Zaawansowane Programowanie obiektowe

Wykład 8
Bazy danych w Javie

dr hab. Szymon Grabowski
dr inż. Wojciech Bieniecki
mgr inż. Michał Paluch
wbieniec@kis.p.lodz.pl
http://wbieniec.kis.p.lodz.pl

CZĘŚĆ 1
JDBC

Wprowadzenie

JDBC: Java Database Connectivity

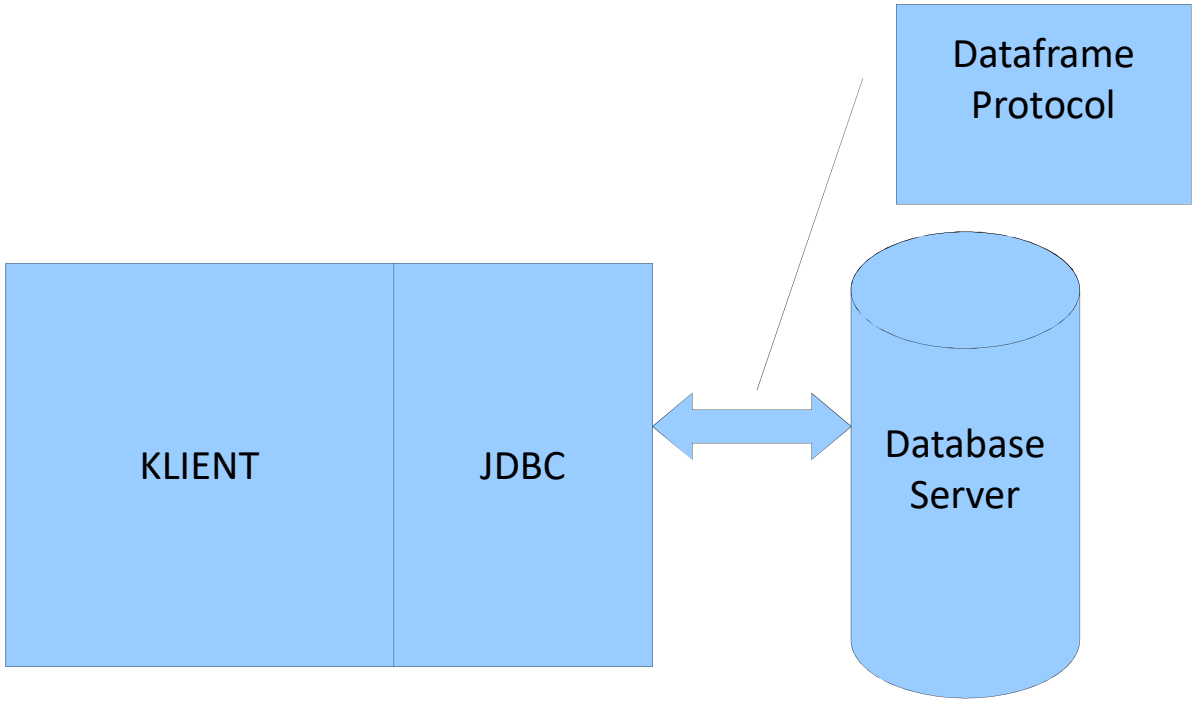
Technologia pozwalająca korzystać z danych zawartych w bazach.

Udostępnia interfejs dzięki któremu można formułować zapytania do bazy, pobierać rekordy - opakowywać je w wygodne obiekty języka Java i potem korzystać z nich w programie.

JDBC przeznaczone jest dla obsługi *tabelarycznych*, ogólnie: relacyjnych baz danych

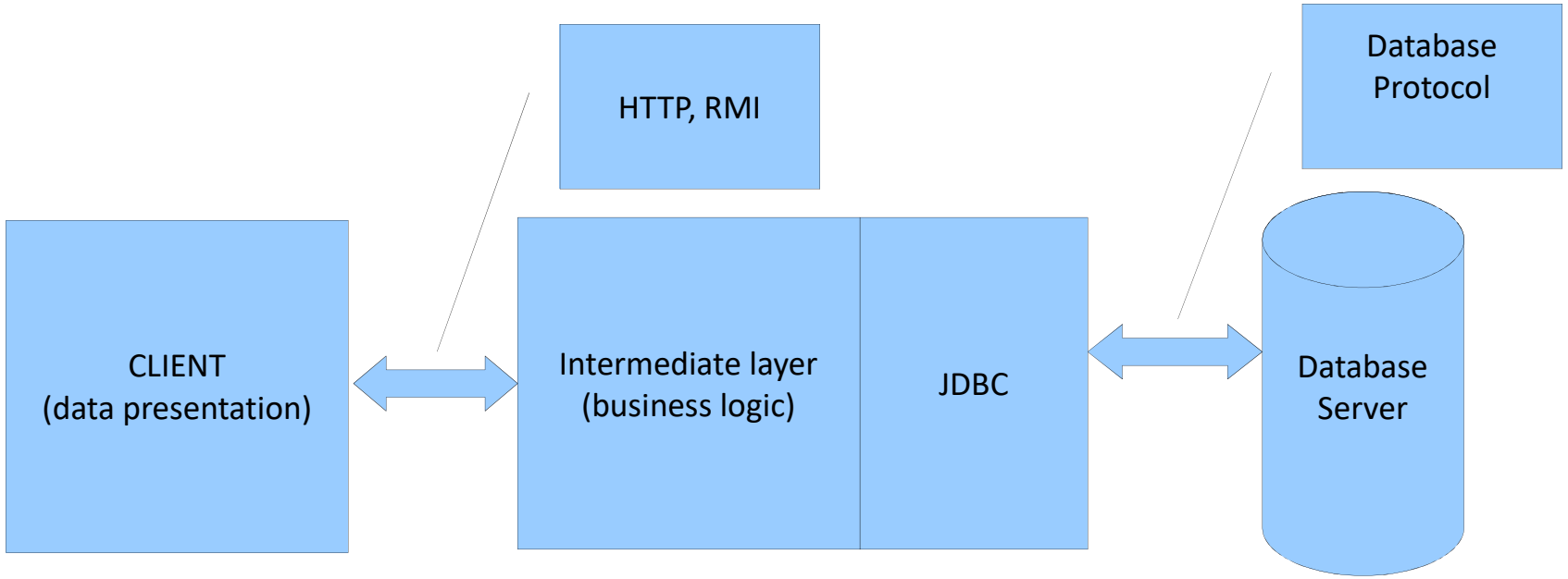
Wykorzystanie JDBC

Tradycyjny model klient-serwer



Wykorzystanie JDBC

Architektura trójwarstwowa



Sterownik JDBC

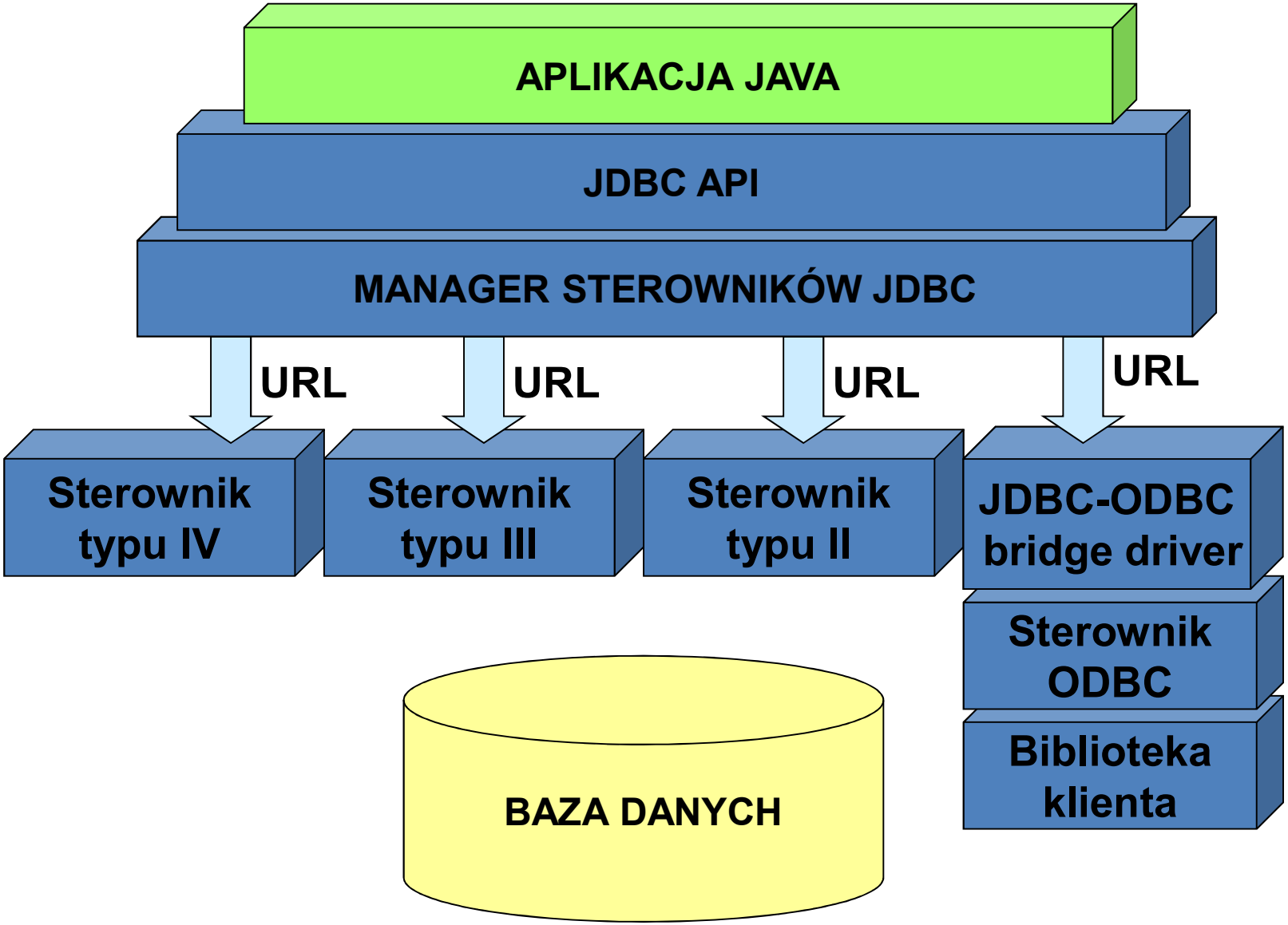
Sterownik JDBC (JDBC driver) jest implementacją API JDBC dostarczaną przez konkretnego dostawcę bazy danych, na potrzeby komunikacji z tą właśnie bazą.

Sterownik implementuje interfejs **java.sql.Driver**

Dostęp do sterowników obywa się poprzez zarządcę sterowników (JDBC driver manager), obiekt klasy **java.sql.DriverManager**

java.sql.DriverManager

Dostęp do *połączeń* z różnymi bazami danych, za pomocą sterowników różnych typów



Rodzaje sterowników JDBC

Typ 1 – mostek JDBC-ODBC

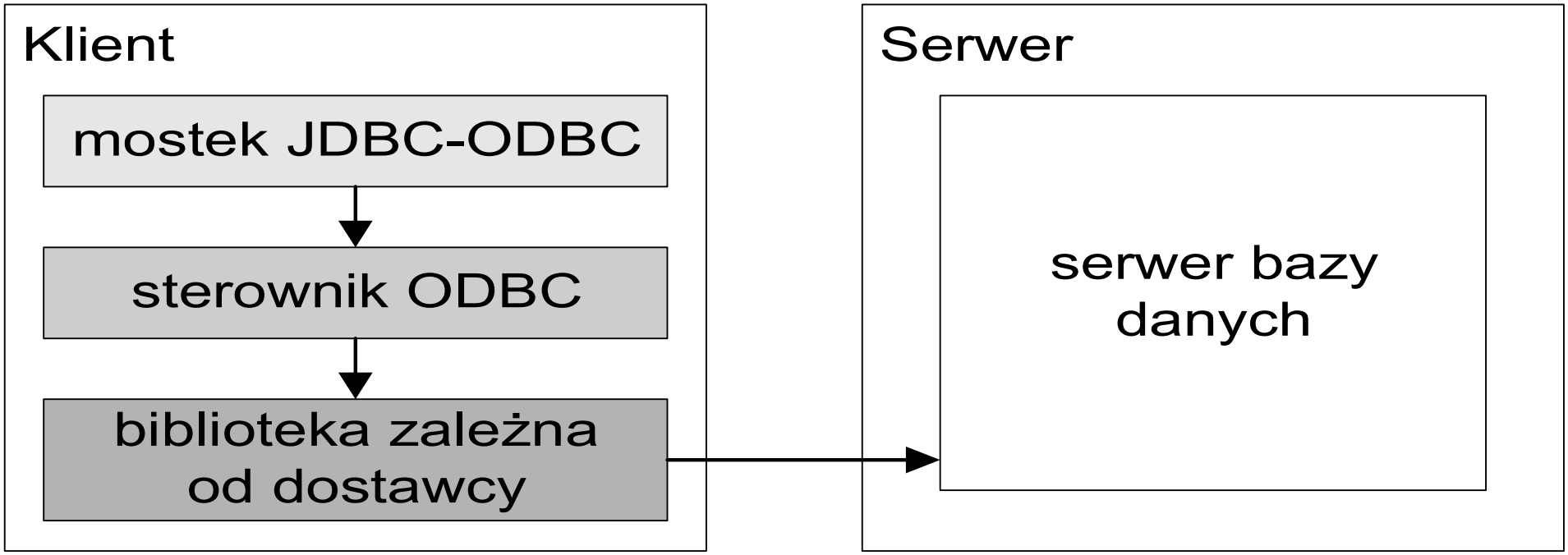
Typ 2 – mieszany kod rodzimy klienta bazy danych i Javy

Typ 3 – czysty kod Javy komunikujący się niezależnym od bazy danych protokołem sieciowym, tłumaczonym przez pośredni serwer

Typ 4 – czysty kod Javy komunikujący się zależnym od bazy danych protokołem sieciowym

Typ I sterownika JDBC

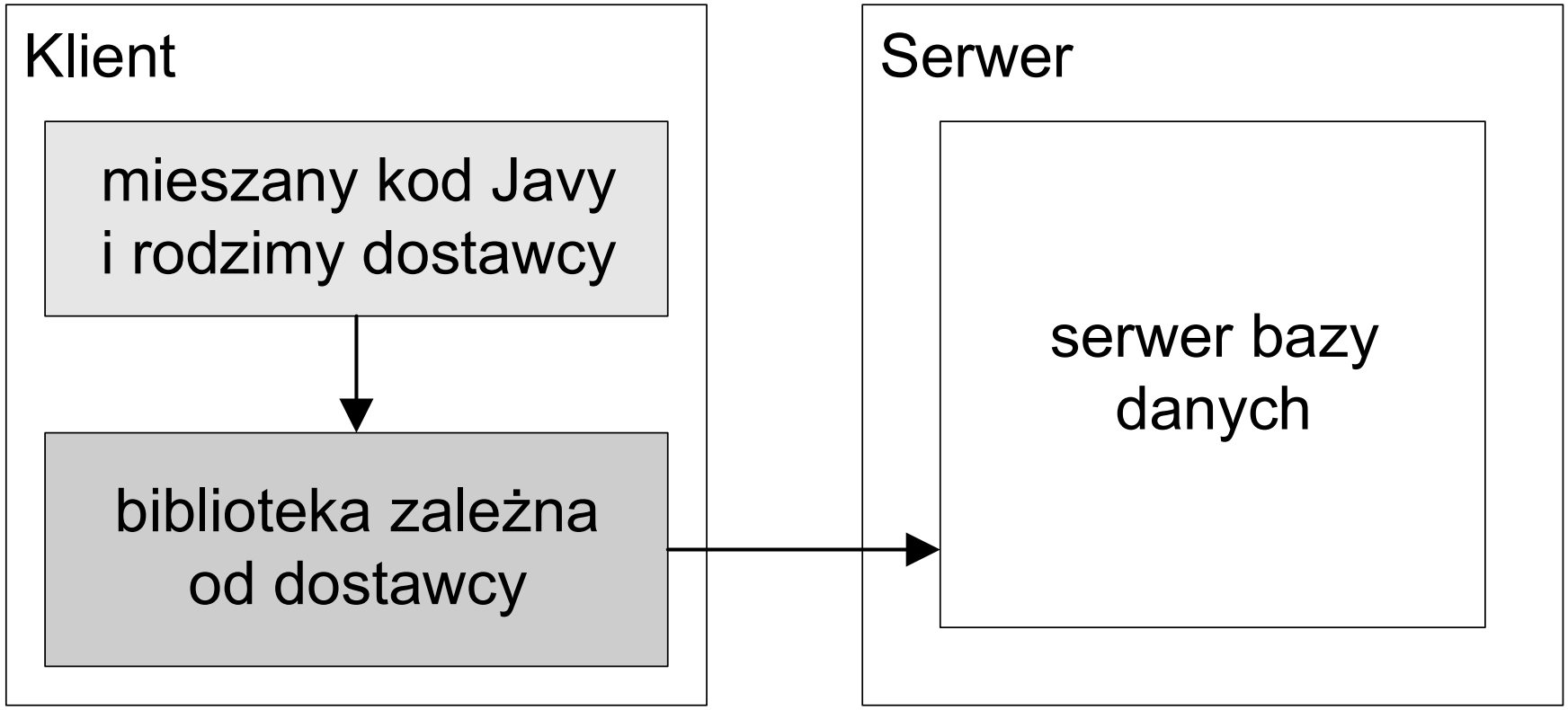
Umożliwia on połączenie z każdą bazą danych, dla której istnieje sterownik ODBC.



Ten sterownik ma znaczenie historyczne i był używany przed pojawieniem się dedykowanych sterowników JDBC.

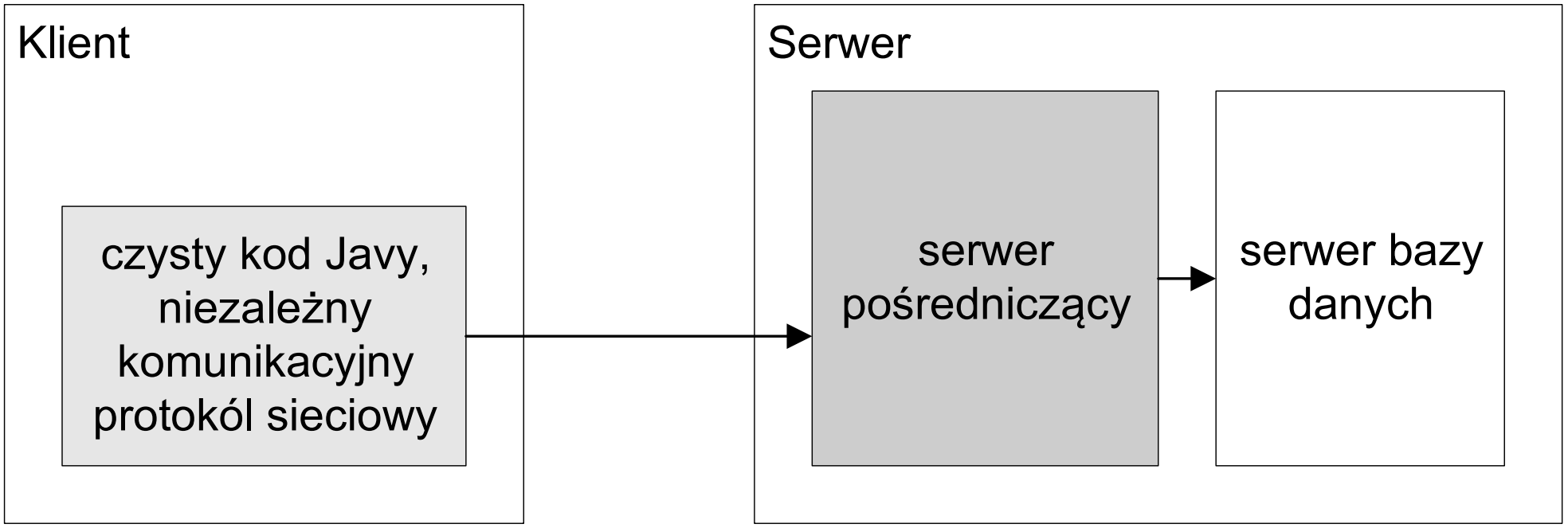
Typ II sterownika JDBC

sterownik napisany w Javie, ale wymagający biblioteki klienta bazy danych najczęściej napisanej w innym języku.



Jest to efektywne rozwiązanie, ale wymaga preinstalowanego oprogramowania klienta bazy danych.

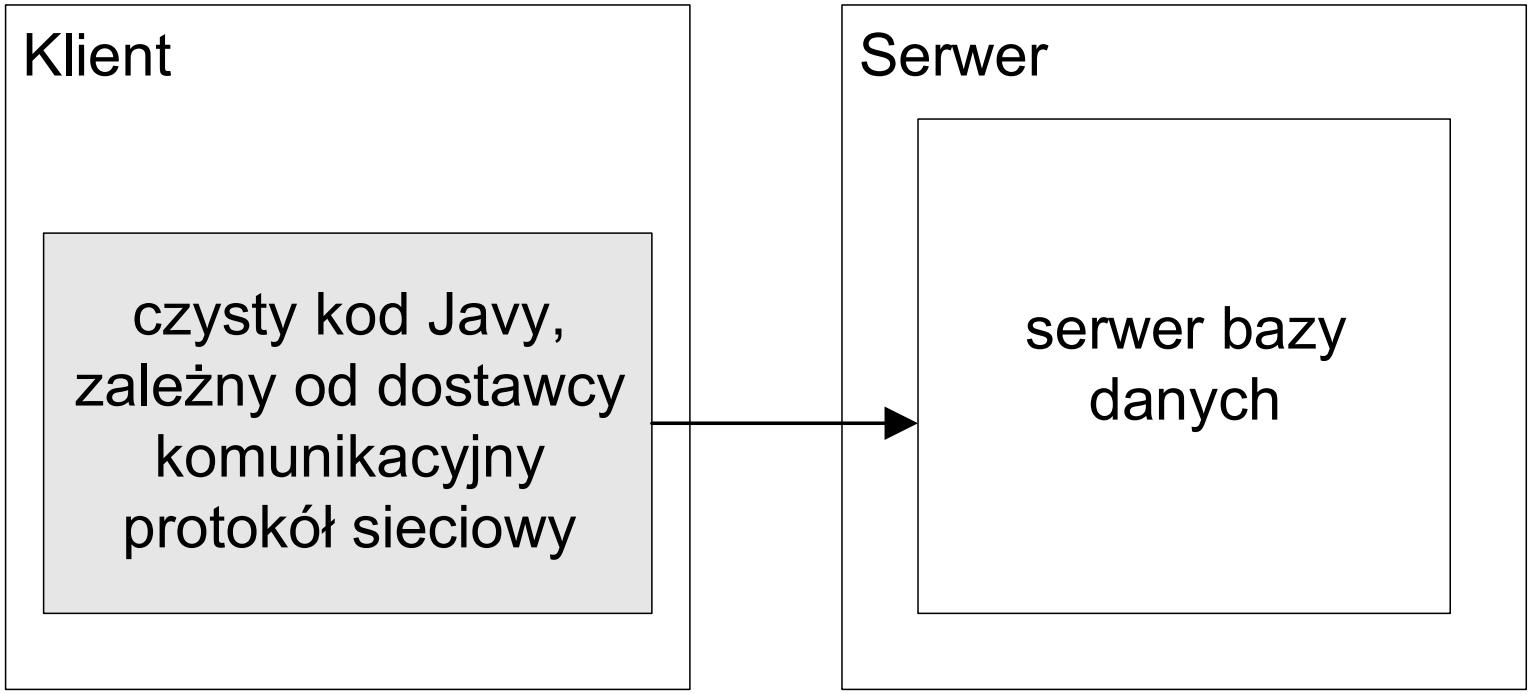
Typ III sterownika JDBC



… z uniwersalnym sterownikiem napisanym w czystej Javie i obsługą poszczególnych rodzajów baz danych w warstwie pośredniej.

Typ IV sterownika JDBC

sterownik napisany w całości w Javie, komunikujący się bezpośrednio z serwerem bazy danych.



W początkach technologii Java to rozwiązanie było uważane za mniej efektywne niż sterownik typu II, ale dziś gdy dostępne są efektywne maszyny wirtualne Java, argument ten stracił na znaczeniu i sterowniki typu czwartego są powszechnie używane.

Standard Query Language (SQL)

Standardowy język dostępu do wszystkich relacyjnych baz danych.
Składa się z:

DML
Data Manipulation Language

INSERT – instruction use to adds rows to a table.
INSERT into students values ('Smith', 'Anna', '21')
SELECT – select data from table
SELECT * from students WHERE age > 20
DELETE - removes a specified row
DELETE * from students WHERE
UPDATE - modifies an existing row
UPDATE students set grade = 5 where age < 22

DDL
Data Definition Language

CREATE TABLE – allows to create a new table in database
TRUNCATE – cleans whole table
DROP TABLE – removes table from database
ALTER TABLE – modifies existing table

Korzystanie z JDBC

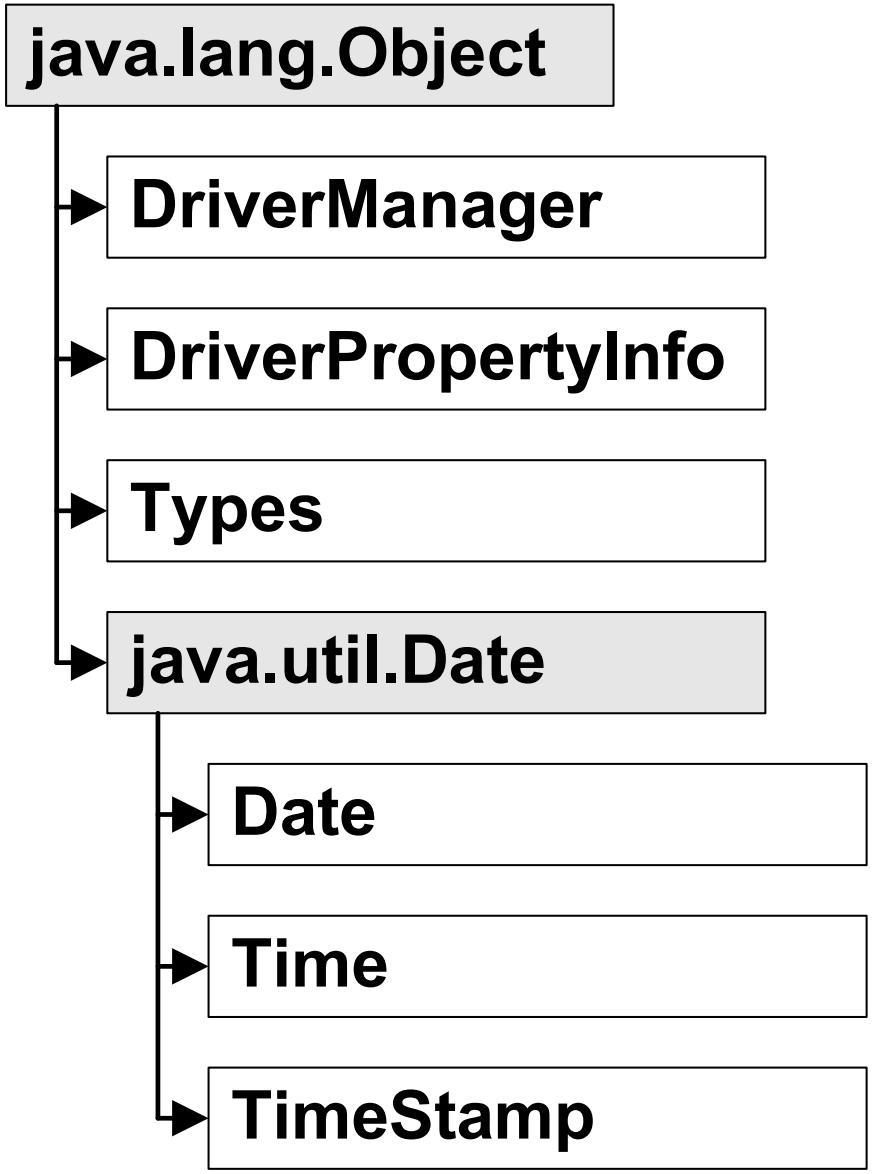
Typowy scenariusz korzystania z JDBC opiera się na trzech krokach.

Nawiązanie połączenia z bazą danych

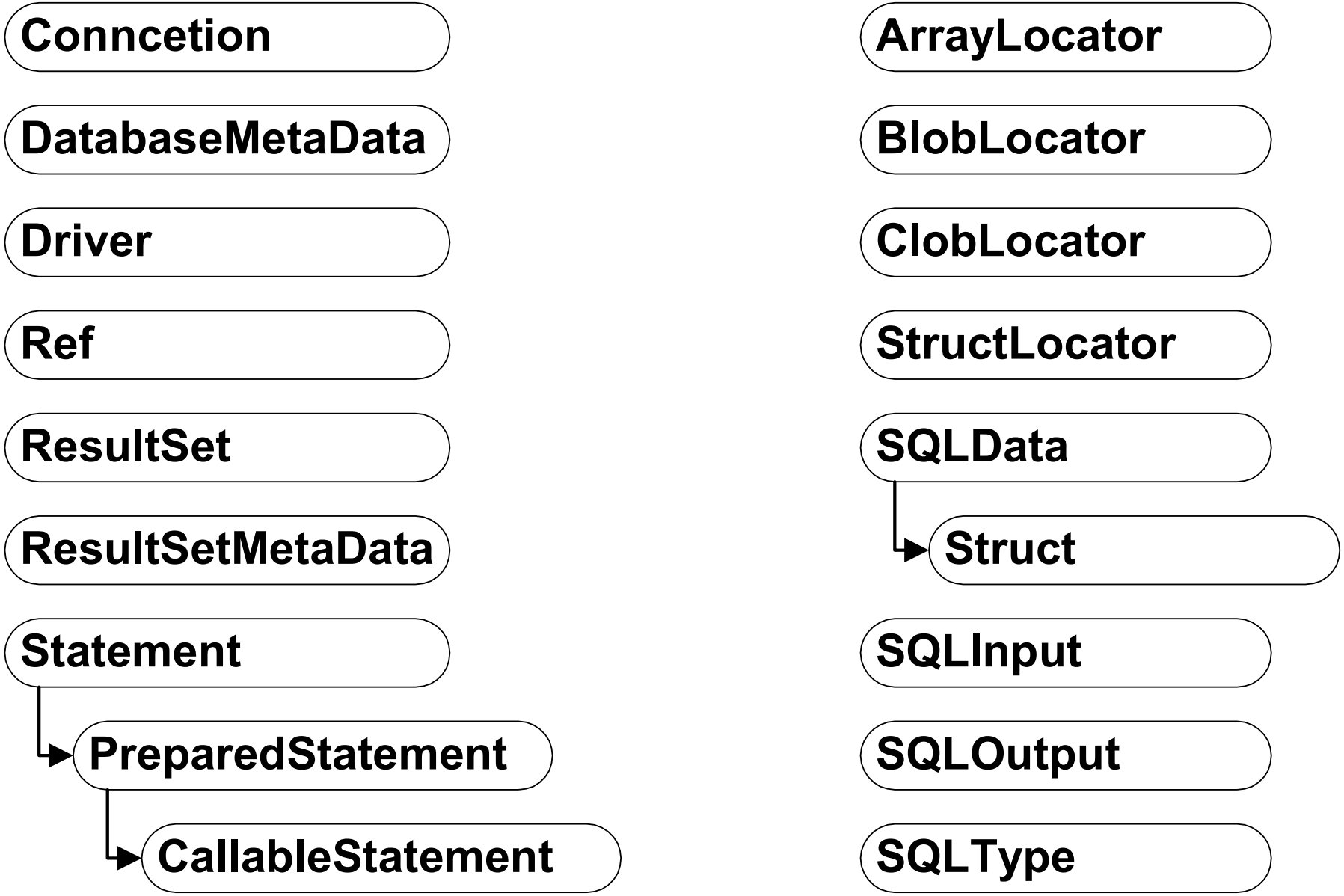
Wysyłanie zapytania, ew. polecenia zmiany, dodania lub usunięcia informacji z bazy.

Opracowanie wyników.

Pakiet java.sql, klasy



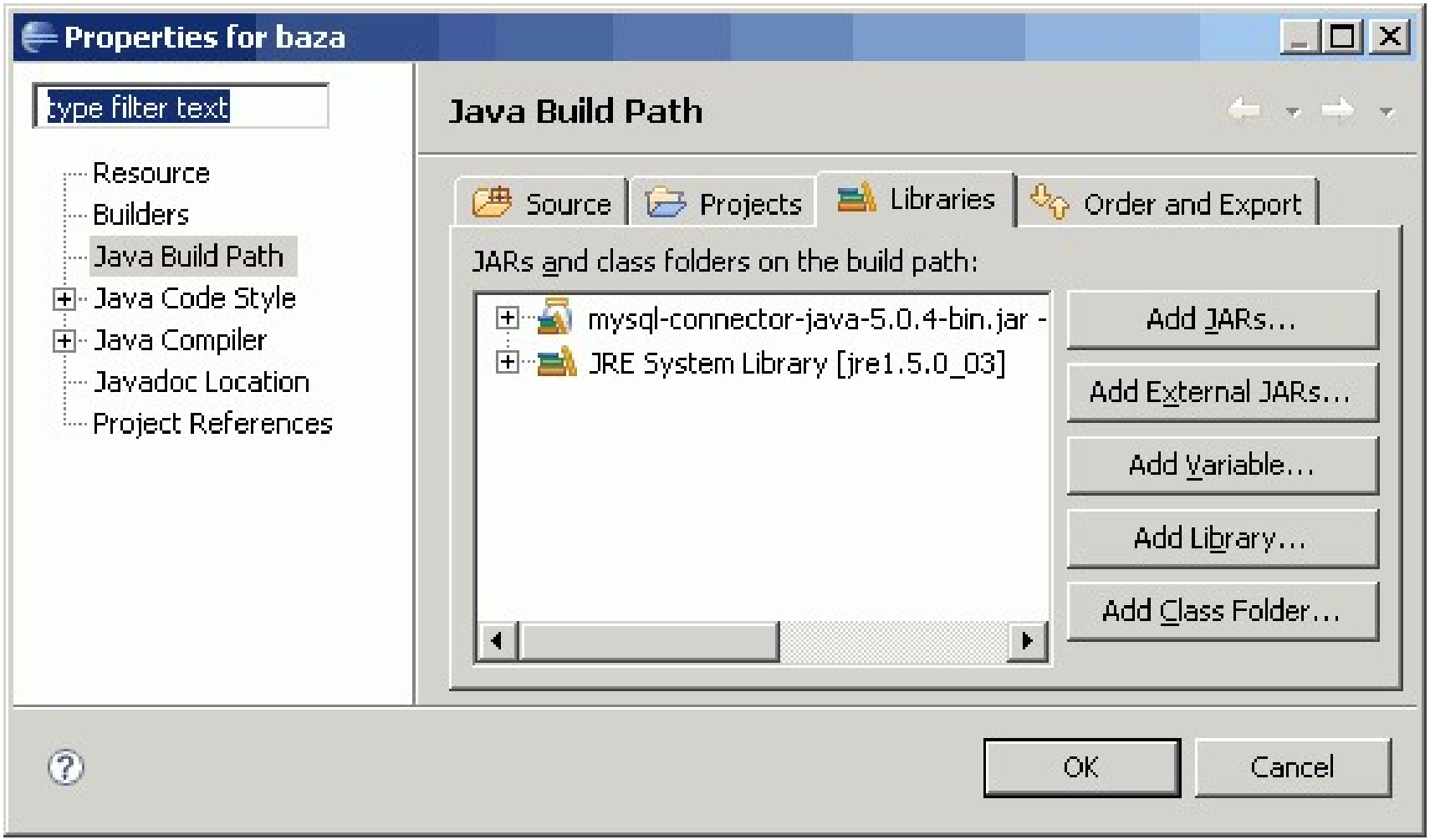
Pakiet java.sql, interfejsy



Uruchomienie JDBC

Potrzebny będzie odpowiedni konektor (zależnie od typu bazy danych)

Do MySQL użyjemy: **mysql-connector-java-5.0.4-bin.jar**



Plik można wgrać również do katalogu jre\lib\ext

Uruchomienie JDBC

```
import java.sql.*;
public class baza {
  public static void main(String[] args)
  {
    try
    {
      Class.forName("com.mysql.jdbc.Driver");
    }
    catch (ClassNotFoundException e)
    {
      e.printStackTrace();
      return;
    }
  }
}
```

W przypadku korzystania z Oracle piszemy:

`Class.forName("oracle.jdbc.driver.OracleDriver");`

Interbase:

`Class.forName("interbase.interclient.Driver");`

Połączenie z bazą danych

```
Connection conn = DriverManager.getConnection(
url,
uzytkownik,
haslo);
```

url: specyficzny dla bazy danych ciąg wskazujący bazę, np.:

Dla Oracle:

```
jdbc:oracle:thin:@nazwa_hosta:1521:nazwa_bazy
```

Dla MySQL

```
jdbc:mysql://nazwa_hosta:3306/nazwa_bazy
```

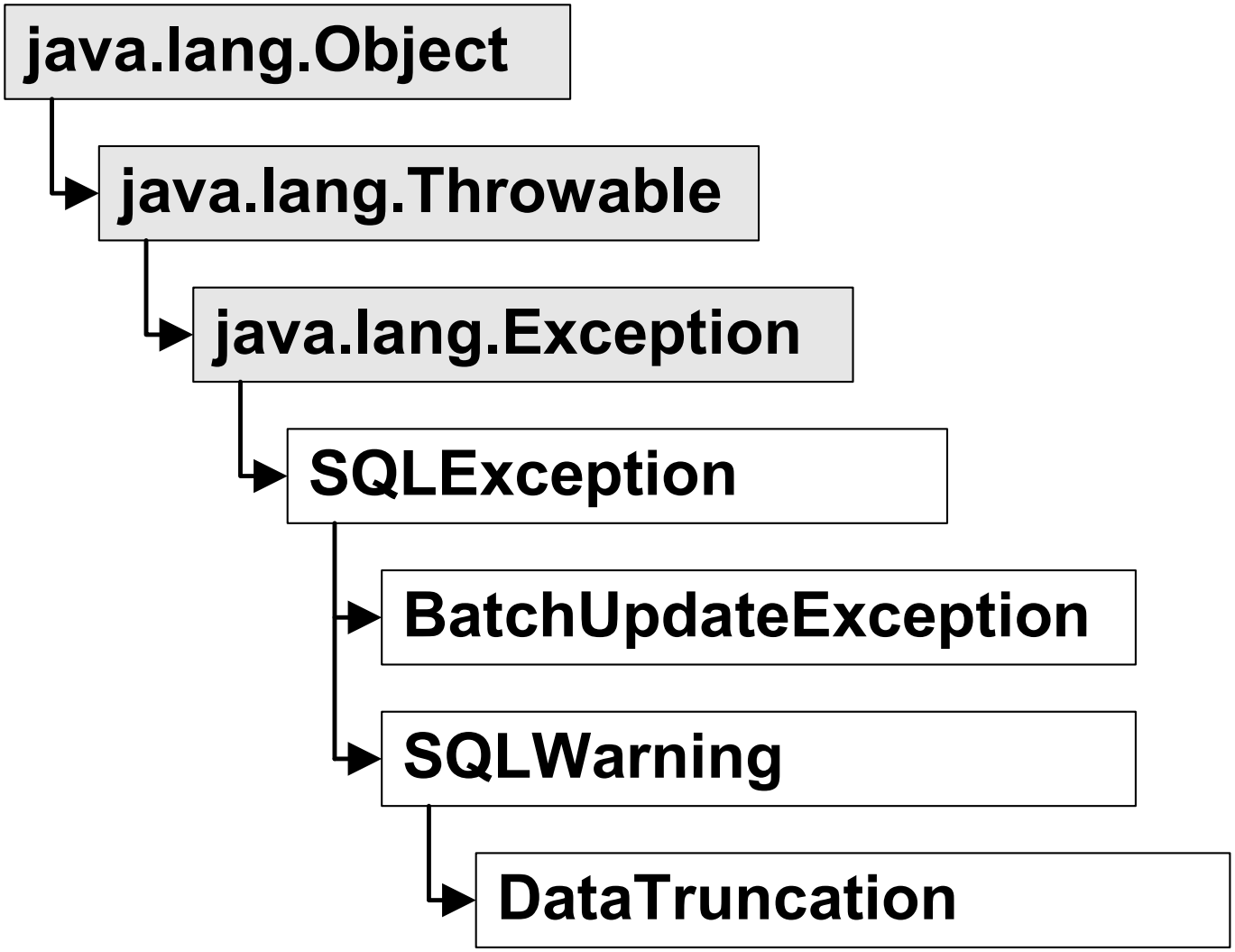
Uwaga: getConnection może wywołać wyjątek SQLException. Trzeba go obsłużyć

Łączenie z bazą

```
try
{
  Class.forName("com.mysql.jdbc.Driver");
  Connection conn = DriverManager.getConnection(
"jdbc:mysql://localhost:3306/mysql", "root","root");
  System.out.println("Połączony!!");
}
catch (SQLException e)
{
//brak połączenia z bazą
}
catch (ClassNotFoundException e)
{
//nie załadowano konektora
}
```

Próbujemy się połączyć z serwerem MySQL, z bazą mysql, przy pomocy użytkownika root

Pakiet java.sql, wyjątki



Wykonywanie zapytań

Tworzenie obiektu zapytania

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

Wykonanie zapytania

```
ResultSet rs = stmt.executeQuery("Zapytanie wybierające");
```

lub

```
stmt.executeUpdate("Zapytanie manipulujące");
```

Prezentacja wyników

Obiekt *ResultSet* to tzw. zbiór rekordów, struktura dynamiczna, posiadająca m.in. funkcje:

first() – przejście do pierwszego rekordu

last() – przejście do ostatniego rekordu

next() – przejście do następnego rekordu

previous() – przejście do poprzedniego rekordu

getFloat("nazwa kolumny") – pobranie wartości float

getInt("nazwa kolumny") – pobranie wartości integer

getDate("nazwa kolumny") – pobranie wartości typu Date

getString("nazwa kolumny") – pobranie wartości String

Wyświetlenie użytkowników bazy MySQL

```
Statement stmt;
try
{
  Class.forName("com.mysql.jdbc.Driver");
  Connection conn = DriverManager.getConnection(
"jdbc:mysql://localhost:3306/mysql", "root","");
  System.out.println("connected");
  stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery("SELECT * FROM user;");
  while (rs.next()) {
    String s = rs.getString("User");
    System.out.println(s);
  }
catch (SQLException e) {
//brak połączenia z bazą
}
catch (ClassNotFoundException e) {
//brak konektora
}
```

Parametryzacja zapytań

```
public static int addPerson(Connection con, Person p) throws
SQLException {
  PreparedStatement ps = null;
  try{
    ps = con.prepareStatement("INSERT INTO PERSONS (PESEL, IMIE,
NAZWISKO, WIEK, DATA_ZAPISU) VALUES (?,?,?,?,?)");
    ps.setString(1, p.getPesel());
    ps.setString(2, p.getImie());
    ps.setString(3, p.getNazwisko());
    ps.setInt(4, p.getWiek());
    ps.setDate(5, p.getDataZapisu());
    return ps.executeUpdate();
  } finally
  {
    if (ps != null)
      ps.close();
  }
}
```

Zamykanie połączenia

wykorzystać metodę klasy **Connection**:

public void close() throws SQLException

Jeśli planuje się za jednym razem wykonać kilka operacji na bazie, wskazane jest wykorzystać do tego celu jedną instancję obiektu **Connection**. Nawiązanie ponownego połączenia z bazą jest dość kosztowne.

Obiekty **Statement** i **ResultSet** też powinny być zamykane (posiadają własne metody **close()**

Transakcje w JDBC

Transakcja składa się z jednego lub więcej poleceń, które zostały wywołane, wykonane i potwierdzone (Commit) lub odrzucone (RollBack).

Transakcja stanowi więc jednostkową operację.

Zarządzanie transakcją jest szczególnie ważne gdy chcemy wykonać naraz kilka operacji traktując je od strony logicznej jako jedną.

Transakcje w JDBC

Domyślnie dla połączenia włączony jest tryb *auto commit*

Po wyłączeniu *auto commit* jawne kończenie transakcji metodami Connection.**commit()** lub **rollback()**

```
Connection conn = DriverManager.getConnection(…);
conn.setAutoCommit(false);
…
…
conn.commit();
…
…
conn.rollback();
```

Nie ma w JDBC wyróżnionej metody rozpoczynającej transakcję. Pierwsze polecenie SQL wykonane po wyłączeniu trybu „auto commit” lub zakończeniu poprzedniej transakcji rozpoczyna nową transakcję.

CZĘŚĆ 2
HIBERNATE

Hibernate – mapowanie baz danych

Jest to technologia pozwalająca mapować dane obiektowe na odpowiadające im struktury w bazach danych.

ORM - Object-to-Relational Mapping.

Jest odpowiedzią na znikomą ilość obiektowych baz danych

Hibernate pozwala na dostęp do relacyjnej bazy danych z poziomu języka Java.

cechy

open source

intuicyjne mapowanie tabel na klasy (za pomocą plików XML).
Są to tzw POJO (Plain Old Java Objects)

Konfiguracja połączenia z bazą za pomocą pliku XML.
Plik zawiera również polecenia DDL do tworzenia bazy i tabel.

mniejsza ilość kodu

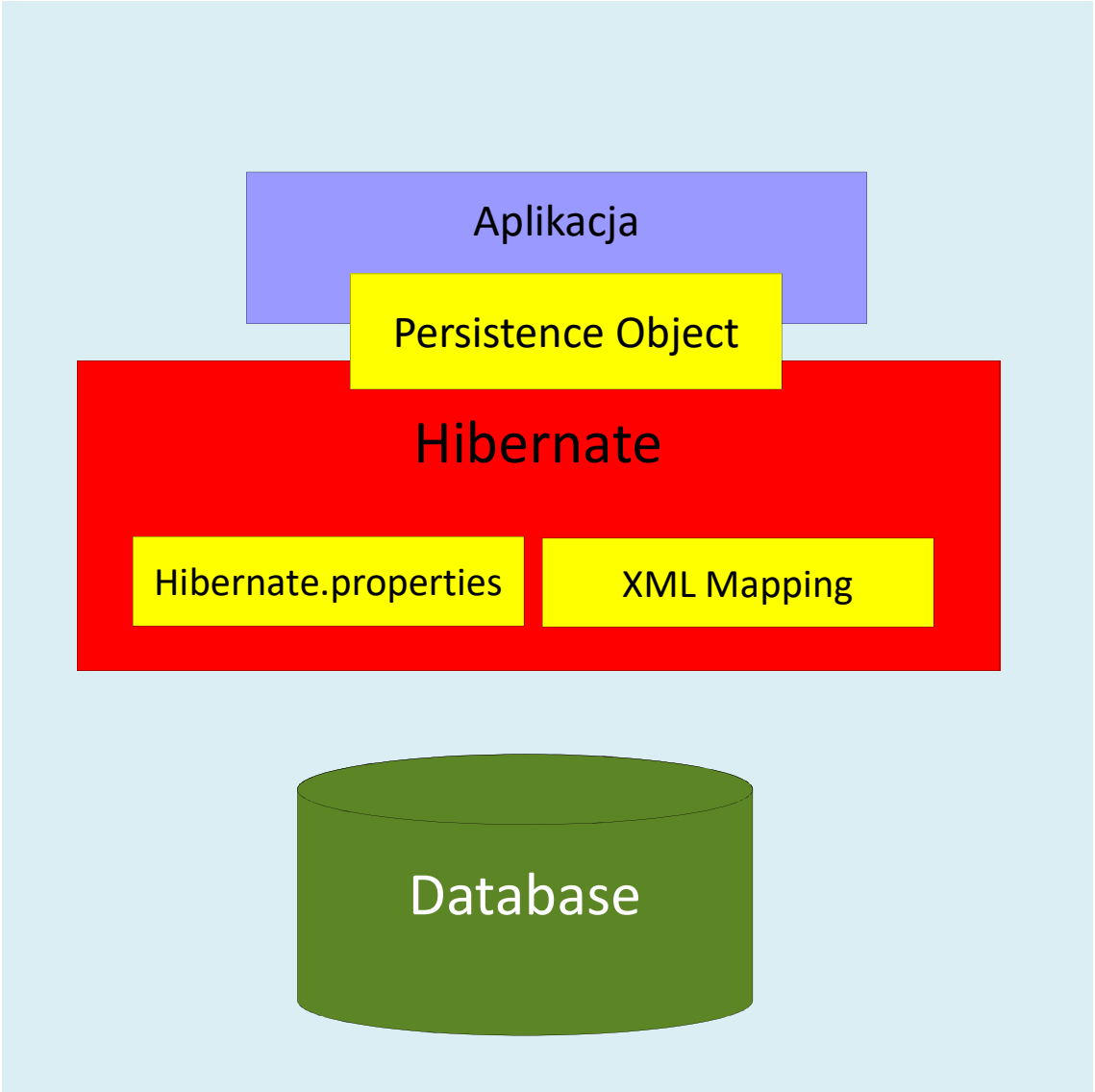
prostota

możliwość korzystania z języka SQL

HQL, Criteria. Korzystanie ze struktury obiektowej (serializacja obiektu) zamiast z zapytań i kolekcji wynikowych

narzędzia wspomagające (np. Hibernate tools)

Architektura Hibernate



Architektura Hibernate

Występują trzy główne komponenty

Connection Management

Serwis zarządzający połączeniem z bazą danych

Transaction management

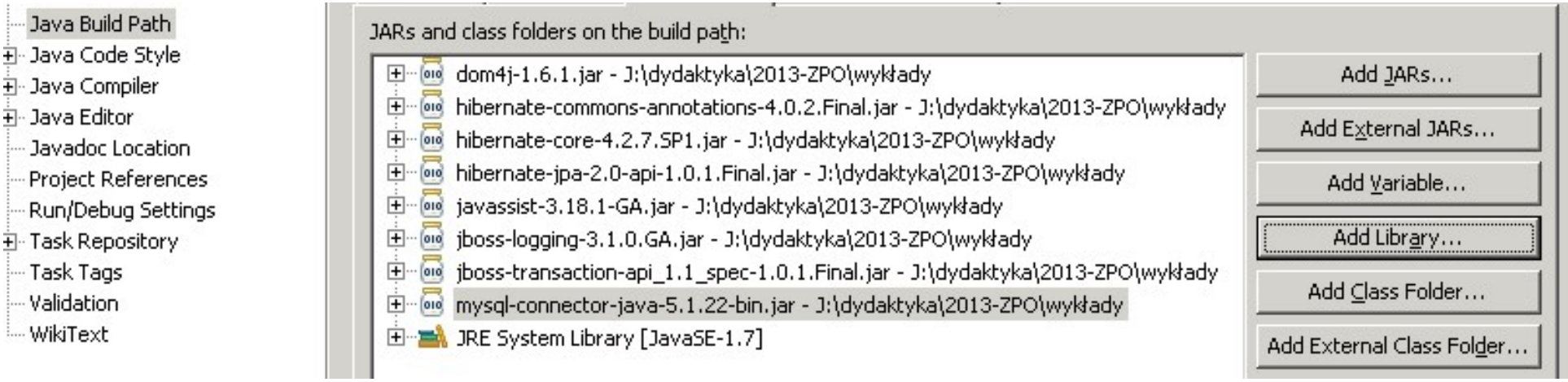
Umożliwia zarządzanie transakcjami czyli wykonywanie grup operacji na bazie

Object relational mapping

Technika mapowania danych pomiędzy modelem obiektowym a relacyjnym.

Tworzymy projekt Hibernate

Pobierz hibernate ze strony http://www.hibernate.org/downloads.html i znajdź oraz zaimportuj potrzebne biblioteki. Również potrzebny jest konektor do bazy



Pozostała część projektu jest zupełnie standardowa

Zastosowanie komponentów i plików konfiguracyjnych XML

Hibernate wykorzystuje metodę konfigurowania komponentów poprzez pliki konfiguracyjne XML.

Mechanizm ten jest szerzej stosowany w Java Spring

W Javie komponentem, ang. Bean (ziarno) jest obiekt klasy, wyposażony w konstruktory, oraz funkcje dostępowe: akcesory (gettery) i mutatory (settery) ewentualnie zdarzenia.

Komponent może zawierać w sobie inne komponenty (pola obiektowe) tworząc skomplikowaną strukturę

Komponenty są wiec od siebie zależne. Rozwiązywanie zależności jest elementem programowania komponentowego.

Konfiguracja i budowa komponentów często jest realizowana poprzez plik XML.

hibernate.cfg.xml

Plik ten definiuje konfigurację komponentu Configuration wykorzystanego przy budowie komponentu Session. Zawiera informację o połączeniu z bazą danych.

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD
3.0//EN" "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="connection.url">jdbc:mysql://localhost:3306/test</property>
<property name="connection.username">root</property>
<property name="connection.password"></property>
<property name="dialect">org.hibernate.dialect.MySQLDialect</property>
<property name="show_sql">true</property>
<property name="hbm2ddl.auto">create</property>
<mapping resource="test/Student.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

Plik hibernate.cfg.xml

Pola w pliku konfiguracyjnym są dosyć intuicyjne. Najważniejsze to podanie:

sterownika

adresu bazy

użytkownika i hasła

dialektu

klas mapujących (pliki *.hbm.xml)

show_sql – pozwala wypisać komendy na konsolę

Dodatkowe opcje:
<property name="hbm2ddl.auto">create</property>
- przy starcie aplikacji tworzy strukturę bazy danych

- update – aktualizuje bazę danych

Konfiguracja Hibernate

Ważne pojęcia:

Configuration.**configure()** – pierwszy krok, odnalezienie pliku konfiguracyjnego (plik pobierany z zasobów)

SessionFactory – fabryka sesji pozwala na otwarcie sesji. Najczęściej wywołuje ją się w bloku statycznym. W całej aplikacji wywołuje się ją raz za pomocą metody: Configuration.**buildSessionFactory()**

Session – sesja, pewna jednostka pracy

Transaction – jeszcze mniejsza jednostka pracy
(musi być zainicjalizowana sesja)

Konfiguracja i praca z Hibernate

```
// 1. Inicjalizacja Hibernate
Configuration cfg = new Configuration().configure();

// 2. Utworzenie fabryki sesji Hibernate
SessionFactory factory = cfg.buildSessionFactory();

// 3. Otwarcie sesji Hibernate
Session session = factory.openSession();

// 4. Rozpoczęcie transakcji
Transaction tx = session.beginTransaction();
//Operacje wykonywane podczas transakcji

// 5. Zatwierdzenie transakcji
    tx.commit();


// 6. Zamknięcie sesji Hibernate
session.close();
```

Mapowanie (*hbm.xml)

Mapowanie polega na odwzorowaniu obiektu utrwalanego (ziarna) na dany rekord w bazie (krotka) W danym przykładzie będzie to Student.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="test">
  <class name="Student" table="Student">
    <id name="id"><generator class="native"/></id>
    <property name="imie" column="IMIE" length="10" not-null="true"/>
    <property name="nazwisko" column="NAZWISKO" length="25" not-null="true"/>
    <property name="ocena" column="OCENA"  not-null="false"/>
    <property name="nrAlbumu" column="NRALBUMU"  not-null="true"/>
    <property name="stacjonarny" column="STACJONARNY"  not-null="true"/>
  </class>
</hibernate-mapping>
```

Hibernate – mapowanie cd.

```
<hibernate-mapping package="test">
  <class name="Student" table="Student">
    <id name="id"><generator class="native"/></id>
    <property name="imie" column="IMIE" length="10" not-null="true"/>
    <property name="nazwisko" column="NAZWISKO" length="25" not-null="true"/>
    <property name="ocena" column="OCENA"  not-null="false"/>
    <property name="nrAlbumu" column="NRALBUMU"  not-null="true"/>
    <property name="stacjonarny" column="STACJONARNY"  not-null="true"/>
  </class>
</hibernate-mapping>
```

Znacznik class zawiera m. in. atrybuty **name**, który odpowiada klasie oraz table określający nazwę tabeli. Pomiędzy znacznikiem **class** znajduje się właściwe mapowanie pól klasy na odpowiadające im kolumny w bazie danych.

Wymaganym znacznikiem jest **id**, który odpowiada za klucz główny, znaczniki property odpowiadają poszczególnym kolumną.

Atrybuty znacznika property pozwalają na przykład określić rozmiar pola, możliwość przyjmowania wartości nullowych, typ złącznia itp.

mapowanie – ziarno

```
public class Student {
	long id;
	String imie;
	String nazwisko;
	int nrAlbumu;
	float ocena;
	boolean stacjonarny;
//...
```

Ponadto klasa musi zawierać:

- Gettery i Settery dla wszystkich pól
- Konstruktor domyślny
- Konstruktor inicjujący

Przykład – zapis obiektu do bazy

```
Configuration cfg = new Configuration().configure();
SessionFactory factory = cfg.buildSessionFactory();
Session session = factory.openSession();
Transaction tx = session.beginTransaction();
Student s =new
Student("Adam","Nowak",123423,3.5f,true);
session.save(s);
tx.commit();
session.close();
```

HQL

HQL jest obiektowym odpowiednikiem języka SQL rozumiejącym takie aspekty jak dziedziczenie czy polimorfizm.

HQL posługuje się nazwami klas zamiast nazwami tabel (relacji)

HQL posługuje się nazwami pól zamiast nazw kolumn tabeli

HQL jest wrażliwy na rozmiar liter tylko jeśli chodzi o nazwy klas bądź pól, w pozostałych przypadkach SeLeCt == SELECT

Przykład:

```
SELECT AVG(us.wiek), SUM(us.wiek), MAX(us.wiek), COUNT(us)
FROM User AS us WHERE us.imie='Jan'
```

Przykłady zapytań HQL

Przykładowe zapytanie SELECT zwróci listę obiektów, dla których wiek jest równy 24

```
Query query =
session.createQuery("select * from Student where age = :age ");
query.setParameter("age", "24");
List list = query.list();
```

Zapytanie UPDATE aktualizuje wiek na 25 dla osób o nazwisku Smith

```
Query query = session.createQuery("update Student set age =
:age" + " where lastName = :lastName");
query.setParameter("age", 25);
query.setParameter("lastName", "Smith");
int result = query.executeUpdate();
```

Zapytanie DELETE usuwa wszystkie osoby o nazwisku Smith

```
Query query =
session.createQuery("delete Student where lastName =:lastName");
query.setParameter("lastName", "Smith");
int result = query.executeUpdate();
```

Hibernate – Criteria

Criteria pozwalają w prosty oraz intuicyjny sposób ograniczyć liczbę zwracanych rekordów. Nie potrzeba znać dokładnie struktury języka zapytań. Wynik zapytania jest zwrócony w postaci listy. Aby ją utworzyć należy w odpowiedniej metodzie podać nazwę klasy (np. User)

```
Criteria crit = sess.createCriteria(User.class);
crit.setMaxResults(50);
List users = crit.list();
```

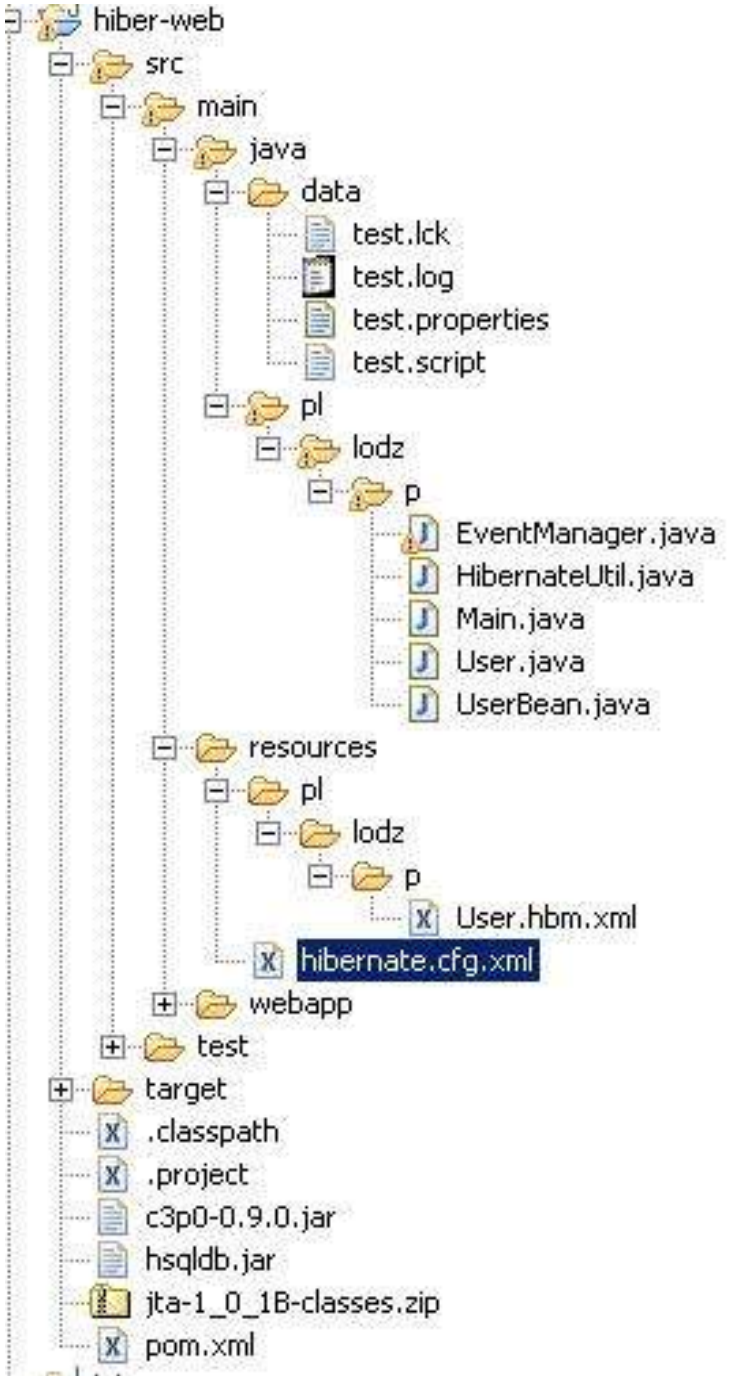
```
List users = sess.createCriteria(User.class)
.add( Restrictions.like("nazwisko", "Now%") )
.add( Restrictions.between("wiek", minWiek, maxWiek)).list();
```

Hibernate - SQL

Hibernate pozwala na korzystanie również z „czystego” języka zapytań SQL.

```
List users = sess.createSQLQuery(
"SELECT {us.*} FROM user us WHERE us.wiek = 17").list();
```

Hibernate – przykład zaawansowany



Być może będziesz potrzebował dodać biblioteki Hibernate:

hibernate-core-4.2.7.SP1.jar
hibernate-commons-annotations-4.0.2.Final.jar
dom4j-1.6.1.jar
jboss-logging-3.1.0.GA.jar
jboss-transaction-api_1.1_spec-1.0.1.Final.jar
hibernate-jpa-2.0-api-1.0.1.Final.jar
antlr-2.7.7.jar
javassist-3.18.1-GA.jar

Oczywiście potrzebujesz też odpowiedni driver ODBC

Hibernate – przykład zaawansowany

W tym przykładzie użyjemy silnika bazy HSQL DB

W katalogu projektu należy utworzyć na poziomie katalogu pakietu podkatalog „data” i tam uruchomić linię poleceń cmd

Używając polecenia
java -classpath lib/hsqldb.jar org.hsqldb.Server
Uruchomić bazę danych.

Uruchomienie aplikacji po raz pierwszy spowoduje utworzenie bazy danych.
Po pierwszym uruchomieniu zaznacz komentarzem linię w pliku hibernate.cfg.xml

<property name="hbm2ddl.auto">create</property>

… lub zmień wartość na „update”

Każda zmiana struktury bazy wymaga przywrócenia poprzedniego stanu w pliku hibernate.cfg.xml

HibernateUtil.java

Plik służy do zarządzania sesją. Otwiera i zamyka fabrykę połączeń.

```
package pl.lodz.p;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;


public final class HibernateUtil {
  private HibernateUtil() {}
  private static final SessionFactory SESSION_FACTORY;
  static {
    try {
      SESSION_FACTORY = new
Configuration().configure().buildSessionFactory();
    } catch (Throwable ex) {/* init error*/ }
  }
  public static final ThreadLocal SESSION = new
ThreadLocal();
```

HibernateUtil.java

```
public static Session currentSession() throws
HibernateException {
    Session s = (Session)SESSION.get();
    if (s == null) {
      s = SESSION_FACTORY.openSession();
      SESSION.set(s);
    }
    return s;
  }
public static void closeSession() throws
HibernateException {
    Session s = (Session)SESSION.get();
    SESSION.set(null);
    if (s != null) s.close();
  }
}
```

EventManager.java

Tworzenie połączenia z bazą SQL

```
package pl.lodz.p;
import java.io.Serializable;
import java.util.*;
import org.hibernate.*;
import pl.lodz.p.User;
public class EventManager {
```

EventManager.java

Wstawianie nowego rekordu

```
public long insertRecord(Object obj) throws Exception {
    long id = -1;
    Transaction tx = null;
    try {

      Session session = HibernateUtil.currentSession();

      tx = session.beginTransaction();

      session.save( obj );

      Serializable ser = session.getIdentifier(obj);
      if (ser != null)  id = Long.parseLong( ser.toString() );

      tx.commit();
      System.out.println("-> end insertRecord");

    } catch (Exception ex) {
      if (tx != null)  tx.rollback();
      id = -1;
      throw ex;
    } finally HibernateUtil.closeSession();
    return id;
  }
```

EventManager.java

Pobranie obiektu wg id

```
public Object selectRecord(Class c, String id, boolean useTx)
throws Exception {
    Object obj = null;
    Transaction tx = null;
    try {
      System.out.println("- Start selectRecord");

      Session session = HibernateUtil.currentSession();
      if (useTx) tx = session.beginTransaction();
      obj = session.get(c, new String(id));
      if (useTx)  tx.commit();
      System.out.println("- end selectRecord");

   } catch (Exception ex) {
      if (tx != null) tx.rollback();
      throw ex;
   } finally {     HibernateUtil.closeSession(); }
   return obj;
}
```

EventManager.java

Pobranie listy rekordów wg zapytania HQL

```
public List selectRecords(String query, boolean useTx) throws
Exception {
  List list = new ArrayList();
  Transaction tx = null;
  String methodName = "selectRecords";
   try {
     Session session = HibernateUtil.currentSession();
     if (useTx) {  tx = session.beginTransaction(); }
     Query q = session.createQuery( query );
     list = q.list();
     if (useTx) {  tx.commit();  }
     System.out.println("- end selectRecords
 list.size="+list.size());
   } catch (Exception ex) {
     if (tx != null) {  tx.rollback();  }
     throw ex;
   } finally {  HibernateUtil.closeSession();  }
   return list;
  }
 }
```

User.java

Klasa mapowana

Musi być w jakimś pakiecie

Musi posiadać funkcje dostępowe

```
private long id;
private String imie;
private String nazwisko;
```

User.hbm.xml – klasa mapująca

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-
3.0.dtd">

<hibernate-mapping>
  <class name="pl.lodz.p.User" table="USER">
    <id name="id" column="USER_ID">
      <generator class="native"/>
    </id>
    <property name="imie"/>
    <property name="nazwisko"/>
  </class>
</hibernate-mapping>
```

hibernate.cfg.xml – plik konfiguracyjny

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD
3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="hibernate.bytecode.use_reflection_optimizer">false</property>
<property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
<property name="connection.url">jdbc:hsqldb:hsql://localhost</property>
<property name="connection.username">sa</property>
<property name="connection.password"></property>
<property name="connection.pool_size">10</property>
<property name="dialect">org.hibernate.dialect.HSQLDialect</property>
<property name="current_session_context_class">thread</property>
<property
name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
<property name="show_sql">true</property>
<property name="hbm2ddl.auto">create</property>
<mapping resource="pl/lodz/p/User.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

CZĘŚĆ 3
JPA

Wprowadzenie

Java Persistence API (JPA) jest kolejnym przykładem mapowania relacyjno-obiektowego (ORM). Jest uogólnieniem Hibernate.
W tym wykładzie zaprezentowano EclipseLink.

JPA pozawala mapować klasę na relację a zatem umieszczać w bazie, aktualizować i pobierać z bazy obiekty.

Proces ten nazywa się **utrwalaniem obiektów** (*pesisting of objects*)
Implementacja JPA nazywana jest **zarządca trwałości** (*persistence provider*).

JPA można używać w aplikacjach Java-EE i Java-SE

Mapowanie jest zrealizowane poprzez metadane.
Metadane mogą być zdefiniowane poprzez plik XML (poprzedni przykład w Hibernate) lub poprzez zastosowanie adnotacji (ewentualnie kombinację obu metod).

W JPA zdefiniowano język zapytań podobny do SQL (HQL).

JPA pozwala na automatyczne tworzenie schematu bazy danych

Encja

Utrwalana klasa powinna mieć adnotację
javax.persistence.Entity.
Taka klasa nazywa się encją (*entity*)

Instancje tej klasy będą rekordami tabeli w bazie danych

Definiowanie encji wymaga:

-Zadeklarowania pola które będzie kluczem głównym
- zdefiniowania konstruktora domyślnego
-Klasa nie może być finalna

JPA pozwala automatycznie tworzyć klucz główny poprzez adnotację
@GeneratedValue

Domyślnie nazwa klasy encji jest jednocześnie nazwą tabeli. Można zmienić to przyporządkowanie poprzez adnotację
@Table(name="NEWTABLENAME").

Encja – utrwalanie pól

Wybrane lub wszystkie pola Encji mogą być utrwalane w bazie.

JPA używa wartości pól Encji lub odpowiednich funkcji dostępowych
Nie wolno mieszać tych dwóch rozwiązań
Użycie funkcji dostępowych wymaga zastosowanie konwencji nazw JavaBeans

JPA domyślnie utrwala wszystkie pola, chyba, że są opatrzone adnotacją
@Transient

Domyślnie każde pole jest mapowane na kolumnę o tej samej nazwie.
Można to zmienić poprzez adnotację
@Column(name="newColumnName")

@Id	Określa klucz główny tabeli
@GeneratedValue	Razem z kluczem określa, że ta wartość będzie generowania automatycznie
@Transient	Pole nie będzie zapisywane w bazie

Mapowanie relacji

JPA pozwala na definiowanie relacji pomiędzy klasami (kompozycja klas). Relacje mogą być następujących typów:

-one-to-one – jeden do jednego – adnotacja ***@OneToOne***
-one-to-many – jeden do wielu – adnotacja ***@OneToMany***
-many-to-one – wiele do jednego – adnotacja ***@ManyToOne***
-many-to-many – wiele do wielu – adnotacja ***@ManyToMany***.

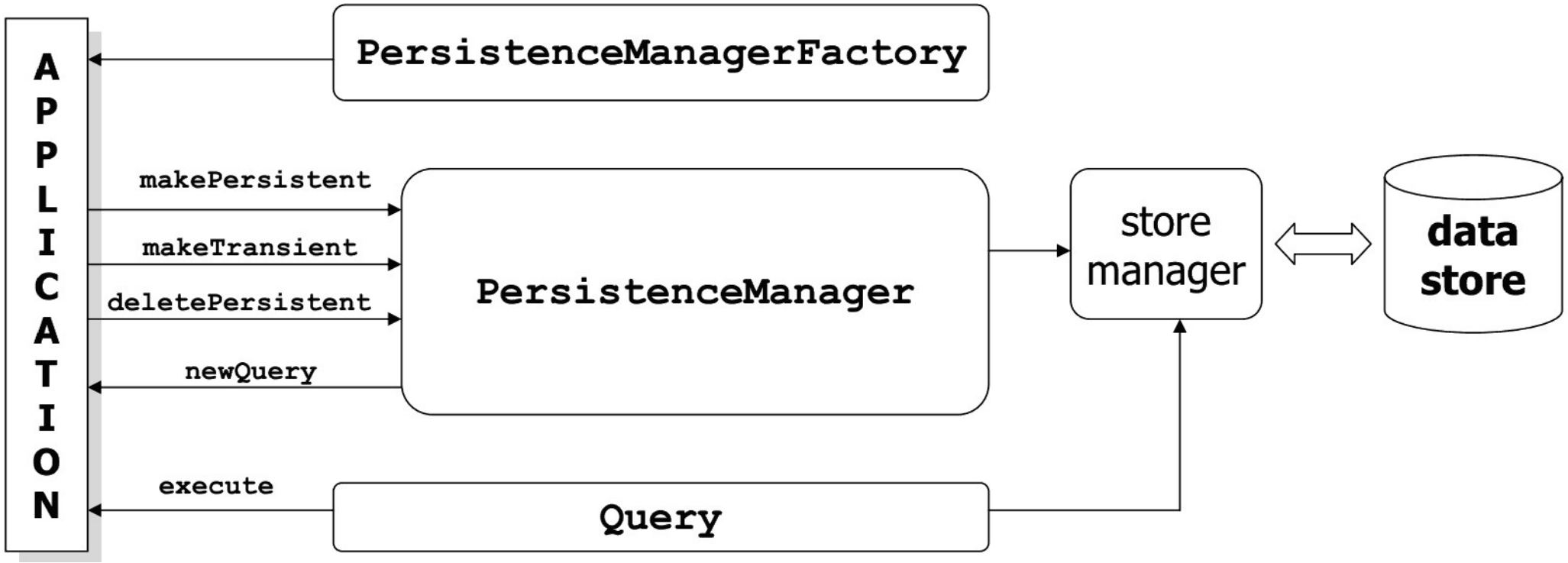
Relacja może być jednokierunkowa lub wielokierunkowa.
W relacji dwukierunkowej obie zależne klasy trzymają do siebie referencje.
W relacji jednokierunkowej jedna klasa przechowuje referencje do drugiej.

W przypadku relacji dwukierunkowej należy zdefiniować właściciela relacji poprzez adnotację
@ManyToMany(mappedBy="attributeOfTheOwningClass").

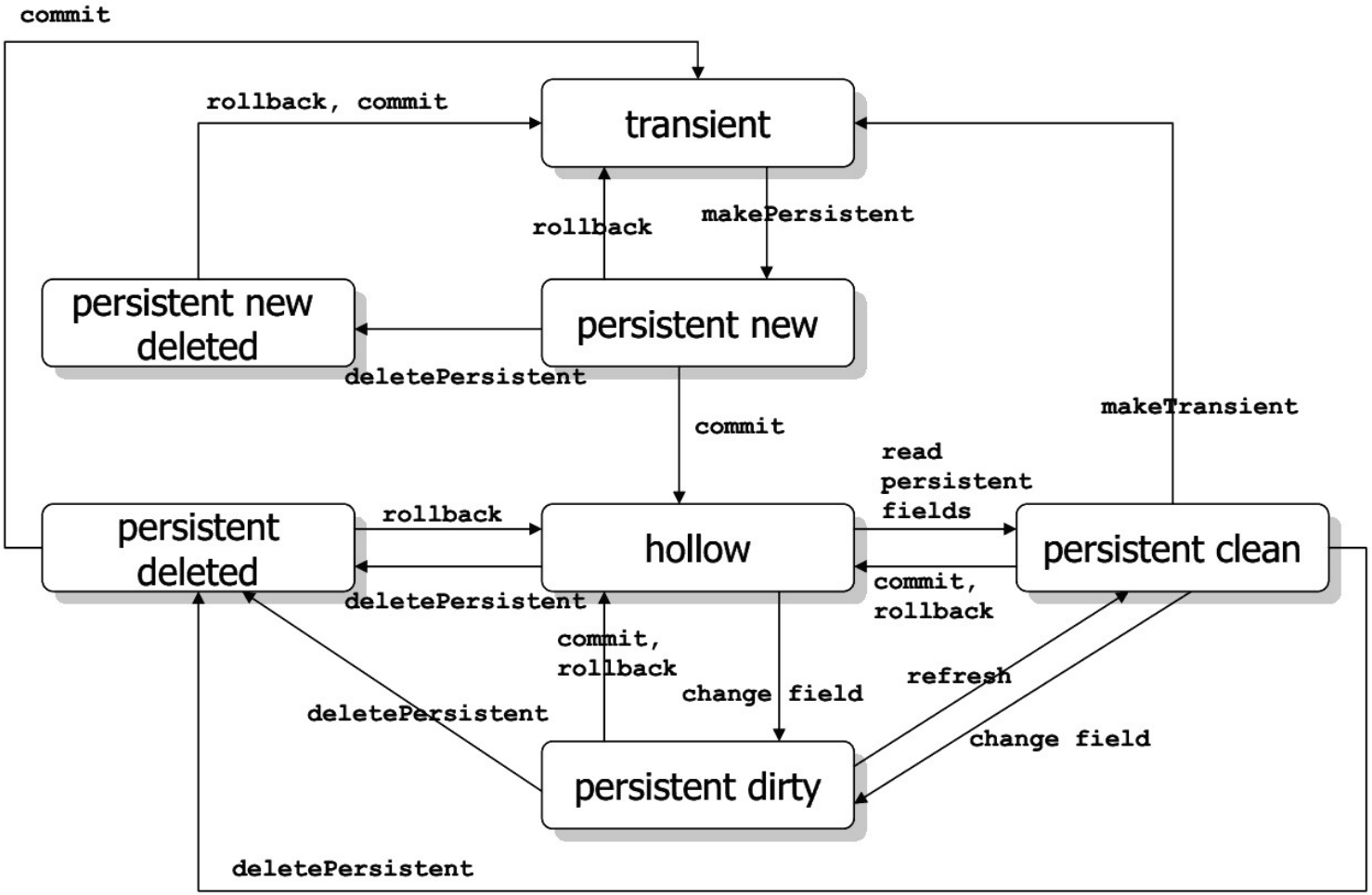
Interfejsy używane przez JPA

PersistenceManagerFactory: podstawowy obiekt służący do uzyskiwania dostępu do zarządcy trwałości
- inicjalizacja na poziomie aplikacji
- inicjalizacja za pomocą parametrów w pliku własności

PersistenceManager: podstawowy obiekt służący do utrwalania obiektów, odczytywania trwałych obiektów ze składnicy danych, usuwania obiektów ze składnicy danych



Stany obiektu



Zarządzanie obiektem

```
EntityManagerFactory  factory =
Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
EntityManager em = factory.createEntityManager();
```

Utrwalenie obiektu

```
Employee e = new Employee(...);
pm.makePersistent(e)
```

Usunięcie obiektu

```
Employee e = pm.getObjectById(id);
pm.deletePersistent(e);
```

Odłączenie obiektu

```
Employee ee = pm.detachCopy(e);
```

Dołączenie obiektu

```
Employee eee = pm.makePersistent(ee);
```

Przykład odwzorowania klasy na tabelę

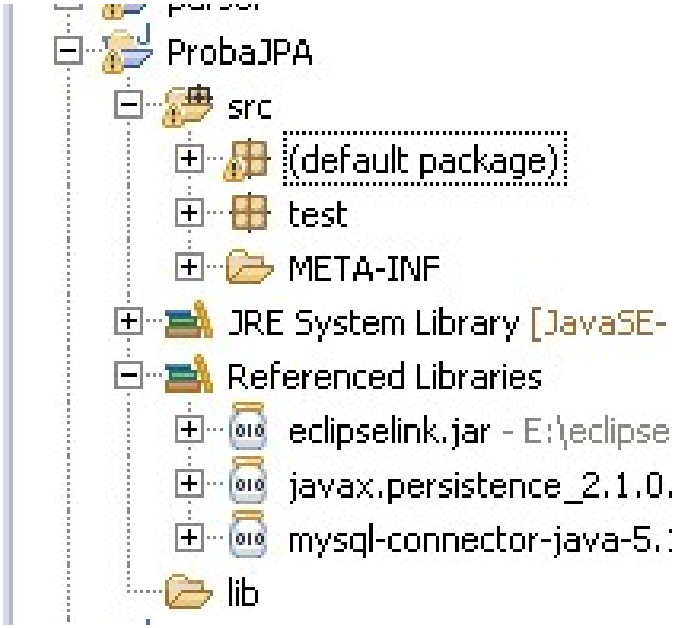
Tworzenie Projektu

Pobrać sterownik bazy danych JDBC, np. **mysql-connector-java-5.1.22-bin.jar**

Ze strony eclipse.org pobrać **EclipseLink**.
Będziemy potrzebować plików

- **eclipselink.jar**
- **javax.persistence_*.jar**

Utworzyć nowy projekt i dodać do niego te biblioteki.
Katalogu src utworzyć katalog META-INF.
Utworzyć też dowolny pakiet dla encji



Przykład odwzorowania klasy na tabelę

Tworzenie encji – w utworzonym pakiecie

```
package test;
import javax.persistence.*;

@Entity
public class Notatka {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String tytul;
  private String opis;
  public String getTytul()            { return tytul;       }
  public void setTytul(String tytul) { this.tytul = tytul;}
  public String getOpis()             { return opis;        }
  public void setOpis(String opis)    { this.opis = opis;   }

  @Override
  public String toString() {  return "Notatka " + tytul + ": " + opis;  }

}
```

Przykład odwzorowania klasy na tabelę

Tworzenie Persistence Unit – w katalogu META-INF tworzymy plik persistence.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="Notatki" transaction-type="RESOURCE_LOCAL">
    <class>test.Notatka</class>
    <properties>
      <property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver" />
      <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost:3306/test" />
      <property name="javax.persistence.jdbc.user" value="root" />
      <property name="javax.persistence.jdbc.password" value="" />
      <!-- EclipseLink should create the database schema automatically -->
      <property name="eclipselink.ddl-generation" value="create-tables" />
      <property name="eclipselink.ddl-generation.output-mode"
        value="database" />
    </properties>
  </persistence-unit>
</persistence>
```

Przykład odwzorowania klasy na tabelę

Klasa testowa

```
import java.util.List;
import javax.persistence.*;
import test.Notatka;
public class Main {
  public static void main(String[] args) {
    EntityManagerFactory factory =
Persistence.createEntityManagerFactory("Notatki");
    EntityManager em = factory.createEntityManager();
// czytamy zawartość tabeli
    Query q = em.createQuery("select t from Notatka t");
    List<Notatka> todoList = q.getResultList();
    for (Notatka todo : todoList)  {
      System.out.println(todo);
    }
    System.out.println("Size: " + todoList.size());
// tworzymy notatkę
    em.getTransaction().begin();
    Notatka todo = new Notatka();
    todo.setTytul("Zakupy");
    todo.setOpis("Kupić karpia w Carfour");
    em.persist(todo);
    em.getTransaction().commit();
    em.close();
  }
}
```

Przykład zastosowania relacji – encja Family

```
package test;
import java.util.*;
import javax.persistence.*;
@Entity
public class Family {
  @Id
  @GeneratedValue(strategy = GenerationType.TABLE)
  private int id;
  private String description;

  @OneToMany(mappedBy = "family")
  private final List<Person> members = new ArrayList<Person>();

  public int getId() {  return id;  }
  public void setId(int id) {  this.id = id;  }
  public String getDescription() { return description; }
  public void setDescription(String description)
  {
     this.description = description;
  }
  public List<Person> getMembers() {  return members;  }

}
```

Przykład zastosowania relacji – encja Person

```
package test;
import java.util*;
import javax.persistence.*;
@Entity
public class Person {
  @Id
  @GeneratedValue(strategy = GenerationType.TABLE)
  private String id;
  private String firstName;
  private String lastName;
  private Family family;
  private String nonsenseField = "";
  private List<Job> jobList = new ArrayList<Job>();
  public String getId() {    return id;  }
  public void setId(String Id) {    this.id = Id;  }
  public String getFirstName() {    return firstName;  }
  public void setFirstName(String firstName) {
    this.firstName = firstName;
}
// Leave the standard column name of the table
  public String getLastName() { return lastName; }
  public void setLastName(String lastName) { this.lastName = lastName;
  }
```

Przykład zastosowania relacji – encja Person cd

```
@ManyToOne
  public Family getFamily() { return family; }
  public void setFamily(Family family) { this.family = family; }

@Transient
  public String getNonsenseField() { return nonsenseField;  }
  public void setNonsenseField(String nonsenseField) {
    this.nonsenseField = nonsenseField;
  }

@OneToMany
  public List<Job> getJobList() {
    return this.jobList;
  }

  public void setJobList(List<Job> nickName) {
    this.jobList = nickName;
  }

}
```

Przykład zastosowania relacji – encja Job

```
package test;
import javax.persistence.*;
@Entity
public class Job
{
  @Id
  @GeneratedValue(strategy = GenerationType.TABLE)
  private int id;
  private double salery;
  private String jobDescr;
  public int getId()                          {   return id;  }
  public void setId(int id)                   {  this.id = id;  }
  public double getSalery()                   {  return salery; }
  public void setSalery(double salery)    { this.salery = salery; }
  public String getJobDescr()                 { return jobDescr; }
  public void setJobDescr(String jobDescr){this.jobDescr = jobDescr;}

}
```

Przykład zastosowania relacji – persistence.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="people" transaction-type="RESOURCE_LOCAL">

    <class>test.Person</class>
    <class>test.Family</class>
    <class>test.Job</class>

 <properties>
   <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
   <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/test" />
   <property name="javax.persistence.jdbc.user" value="root" />
   <property name="javax.persistence.jdbc.password" value="" />
<!-- EclipseLink should create the database schema automatically -->
   <property name="eclipselink.ddl-generation" value="update-tables" />
   <property name="eclipselink.ddl-generation.output-mode" value="database" />
 </properties>
</persistence-unit>
</persistence>
```

Przykład zastosowania relacji – klasa testowa

```
import javax.persistence.*;
import test. Family;
import test.Person;

public class Main {
  private EntityManagerFactory factory;
  public void setUp() throws Exception {
    factory = Persistence.createEntityManagerFactory("people");
    EntityManager em = factory.createEntityManager();
    em.getTransaction().begin();  //nowa transakcja
    Query q = em.createQuery("select m from Person m"); //lista rekordów
    boolean createNewEntries = (q.getResultList().size() == 0);
    if (createNewEntries) {// No, so lets create new entries
    Family family = new Family();
    family.setDescription("Family for the Knopfs");
    em.persist(family);
      for (int i = 0; i < 40; i++) {
        Person person = new Person();
        person.setFirstName("Jim_" + i);
        person.setLastName("Knopf_" + i);
        em.persist(person);
        family.getMembers().add(person);
        em.persist(person);
        em.persist(family);
      }
    }
    em.getTransaction().commit();
    em.close();
  }
```

Przykład zastosowania relacji – klasa testowa cd

```
public void checkAvailablePeople()
{
  EntityManager em = factory.createEntityManager();
  Query q = em.createQuery("select m from Person m");
// We should have 40 Persons in the database
    System.out.println(q.getResultList().size());
    em.close();
}

public void checkFamily()
{
    EntityManager em = factory.createEntityManager();
    Query q = em.createQuery("select f from Family f");
// We should have one family with 40 persons
    System.out.println(q.getResultList().size());
    System.out.println(((Family)q.getSingleResult()).getMembers().size());
    em.close();
}
```

Przykład zastosowania relacji – klasa testowa cd

```
public void deletePerson()
{
    EntityManager em = factory.createEntityManager();
    // Begin a new local transaction so that we can persist a new entity
    em.getTransaction().begin();
    Query q = em
        .createQuery("SELECT p FROM Person p WHERE p.firstName = :firstName AND
p.lastName = :lastName");
    q.setParameter("firstName", "Jim_1");
    q.setParameter("lastName", "Knopf_!");
    Person user = (Person) q.getSingleResult();
    em.remove(user);
    em.getTransaction().commit();
    Person person = (Person) q.getSingleResult();
    // Begin a new local transaction so that we can persist a new entity
    em.close();
  }
}
```