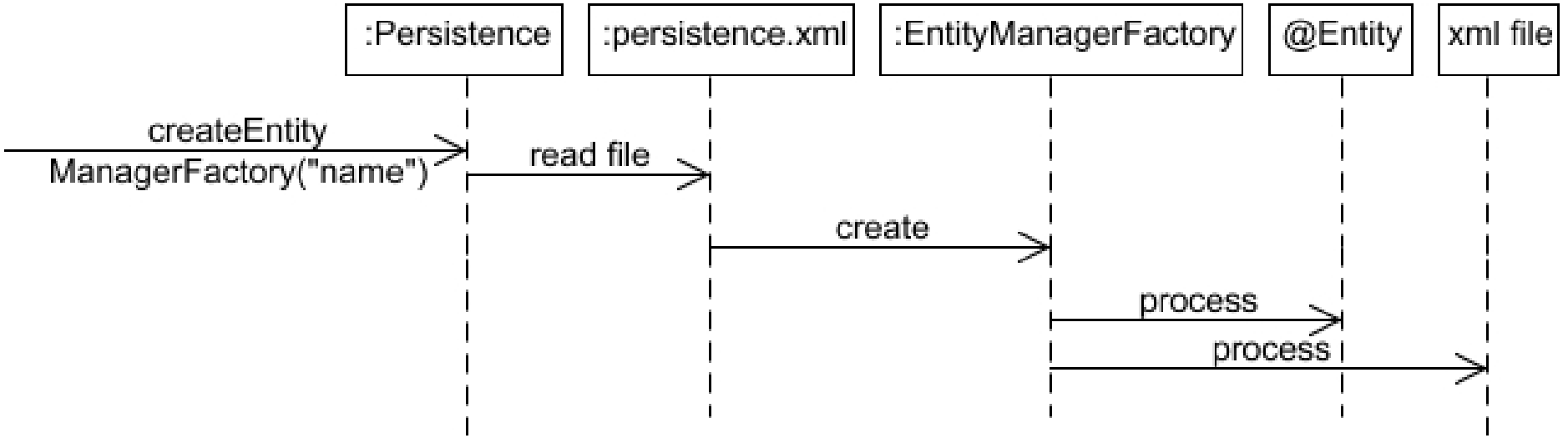
Java Enterprise Edition

Java Persistence API

Trwałość

- Serializacja
  - dane reprezentowane binarnie lub w XMLu
  - brak wyszukiwania
  - brak transakcji
- O/R mapping
  - zazwyczaj: klasa->tabela, obiekt->wiersz, atrybut->wartość w kolumnie
  - ręcznie – JDBC
  - automatycznie – Hibernate, Toplink, JDO
    - można dostroić do istniejącej bazy
- Java Persistence
  - specyfikacja wyłapująca najlepsze pomysły z istniejących rozwiązań
  - POJO
  - adnotacje (deskryptory mają większy priorytet)

Jak zacząć?



- *`Persistence`*
  - punkt wejściowy
  - czyta `persistence.xml` z `META-INF`
- Persistence Unit
  - ma nazwę i opisuje połączenie z bazą
  - zdefiniowane w `persistence.xml`
  - może wymieniać encje, które nadzoruje/nie nadzoruje
- *`EntityManagerFactory`*
  - reprezentuje Persistence Unit

- Persistence Context
  - zbiór encji, których utrwalanie nadzoruje Persistence Provider
  - w jego ramach encje mają unikatową tożsamość i reprezentują trwałe dane
- *`EntityManager`*
  - metody pozwalające nadzorować trwałość
  - cache pierwszego poziomu
  - nie wielowątkowy

Pierwszy przykład

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private int id;
    private String firstName;
    private char middleInitial;
    private String lastName;
    private String streetAddress1;
    private String streetAddress2;
    private String city;
    private String state;
    private String zip;
...
```

Pierwszy przykład

```
EntityManagerFactory emf =
   Persistence.createEntityManagerFactory("examplePersistenceUnit");

EntityManager em = emf.createEntityManager();

Person p1 = new Person("Brett", 'L', "Schuchert", "Street1", "Street2", "City",
                       "State", "Zip");
Person p2 = new Person("FirstName", 'K', "LastName", "Street1", "Street2", "City",
                       "State", "Zip");

em.getTransaction().begin();
em.persist(p1);
em.persist(p2);
em.getTransaction().commit();

final List<Person> list = em.createQuery("select p from Person p").getResultList();

for (Person current : list)
   System.out.println(current.getFirstName());

em.close();
```

Wydzielamy adres

```
@Embeddable
public class Address {
    private String streetAddress1;
    private String streetAddress2;
    private String city;
    private String state;
    private String zip;
...
```

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    int id;
    private String firstName;
    private char middleInitial;
    private String lastName;

    @Embedded
    private Address address;
...
```

Dodajemy firmę

```
@Entity
public class Company {
    @Id
    @GeneratedValue
    int id;
    private String name;
    @Embedded
    private Address address;
    @OneToMany
    private Collection<Person> employees;
...

Company c1 = new Company();
c1.setName("The Company");
c1.setAddress(new Address("D Rd.", "", "Paris", "TX", "77382"));

List<Person> people = generatePersonObjects();
for (Person p : people) c1.hire(p);

em.getTransaction().begin();
for (Person p : people) em.persist(p);
em.persist(c1);
em.getTransaction().commit();
```

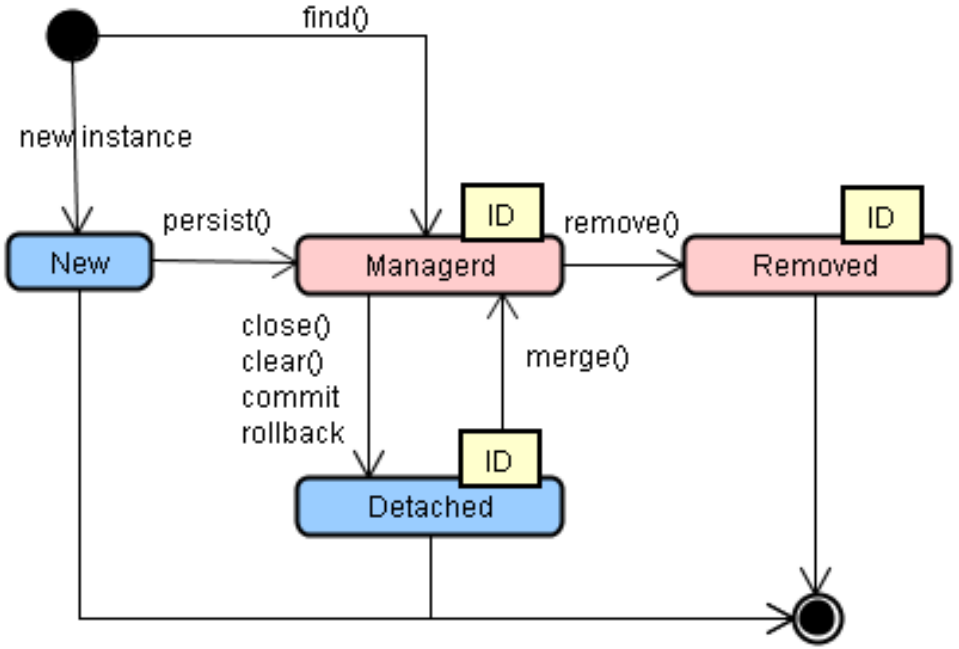
Encje: podstawowe informacje

- Encje to POJO zanotowane `javax.persistence.Entity`
  - tych samych klas da się używać w całej aplikacji
- Musi istnieć publiczny lub chroniony, bezargumentowy konstruktor
- Klasa, metody ani utrwalane atrybuty nie mogą być oznaczone jako `final`
- Encje mogą rozszerzać zarówno inne encje jak i zwykłe klasy
- Zwykłe klasy mogą rozszerzać encje
- Stan encji może być badany przy pomocy bezpośredniego dostępu do atrybutów (`private`, `protected` lub domyślna wid. pak.) lub metodami `get/set`
  - dostęp do atrybutów nie powinien być bezpośredni
  - adnotacje dla właściwości umieszczamy przy getterze
- Atrybuty które mają nie być utrwalane w bazie danych należy zadnotować `javax.persistence.Transient` lub oznaczyć modyfikatorem `transient`

Typy atrybutów/właściwości

- typy podstawowe
- `java.lang.String`
- inne typy serializowalne, w tym:
  - typy opakowujące
  - typy serializowalne zdefiniowane przez użytkownika
  - `java.math.BigInteger, java.math.BigDecimal`
  - `java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.TimeStamp`
  - `byte[], Byte[], char[], Character[]`
- typy wyliczeniowe
- inne encje
- kolekcje encji
  - tylko: `Collection, Set, List, Map`
  - ewentualnie odmiany generyczne
- klasy zanurzone (*embeddable classes*)

Cykl życia



- *new* – nie ma persistence identity, ani nie należy do persistence context
- *managed* – ma i należy
- *detached* – ma i nie należy
- *removed* – należy, ale ma być usunięta z bazy
- po wywołaniu `remove()` encja jest zaplanowana do usunięcia, ale usuwana dopiero przy zatwierdzaniu transakcji lub po wywołaniu `flush()`
- po `merge()` encja zaczyna na powrót należeć
  - przestać może jak zakończył się persistence context (patrz SLSB) lub została zdeserializowana
  - `merge()` zwraca nową encję, a nie przekazany parametr (chyba, że przekazaliśmy encję *managed*)

Związki

- Rodzaje związków
  - 1-1, 1-m, n-1, n-m
  - jedno/dwukierunkowe
  - czyli 7 możliwych kombinacji!
- Związki dwukierunkowe mają dwie strony „owning” oraz „inverse”
  - Strona „inverse” musi się odwoływać do strony „owning” przy pomocy elementu `mappedBy` anotacji `@OneToOne`, `@OneToMany` oraz `@ManyToMany`
  - W dwukierunkowych związkach n-1 strona wiele jest zawsze „owning” i nie może posiadać elementu `mappedBy`
  - W dwukierunkowych związkach 1-1 strona „owning” posiada klucz obcy
  - W dwukierunkowych związkach n-m każda ze stron może być „owning”

Związki c.d.

- Kaskadowość
  - np. `@OneToOne(cascade={CascadeType.PERSIST})`
  - możliwe elementy tablicy – `PERSIST`, `MERGE`, `REMOVE`, `REFRESH` oraz `ALL` – wskazują jakie operacje przenoszą się kaskadowo na elementy pozostające w związku
  - domyślnie tablica jest pusta
- Leniwe/Gorliwe ładowanie
  - ze względu na efektywność domyślnie jest `FetchType.LAZY`
  - jeżeli encje są przekazywane klientowi (*datached*) używamy `FetchType.EAGER`
  - *fetch joins*

Przykład 1-1

```
@Entity(name="OrderUni")
public class Order implements Serializable {
  private int id;
  private String orderName;
  private Shipment shipment;

  @Id
  @GeneratedValue
  public int getId() { return id; }

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

  @OneToOne(cascade={CascadeType.PERSIST})
  public Shipment getShipment() {
    return shipment;
  }

  public void setShipment(Shipment shipment)
   {
    this.shipment = shipment;
  }

  ...
}
```

- Związek w bazie

```
CREATE TABLE ORDERUNI (
  ID INTEGER NOT NULL,
  ORDERNAME VARCHAR(255),
  SHIPMENT_ID INTEGER
);
```

  - specyfikacja określa zasady nazywania klucza obcego
- Dla wersji jednokierunkowej w `Shipment` nie trzeba nic zmieniać

Przykład 1-1

```
BasicConfigurator.configure();
Logger.getLogger("org").setLevel(Level.ERROR);

final EntityManagerFactory emf = Persistence.createEntityManagerFactory("jee6");
final EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

doSomeStuff(em);
System.out.println("Unidirectional One-To-One test\n");

@SuppressWarnings("unchecked")
List<Order> li = em.createQuery("SELECT o FROM OrderUni o").getResultList();
for (Order o : li) {
  System.out.println("Order "+o.getId()+": "+o.getOrderName());
  System.out.println("\tShipment details: "+o.getShipment().getCity()+
                     +o.getShipment().getZipcode());
}

em.getTransaction().commit();
```

1-1 wersja dwukierunkowa

```
@Entity(name="ShipmentBi")
public class Shipment {
  private int id;
  private String city;
  private String zipcode;
  private Order order;

  @Id
  @GeneratedValue
  public int getId() {
    return id;
  }

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

  @OneToOne(mappedBy="shipment")
  public Order getOrder() {
    return order;
  }

  public void setOrder(Order order) {
    this.order = order;
  }
  ...
}
```

- Przy pomocy parametru `mappedBy` wskazujemy, który atrybut encji `Order` realizuje związek
- Reprezentacja w bazie i encja `Order` się nie zmieniają
- Trzeba pamiętać o utrzymywaniu związku między obiektami

```
Shipment s = new Shipment();
s.setCity("Austin");
s.setZipcode("78727");

Order o = new Order();
o.setOrderName("Software Order");
o.setShipment(s);
s.setOrder(o);

em.persist(o);
```

1-m wersja dwukierunkowa

```
@Entity(name="Employee1MBid")
public class Employee {
  private int id;
  private String name;
  private char sex;
  private Company company;

  ...

  @ManyToOne
  public Company getCompany() {
    return company;
  }

  public void setCompany(Company company){
    this.company = company;
  }
}
```

```
@Entity(name="Company1MBid")
public class Company {
  private int id;
  private String name;
  private Collection<Employee> employees =
                        new ArrayList<Employee>();

  ...

  @OneToMany(cascade={CascadeType.ALL}
             fetch=FetchType.EAGER,
             mappedBy="company")
  public Collection<Employee> getEmployees() {
    return employees;
  }

  public void setEmployees(
             Collection<Employee> employees) {
    this.employees = employees;
  }
}
```

1-m wersja dwukierunkowa

```
final EntityManagerFactory emf = Persistence.createEntityManagerFactory("jwt10");
final EntityManager em = emf.createEntityManager();

em.getTransaction().begin();
deleteAll(em);
doSomeStuff(em);

@SuppressWarnings("unchecked")
List<Company> lic = em.createQuery("SELECT c FROM Company1MBid c").getResultList();
for (Company c : lic) {
  System.out.println("Employees of company: "+c.getName());
  for (Employee e : c.getEmployees()) {
    System.out.println("\tName: "+e.getName()+", Sex: "+e.getSex());
  }
  System.out.println();
}
System.out.println();

@SuppressWarnings("unchecked")
List<Employee> lie = em.createQuery("SELECT e FROM Employee1MBid
   e").getResultList();
for (Employee e : lie) {
  Company c = e.getCompany();
  System.out.println("Employee: "+e.getName()+" works for: "+c.getName());
}
em.getTransaction().commit();
```

1-m jedno vs wielokierunkowe

związek jednokierunkowy

```
CREATE TABLE COMPANY1MUNI (
  ID INTEGER NOT NULL,
  NAME VARCHAR(255)
);

CREATE TABLE COMPANY1MUNI_EMPLOYEE1MUNI (
  COMPANY1MUNI_ID INTEGER NOT NULL,
  EMPLOYEES_ID INTEGER NOT NULL
);
--po stronie wiele powinny być więzy UNIQUE

CREATE TABLE EMPLOYEE1MUNI (
  ID INTEGER NOT NULL,
  SEX CHAR(1) NOT NULL,
  NAME VARCHAR(255)
);
```

związek wielokierunkowy

```
CREATE TABLE COMPANY1MBID (
  ID INTEGER NOT NULL,
  NAME VARCHAR(255)
);

CREATE TABLE EMPLOYEE1MBID (
  ID INTEGER NOT NULL,
  SEX CHAR(1) NOT NULL,
  NAME VARCHAR(255),
  COMPANY_ID INTEGER
);
```

EJB-QL

- Grupowe operacje `UPDATE` i `DELETE`
- Złączenia
- `GROUP BY`
- `HAVING`
- Projekcje
- Podzapytania
- Zapytania dynamiczne
- Nazwane zapytania
- Tworzenie w zapytaniu nowych obiektów

Grupowe operacje UPDATE i DELETE

- Operacja dotyczy encji (i wszystkich jej podklas)
- Operacja nie przenosi się kaskadowo na powiązane encje
  - np. w przykładzie 1-m w wersji jednokierunkowej zadziałają więzy dla tabeli złączeniowej
- Zmiany zachodzą bezpośrednio w bazie danych
  - obchodzone jest optymistyczne blokowanie (kolumna z wersją nie jest automatycznie uaktualniana)
- Persistence Context nie jest synchronizowany z wynikiem operacji

```
Query q = em.createQuery("DELETE FROM RoadVehicleSingle");
q.executeUpdate();

Query q = em.createQuery("UPDATE RoadVehicleJoined r SET r.numPassengers = 1");
q.executeUpdate();
```

Złączenia

Mogą służyć do:

- selekcji danych, dla których spełniony jest warunek złączenia
- do kontrolowania gorliwego ładowania

Przykłady (można też używać składni tradycyjnej):

- `JOIN`
  - `SELECT DISTINCT c FROM Company1MUni c JOIN c.employees`
- `LEFT JOIN`
  - `SELECT DISTINCT c FROM Company1MUni c LEFT JOIN c.employees`
- `LEFT OUTER JOIN`
  - `SELECT DISTINCT c FROM Company1MUni c LEFT JOIN FETCH c.employees`

Klauzule GROUP BY i HAVING

Można używać GROUP BY i HAVING

```
SELECT e.sex, count(e)

FROM Employee1MUni e

GROUP BY e.sex

HAVING count(e) > 0
```

Wynikiem jest lista krotek (tu par) obiektów

```
@SuppressWarnings("unchecked")
List<Object[]> lic1 = em.createQuery("SELECT e.sex, count(e)
                                      FROM Employee1MUni e
                                      GROUP BY e.sex
                                      HAVING count(e) > 0").getResultList();
for (Object[] o : lic1) {
  boolean czyKobieta = (Character) o[0] == 'F';
  System.out.println("Na uczelniach pracuje: "+o[1]+" "+
                     ((czyKobieta) ? "kobiet" : "mężczyzn"));
}
```

Projekcje/Podzapytania

Wersja dwukierunkowa

```
SELECT e.name, c.name

FROM Employee1MBid e, Company1MBid c

WHERE e.company = c
```

Wersja jednokierunkowa

```
SELECT e.name, c.name

FROM Employee1MUni e, Company1MUni c

WHERE e IN (SELECT em FROM c.employees em)
```

Albo po prostu

```
SELECT e.name, c.name

FROM Company1MUni c JOIN c.employees e
```

Tworzenie w zapytaniu nowych obiektów

```
class PracFirma {
  String pName;
  String cName;

  public PracFirma(String name, String name2) {
    pName = name;
    cName = name2;
  }

  public String toString() {
    return "PracFirma("+pName+","+cName+")";
  }
}
```

```
@SuppressWarnings("unchecked")

List<PracFirma> li7 =
   em.createQuery("SELECT NEW jee6.reszta.PracFirma(e.name, c.name)
                   FROM Company1MUni c JOIN c.employees e").getResultList();

for (PracFirma o : li7) System.out.println(o);
```

Zapytania dynamiczne

```
Query q4 = em.createQuery("SELECT e FROM Employee1MUni e WHERE e.name LIKE :empName");
@SuppressWarnings("unchecked")
List<Employee> lic4 = q4.setParameter("empName", "%").getResultList();
for (Employee e : lic4) {
  System.out.println("Name: "+e.getName()+", Sex: "+e.getSex()+"\n");
}
```

- uwaga na SQL-injection
- jednokrotnie przygotowywany plan zapytania