisko
  - Plec
- Dla każdego pola getter i setter

Pracownik.java

```
public class Pracownik {
    private int numer;
    private String nazwisko;
    private String plec;
    public int getNumer() {return numer;}
    public void setNumer(int numer) {this.numer = numer;}
    public String getNazwisko() {return nazwisko;}
    public void setNazwisko(String nazwisko)
            {this.nazwisko = nazwisko;}
    public String getPlec() {return plec;}
    public void setPlec(String plec) {this.plec = plec;}
}
```

DBHandler.getList2()

```
public List<String> getList2() {
List<String> lista = new ArrayList<String>();
try{
    ResultSet rs = DBCon.getConnection()
            .createStatement()
                    .executeQuery("select * from pracownicy");
    while(rs.next())
            String txt = rs.getInt("nr_prac")+""+rs.getString("nazwisko");
            lista.add(txt);
}catch(SQLException ec) {ec.printStackTrace();}
return lista;
}
```

DBHandler.getList3()

```
public List<Pracownik> getList3() {
List<Pracownik> lista = new ArrayList<Pracownik>();
try{
    ResultSet rs = DBCon.getConnection()
        .createStatement()
            .executeQuery("select * from pracownicy");
    while(rs.next())
        Pracownik p = new Pracownik();
        p.setNumer(rs.getInt("nr_prac"));
        p.setNazwisko(rs.getString("nazwisko"));
        p.setPlec(rs.getString("plec"));
        lista.add(p);
}catch(SQLException ec) {ec.printStackTrace();}
return lista;}
```

Nowy index.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
Lista pracowników:

<jsp:useBean id="dbh" class="pl.kurs.db.DBHandler"/>

<c:forEach var="linia" items="${dbh.list2}">
    ${linia}<br/>
</c:forEach>
```

Nowy index.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
Lista pracowników:

<jsp:useBean id="dbh" class="pl.kurs.db.DBHandler"/>

<c:forEach var="linia" items="${dbh.list2}">
    ${linia}<br/>
</c:forEach>

<c:forEach var="prac" items="${dbh.list3}">
    ${prac.numer}  ${prac.nazwisko}  ${prac.plec}<br/>
</c:forEach>
```

Efekt

- Do JSP przychodzi jeden obiekt, który jest typu "kolekcja pracowników" List<Pracownik>
- W JSP można prostą pętlą wydobyć wszystkie dane z kolekcji i dowolnie formatować

```
<c:forEach var="prac" items="${dbh.list3}">
    <c:if test="${prac.plec=='K'}"><font color="#ff0000"></c:if>
    ${prac.numer} ${prac.nazwisko} ${prac.plec}<br/>
    <c:if test="${prac.plec=='K'}"></font></c:if>
</c:forEach>
```

Dziękuję za uwagę