# Konfiguracja bazy danych H2

To baza danych przechowywana w pamięci. Resetuje się za każdym razem gdy uruchamiamy nasz program od nowa.

1)dodanie do POM xml odpowiedniej zależności

<!-- https://mvnrepository.com/artifact/com.h2database/h2 -->

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<version>1.0.60</version>

</dependency>

2)podłączenie do bazy danych

**Klasa AppH2**

**private static** String *h2Driver*=**"org.h2.Driver"**;  
**private static** String *h2Address*=**"jdbc:h2:mem:test\_mem;DB\_CLOSE\_DELAY=-1"**;  
*//domyślne wartości dla bazy danych w pamięci***private static** String *user*=**""**;  
**private static** String *password*=**""**;

By default, closing the last connection to a database closes the database. For an in-memory database, this means the content is lost. To keep the database open, add ;DB\_CLOSE\_DELAY=-1 to the database URL. To keep the content of an in-memory database as long as the virtual machine is alive, use jdbc:h2:mem:test;DB\_CLOSE\_DELAY=-1.

**public static** Connection getConnection() **throws** ClassNotFoundException, SQLException {  
 Connection connection=**null**;  
 Class.*forName*(*h2Driver*);  
 connection=DriverManager.*getConnection*(*h2Address*,*user*,*password*);  
 **return** connection;

}

**public static void** main(String[] args) **throws** SQLException, ClassNotFoundException {  
 Connection connection=*getConnection*();  
 **if** (connection!=**null**){  
 System.***out***.println(**"SUKCES"**);  
 }  
}

# JDBC

JDBC polega na przesyłaniu zapytań do bazy.

Będziemy chcieli wrzucić obiekty klasy Student do bazy danych.

@AllArgsConstructor  
@Getter  
@Setter  
@NoArgsConstructor

@ToString  
**public class** Student {  
 **private int id**;  
 **private** String **name**;  
}

Utworzenie tabeli USERS w bazie danych w pamięci (H2)

**private static void** createTableForStudent() **throws** SQLException, ClassNotFoundException {  
 Connection connection=AppH2.*getConnection*();  
 Statement statement=connection.createStatement();  
 String query=**"CREATE TABLE STUDENTS(id int primary key,name varchar(255))"**;  
 statement.execute(query);  
 connection.commit();  
}

Wstawienie do bazy danych obiektów klasy Student

**private static void** insertStudent(Student student) **throws** SQLException, ClassNotFoundException {  
 Connection connection=AppH2.*getConnection*();  
 Statement statement=connection.createStatement();  
 String query=**"INSERT INTO STUDENTS VALUES("**+  
 student.getId()+**",\'"**+student.getName()+**"\')"**;  
 statement.execute(query);  
 connection.commit();  
}

Pobranie studentów

**public static** List<Student> getStudent() **throws** SQLException, ClassNotFoundException {  
 List<Student> students=**new** ArrayList<>();  
 Connection connection=AppH2.*getConnection*();  
 Statement statement=connection.createStatement();  
 String query=**"SELECT** *\** **FROM STUDENTS"**;  
 ResultSet resultSet=statement.executeQuery(query);  
 **while** (resultSet.next()){  
 **int** id=resultSet.getInt(**"id"**);  
 String name=resultSet.getString(**"name"**);  
 students.add(**new** Student(id,name));  
 }  
 **return** students;  
}

Jak widzimy musi tu być **ResultSet resultSet=statement.executeQuery(query);**  
bo odbieramy wyniki

# JPA

Java Persistence API – zestaw interfejsów, które wspomagają programistę w operowaniu na bazie danych. Najpopularniejszą biblioteką implementującą standard JPA jest Hibernate. Operacje są cachowane. JPA decyduje kiedy dane mają być przesłane.

# JPA/Hibernate

Dodajemy zależność w POM.xml

<**dependency**>  
 <**groupId**>org.hibernate</**groupId**>  
 <**artifactId**>hibernate-core</**artifactId**>  
 <**version**>5.2.16.Final</**version**>  
</**dependency**>

Rozwiązanie problemu gdy nie ma folderu resources

<**resources**>  
 <**resource**>  
 <**directory**>src/main/resources</**directory**>  
 </**resource**>  
</**resources**>

Należy to w <build> wpisać

**Konfiguracja JPA**

1)utworzyć katalog **META-INF** w resources a w nim plik **persistent.xml –** standardowy plik konfiguracyjny JPA.

Można również użyć pliku **hibernate.config.xml**

Hibernate jest zwykła biblioteką z której można korzystać bezpośrednio używając klas oraz korzystać z pliku konfiguracyjnego hibernate.config.xml. Jednak gdy chcemy przejść na wyższy poziom abstrakcji to od tego jest JPA i plik konfiguracyjny persistent.xml. Wtedy korzystamy z paczek JPA, a nie z paczek Hibernate. Zaletą jest fakt, że podczas implementacji naszego programu można podmienić bibliotekę Hibernate na inną która również implementuje standard JPA. Jest to zaleta tylko teoretyczna. Nie zdarza się taka sytuacja. W praktyce korzysta się z JPA tam gdzie można.

Przykładowy plik persistence.xml

*<?***xml version="1.0" encoding="UTF-8"** *?>*<**persistence xmlns="http://java.sun.com/xml/ns/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\_1\_0.xsd" version="1.0"**>  
  
 <**persistence-unit name="persystencja" transaction-type="RESOURCE\_LOCAL"**>  
 <**provider**>org.hibernate.jpa.HibernatePersistenceProvider</**provider**>  
  
 <**properties**>  
 <**property name="connection.driver\_class" value="org.h2.Driver"**/>  
 <**property name="hibernate.connection.url" value="jdbc:h2:./db/repository"**/>  
 <**property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"**/>  
 <**property name="hibernate.hbm2ddl.auto" value="create-drop"**/>  
 <**property name="hibernate.show\_sql" value="true"** />  
 </**properties**>  
 </**persistence-unit**>  
</**persistence**>

JPA operuje na dwóch podstawowych obiektach

**EntityManagerFactory** – singleton. Używa się głównie do tworzenia EntityManagera.

**EntityManager** – to nie jest singleton. Odpowiada za jednorazowe połączenie z bazą danych. Komunikuje się bezpośrednio z JPA, a pośrednio z bazą danych.

**public class** AppJPA {  
 **public static void** main(String[] args) {  
 EntityManagerFactory factory= Persistence.*createEntityManagerFactory*(**"persystencja"**);  
 EntityManager manager=factory.createEntityManager();  
 }  
}

# Encja

Encja to obiekt javowy, który posiada pola i metody.

Aby ze Student zrobić encję należy dodać adnotację @Entity i inne zgodnie z tym co poniżej

Należy pamiętać, że Hibernate wymaga pustego konstruktora, nawet prywatnego.

@Entity(name = **"students"**)  
**public class** Student {  
 @Id  
 **private int id**;  
 @Column(name = **"name"**,nullable = **false**)  
 **private** String **name**;  
}

@Transient oznacza, aby tego pola nie mapować.

# Crud

Create, Read, Update, Delete

## Create

W ten sposób można **dodać studenta** do bazy

**public static void** createStudent(){  
 Student student1=**new** Student(15,**"Jan"**);  
 ***manager*.persist(student1);**  
}

Jednak nie zawsze od razu widać wynik zapytania, bo operacje przechowywane **są w cachu**. Aby zmusić JPA do działania

**public static void** createStudent(){  
 Student student1=**new** Student(15,**"Jan"**);  
 ***manager*.getTransaction().begin();**  
 *manager*.persist(student1);  
 ***manager*.getTransaction().commit();**  
}

@@LOGI

INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@70a36a66'

Hibernate: insert into students (name, id) values (?, ?)

@@LOGI

**Transakcja** to wrapper na jakiś algorytm. **Transkacja** musi wykonać cała albo w ogóle.

## Read

Jak **odczytać** Studenta?

**public static void** readStudent(){  
 Student student=***manager*.find**(Student.**class**,15);  
 System.***out***.println(student);  
}

@@LOGI

Hibernate: insert into students (name, id) values (?, ?)

Student(id=15, name=Jan)

@@LOGI

A jeśli chcemy **pobrać więcej studentów?**

Musimy napisać zapytanie i użyć pseudojęzyka JPQL, o którym będzie poniżej

A najlepsza wersja jest taka

**@Entity  
@Table(name = "students")**  
**public class** Student {  
 @Id  
 **private int id**;  
 @Column(name = **"name"**,nullable = **false**)  
 **private** String **name**;  
}

Oraz

**public static void** readStudents(){  
 String query=**"FROM Student"**;  
 List<Student> studenci=*manager*.createQuery(query).getResultList();  
 studenci.forEach(System.***out***::println);  
}

## Update

Aktualizacja danych rekordu (Studenta)

**public static void** updateStudent(**int** id){  
 Student student=*manager*.find(Student.**class**,id);  
 student.setName(**"Tomasz"**);  
 *manager*.getTransaction().begin();  
 ***manager*.merge(student);**  
 *manager*.getTransaction().commit();  
}

merge – jeśli nie ma studenta, to zapisuje go do bazy danych. Merge można użyć jeszcze wtedy gdy chcemy, aby obiekt był zwrócony od razu.

@@LOGI

Student(id=15, name=Jan)

Student(id=11, name=Józef)

Student(id=1, name=Tomasz)

@@LOGI

## Delete

Kasowanie

**public static void** removeStudent(**int** id){  
 Student student=*manager*.find(Student.**class**,id);  
 *manager*.getTransaction().begin();  
 *manager*.remove(student);  
 *manager*.getTransaction().commit();  
}

# Embedded

A co jeśli chcemy w naszej klasie zrobić kompozycję? Dla Embedded nie są tworzone osobne tabele w bazie danych.

**@Embeddable** – klasa ta jest wstawiana do innej tabeli(klasy). I wtedy w klasie w której występuje należy użyć **@Embedded** przed nazwą zmiennej

@AllArgsConstructor  
@NoArgsConstructor  
@Embeddable  
**public class** Address {  
 **private** String **street**;  
 **private** String **postalCode**;  
 **private** String **city**;  
}

@AllArgsConstructor  
@Getter  
@Setter  
@NoArgsConstructor  
@ToString  
@Entity  
@Table(name = **"students"**)

**public class** Student {  
 @Id  
 **private int id**;  
 @Column(name = **"name"**,nullable = **false**)  
 **private** String **name**;  
 @Embedded  
 **private** Address **address**;  
}

# OneToOne

Połączymy tabelę Students z tabelą Indeks

W tabeli Students

@OneToOne  
**private** Indeks **indeks**;

Wtedy w main

*manager*.getTransaction().begin();  
Student jan=**new** Student(15,**"Jan"**); // lub merge  
Indeks indeks=**new** Indeks(1,**"indeks1"**); //lub merge  
jan.setIndeks(indeks);  
*manager*.getTransaction().commit();  
System.***out***.println(jan);

UWAGA: indeks musi być już w bazie danych

To jest relacje 1do1 jednostronna bo student ma indeks, ale Indeks nie ma nic o Studencie

**Jak zrobić w drugą stronę?**

W klasie Indeks

**public class** Indeks {  
 @Id  
 **private int id**;  
 **private** String **number**;  
  **@OneToOne(mappedBy = "indeks")**  
 **private** Student **student**;  
}

i na koniec

@AllArgsConstructor  
@Getter  
@Setter  
@NoArgsConstructor  
@ToString  
@Entity  
@Table(name = **"students"**)  
**public class** Student {  
 @Id  
 **private int id**;  
 @Column(name = **"name"**,nullable = **false**)  
 **private** String **name**;  
 @OneToOne  
 **@ToString.Exclude**  
 **private** Indeks **indeks**;  
 **public** Student(**int** id,String name){  
 **this**.**id**=id;  
 **this**.**name**=name;  
 }  
*// @Embedded  
// private Address address;*}

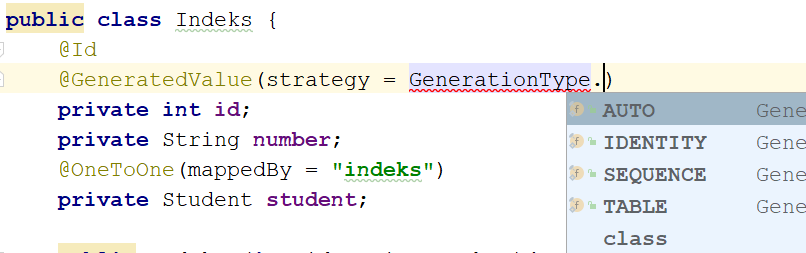
Main

**public static void** main(String[] args) {  
 *manager*.getTransaction().begin();  
 Student jan=*manager*.merge(**new** Student(15,**"Jan"**));  
 Indeks indeks=*manager*.merge(**new** Indeks(1,**"indeks1"**));  
 jan.setIndeks(indeks);  
 jan=*manager*.merge(jan);  
 indeks.setStudent(jan);  
 *manager*.merge(indeks);  
 *manager*.getTransaction().commit();  
 System.***out***.println(jan);  
  
 Indeks ind=*manager*.find(Indeks.**class**,1);  
 System.***out***.println(ind);

}

# Generowanie unikalnego Id

Nie należy ręcznie podawać id. Należy używać odpowiednej adnotacji



AUTO – czyli +1

IDENTITY – baza danych automatycznie dba o to, by ta wartość była unikalna

SEQUENCE – według pewnej sekwencji np. co 2 lub 3.

Podobnie należy zmienić dla klasy Student.

UWAGA: Automatyczna numeracja indeksów zaczyna się od 1 a nie od 0.

Aby pobrać prawidłowo indeks należy użyć

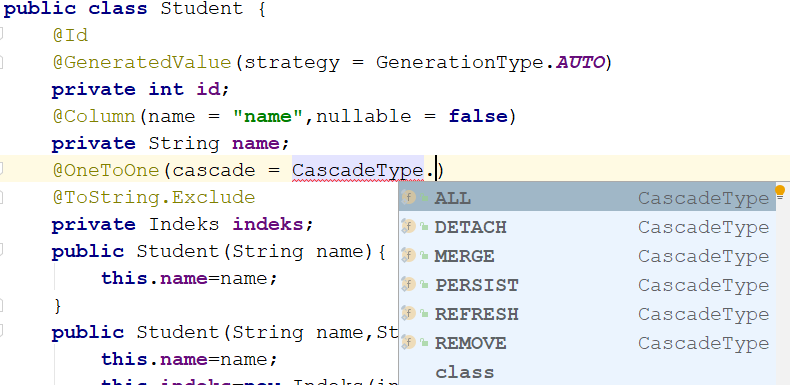
Indeks mergeIndeks=*manager*.merge(indeks);  
Indeks indx=*manager*.find(Indeks.**class**,mergeIndeks.getId());  
System.***out***.println(indx);

Student(id=1, name=Jan)

Indeks(id=2, number=indeks1, student=Student(id=1, name=Jan))

# Operacje kaskadowe

Chcemy za jednym zamachem utworzyć Studenta i dla niego indeks. Musimy użyć operacji kaskadowych aby zapisać Studenta wraz z polem Indeks do bazy danych. Należy to zrobić jednocześnie.



Cascade mówi, że jeśli wykonamy operacje na Student to ma się ona zpropagować w dół do wszystkich obiektów, które są w relacji ze Student

ALL – każdy typ operacji

*manager*.getTransaction().begin();  
Student jan=*manager*.merge(**new** Student(**"Jan"**,**"123456"**));  
Student mergeStudent=*manager*.merge(jan);  
*manager*.getTransaction().commit();  
System.***out***.println(jan);

System.***out***.println(jan.getIndeks());

Student(id=2, name=Jan)

Indeks(id=1, number=123456, student=null)

Null?

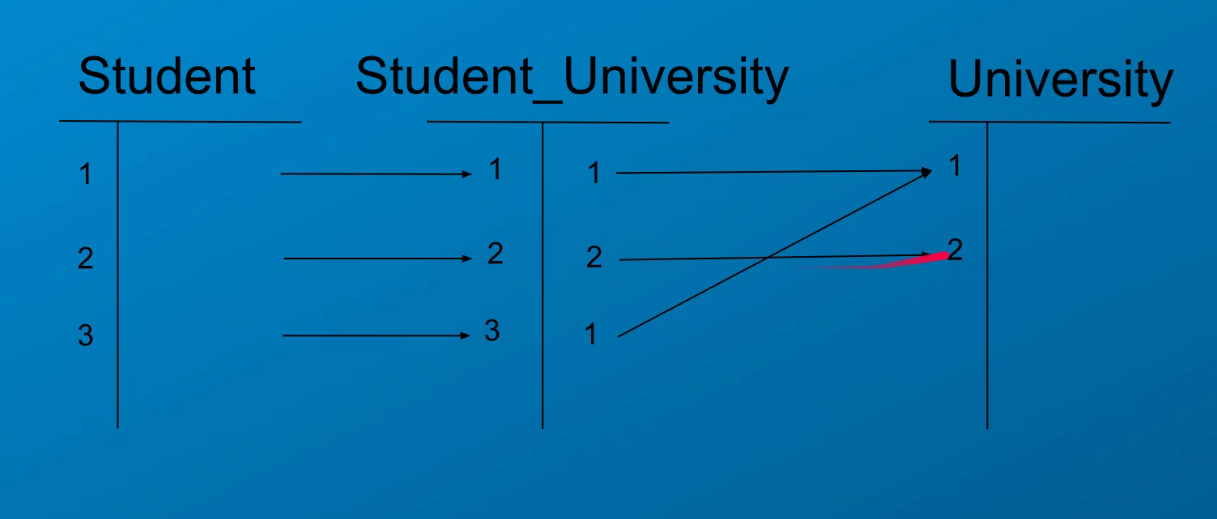
# Jednostronna relacja OneToMany

Zaimplementujemy następującą sytuację, że jeden uniwersytet może mieć wielu studentów

@AllArgsConstructor  
@NoArgsConstructor  
@Getter  
@Setter  
@Entity  
**public class** University {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***AUTO***)  
 **private int id**;  
 **private** String **name**;  
 **@OneToMany**  
 **private** Set<Student> **students**;  
 **public** University(String name) {  
 **this**.**students**=**new** HashSet<>();  
 **this**.**name** = name;  
 }  
 **public void** addStudent(Student student){  
 **students**.add(student);  
 }  
}

*manager*.getTransaction().begin();  
Student jan=*manager*.merge(**new** Student(**"Jan"**,**"123456"**));  
Student mergeStudent=*manager*.merge(jan);  
*manager*.getTransaction().commit();  
  
*manager*.getTransaction().begin();  
University umcs=*manager*.merge(**new** University(**"UMCS"**));  
umcs.addStudent(jan);  
*manager*.getTransaction().commit();

Aby zmapować odpowiednio relacje OneToMany potrzebujemy dodatkowej tabeli, w której będą dwie kolumny iduniwersytetu, idstudenta



# Dwukierunkowa relacja ManyToOne

W relacji ManyToOne nie ma opcji mappedBy więc musimy odwrócić sytuację.

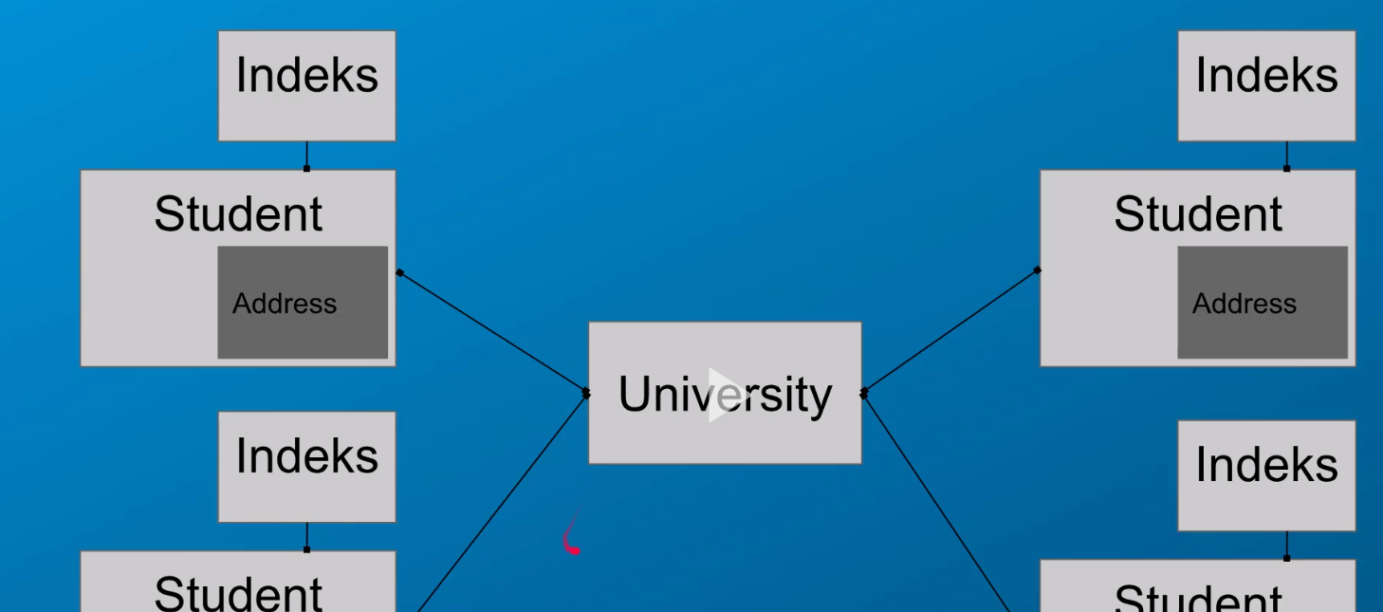
Klasa Student

@ManyToOne  
**private** University **university**;

Klasa University

@OneToMany(mappedBy = **"university"**)  
**private** Set<Student> **students**;

Otrzymamy



Nie ma już dodatkowej tabeli.

Łączymy teraz studenta z konkretnym uniwersytetem, a nie uniwersytet z grupą studentów.

Należy pamiętać, że JPA nie uzupełni za nas obu stron relacji.

**public static void** main(String[] args) {  
 *manager*.getTransaction().begin();  
 Student jan=*manager*.merge(**new** Student(**"Jan"**,**"123456"**));  
 University umcs=*manager*.merge(**new** University(**"UMCS"**));  
 jan.setUniversity(umcs);  
 umcs.addStudent(jan);  
 *manager*.merge(jan);  
 *manager*.merge(umcs);  
 *manager*.getTransaction().commit();  
 University university=*manager*.find(University.**class**,umcs.getId());  
 System.***out***.println(university);  
}

Klasa Student, aby się nie zapętlić

@ManyToOne  
**@ToString.Exclude**  
**private** University **university**;

Podsumowanie: Dwustronna relacja ManyToOne

1. Stronę ManyToOne (Student) trzeba zrobić główną stroną relacji
2. W OneToMany (University) dajemy mapowanie
3. Jesteśmy odpowiedzialny za uzupełnienie obu stron relacji. JPA nie zrobi tego za nas

jan.setUniversity(umcs);  
umcs.addStudent(jan);

# ManyToMany

Wielu studentów uczęszcza na wiele zajęć. Dodamy zatem klasę Classes oraz zmienimy klasę Student.

@AllArgsConstructor  
@NoArgsConstructor  
@Getter  
@Setter  
@Entity  
@Table(name = **"classes"**)  
**public class** Classes {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***AUTO***)  
 **private int id**;  
 **private** String **name**;  
 **private** Set<Student> **students**;  
  
 **public** Classes(String name) {  
 **this**.**students**=**new** HashSet<>();  
 **this**.**name** = name;  
 }  
  
 **public void** addStudent(Student student){  
 **students**.add(student);  
 }  
}

**public class** Student {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***AUTO***)  
 **private int id**;  
 @Column(name = **"name"**,nullable = **false**)  
 **private** String **name**;  
 @OneToOne(cascade = CascadeType.***ALL***)  
 @ToString.Exclude  
 **private** Indeks **indeks**;  
 @ManyToOne  
 @ToString.Exclude  
 **private** University **university**;  
 **private Set<Classes> classes;**  
 **public** Student(String name){  
 **this**.**name**=name;  
 }  
 **public** Student(String name,String indexNumber){  
 **this**.**name**=name;  
 **this**.**indeks**=**new** Indeks(indexNumber);  
 **this.classes=new HashSet<>();**  
 }  
  
 **public void addClasses(Classes classes){  
 this.classes.add(classes);  
 }**

Zaczynamy tworzenie relacji

Classes

@ManyToMany  
**private** Set<Student> **students**;

Student

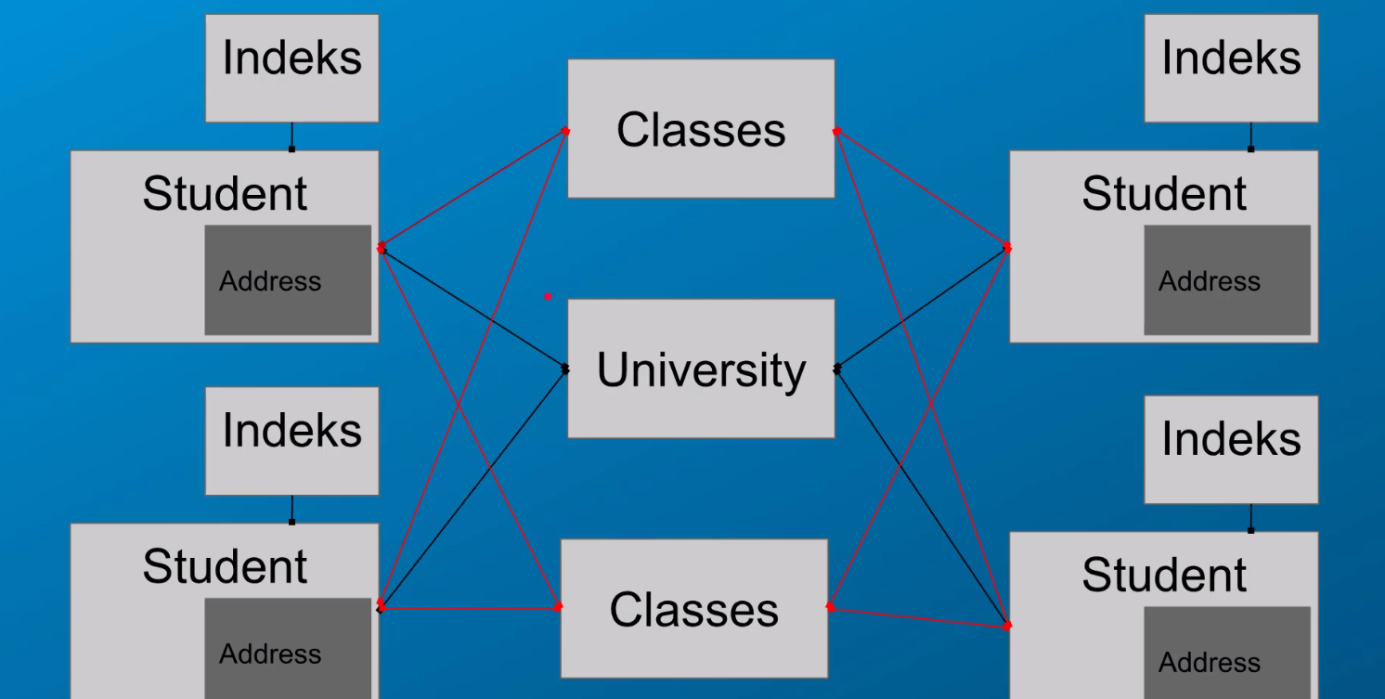
@ManyToMany  
**private** Set<Classes> **classes**;

Wygląda na to (w logach), że mamy dwie relacje 1:N. Wypadało by jednak określić, która strona relacji jest nadrzędna. Będzie to student. A więc w Classes ustawiamy mapowanie.

Classes:

@ManyToMany(mappedBy = **"classes"**)  
**private** Set<Student> **students**;

Mamy w tej chwili encje



# JPQL – podstawy

Służy, podobnie jak SQL, do tworzenia zapytań do bazy danych.

Nie operuje on jednak na tabelach ale na obiektowym modelu encji.

Pobranie wszystkich studentów z bazy

*manager*.createQuery(**"SELECT s FROM Student s"**).getResultList().forEach(System.***out***::println);

lub krócej

*manager*.createQuery(**"FROM Student "**).getResultList().forEach(System.***out***::println);

A tak pobierzemy wszssytkie indeksy do listy

List<Indeks> indeksy=*manager*.createQuery(**"SELECT s.indeks FROM Student s"**,Indeks.**class**).getResultList();

# JPQL – parametryzacja zapytań

1 sposób

TypedQuery<Indeks> query=*manager*.createQuery(**"SELECT s.indeks FROM Student s WHERE s.name=?1"**,Indeks.**class**);  
query.setParameter(**1,"Jan"**);  
System.***out***.println(query.getSingleResult());

2 sposób

TypedQuery<Indeks> query=*manager*.createQuery(**"SELECT s.indeks FROM Student s WHERE s.name=:name"**,Indeks.**class**);  
query.setParameter(**"name"**,**"Jan"**);  
System.***out***.println(query.getSingleResult());

//getSingleResult() zwraca pierwszy wynik zapytania

# JPQL – zwracanie wybranych wartości

Czyli co zrobić gdy chcemy zwrócić z bazy kilka różnych rzeczy.

Query query1=*manager*.createQuery(**"SELECT s,s.indeks FROM Student s"**);  
System.***out***.println(query1.getResultList());

Niestety wyniki są takie:

[[Ljava.lang.Object;@67af833b, [Ljava.lang.Object;@d1f74b8]

Musimy to jakoś sparsować do prawdziwych obiektów.

Najlepiej będzie zrobić klasę, która ma takie pola, jakich spodziewamy się jako wyniku zapytania JPQL.

@ToString  
**public class** QueryResult {  
 **private** String **name**;  
 **private** String **indeksNumber**;  
  
 **public** QueryResult(String name, String indeksNumber) {  
 **this**.**name** = name;  
 **this**.**indeksNumber** = indeksNumber;  
 }  
}

A w main

TypedQuery<QueryResult> query1=*manager*.createQuery(**"SELECT new persystencja.QueryResult(s.name,s.indeks.number) FROM Student s"**,QueryResult.**class**);  
query1.getResultList().forEach(System.***out***::println);

# JPQL – group, having, order

Zadanie:

Policzyć ile jest studentów, których imię jest Paweł albo Jan. Posortować.

@AllArgsConstructor  
@NoArgsConstructor  
@Getter  
@Setter  
@ToString  
**public class** CountResult {  
 **private** String **name**;  
 **private long counter**;  
}

List<CountResult> results=*manager*.createQuery(**"SELECT new persystencja.CountResult(s.name, count(s)) from Student s group by s.name having s.name in ('Paweł','Jan') order by s.name"**,CountResult.**class**).getResultList();  
System.***out***.println(results);

# JPQL – named queries

Co zrobić gdy nasze zapytanie będzie się pojawiało w różnych miejscach w kodzie?

Można je umieścić w stringu, ale najlepszym rozwiązaniem będzie named query – adnotacja.

@Entity  
@NamedQuery(name = **"Student.getAll"**,query = **"SELECT s from Student s"**)  
@Table(name = **"students"**)  
**public class** Student {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***AUTO***)

List<Student> students=*manager*.createNamedQuery(**"Student.getAll"**,Student.**class**).getResultList();  
System.***out***.println(students);

Można również zrobić więcej niż jedno namedQuery()

@NamedQueries(  
 {@NamedQuery(name = **"Student.getAll"**,query = **"SELECT s from Student s"**),  
 @NamedQuery(name = **"Student.get"**,query = **"SELECT s from Student s"**)}  
)  
@Table(name = **"students"**)  
**public class** Student {

# JPQL – Lazy/Eager

**Lazy** nie zaciąga niepotrzebnych w danej chwili danych, a **Eager** przeciwnie - zawsze zaciąga wszystko.

Jeśli chcemy mieć tylko informacje o Uniwersytetach, a nie chcemy ściągać niepotrzebnych danych o studentach:

**private** String **name**;  
@OneToMany(mappedBy = **"university"**,**fetch = FetchType.*LAZY*)**  
**private** Set<Student> **students**;

@ManyToOne , @ManyToMany – domyślny typ LAZY

@OneToOne – domyślny typ EAGER

# MySQL

Do tej pory wszystko co wpisaliśmy do bazy znikało po zamknięciu programu.

<**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**version**>5.1.6</**version**>  
</**dependency**>