Serverske veb tehnologije
- Java Persistence API (JPA) -

Dragan Ivanović

Katedra za informatiku, Fakultet Tehničkih Nauka, Novi Sad

2022.

Java Persistence API (JPA)

- Standardan API koji omogućava snimanje POJO objekata u relacionu bazu
- API implementira neka konkretna biblioteka – Hibernate, TopLink, ...
- JPA-QL: upitni jezik, "objektna varijanta" SQL-a
- O/R mapiranje se opisuje anotacijama
- Nema posebnih konfiguracionih fajlova (kao za klasičan Hibernate)
- JPA nije vezan za EJB kontejner – može da se koristi i za Java SE aplikacije!

Persistence Unit

- Persistence unit predstavlja jednu grupu perzistentnih klasa i parametara mapiranja
- Jedna aplikacija može raditi sa više persistence unita
- Persistence uniti se opisuju u fajlu META-INF/persistence.xml koji mora biti u CLASSPATH-u

EntityManagerFactory

- Na osnovu persistence unita opisanog XML fajlom u programu se kreira **EntityManagerFactory**
- Predstavlja in-memory reprezentaciju O/R mapiranja
- Thread-safe klasa
- Kreiranje je skupo

EntityManager

- Komunikacija sa bazom odvija se u sesijama
- Svaku sesiju opisuje jedan **EntityManager** objekat
- Kreira ga EntityManagerFactory
- Nije thread-safe
- Kreiranje nije skupo

EntityManager metode

- void persist(Object entity)
- T merge(T entity)
- void remove(Object entity)
- T find(Class<T> entityClass, Object primaryKey)
- Query createQuery(String query)
- EntityTransaction getTransaction()
- close()
- ...

JPA entity

- Entity je POJO klasa sa anotacijom **@Entity**
- Mora imati default konstruktor
- Najčešće se mapira 1 klasa $\leftrightarrow$ 1 tabela
- Atributi klase se mapiraju na kolone tabele
- Parametri mapiranja se opisuju anotacijama
- Anotacije se vezuju za atribute ili getter metode

JPA entity

- Entity ne mora da implementira Serializable
- Ako ga implementira, entitiji se mogu prenositi u druge slojeve aplikacije
- Poseban DTO (Data Transfer Object) nije potreban

JPA entity

- Primer 16
  - AdminTest: primer rukovanja entitijem pomoću EntityManagera
  - Admin: primer entity klase
  - persistence.xml: definicija persistence unita

Identitet u Javi

- Identitet objekta (lokacija u memoriji): `x == y`
- Jednakost objekata: `x.equals(y)`
  - da li su jednaka dva `User` objekta sa istim `username` a različitim `password`?

Identitet u bazi podataka

- Isti red (lokacija na disku)
- Vrednost primarnog ključa

Java identitet vs DB identitet

- Kada je Java identitet $\Leftrightarrow$ DB identitet?
- Kada je Java jednakost $\Leftrightarrow$ DB jednakost?

Primer 1

- U prvoj transakciji:
  ```
  x = em.find(User.class, "mbranko");
  ```
- U drugoj transakciji:
  ```
  y = em.find(User.class, "mbranko");
  ```
- Da li je `x == y` ?

Primer 2

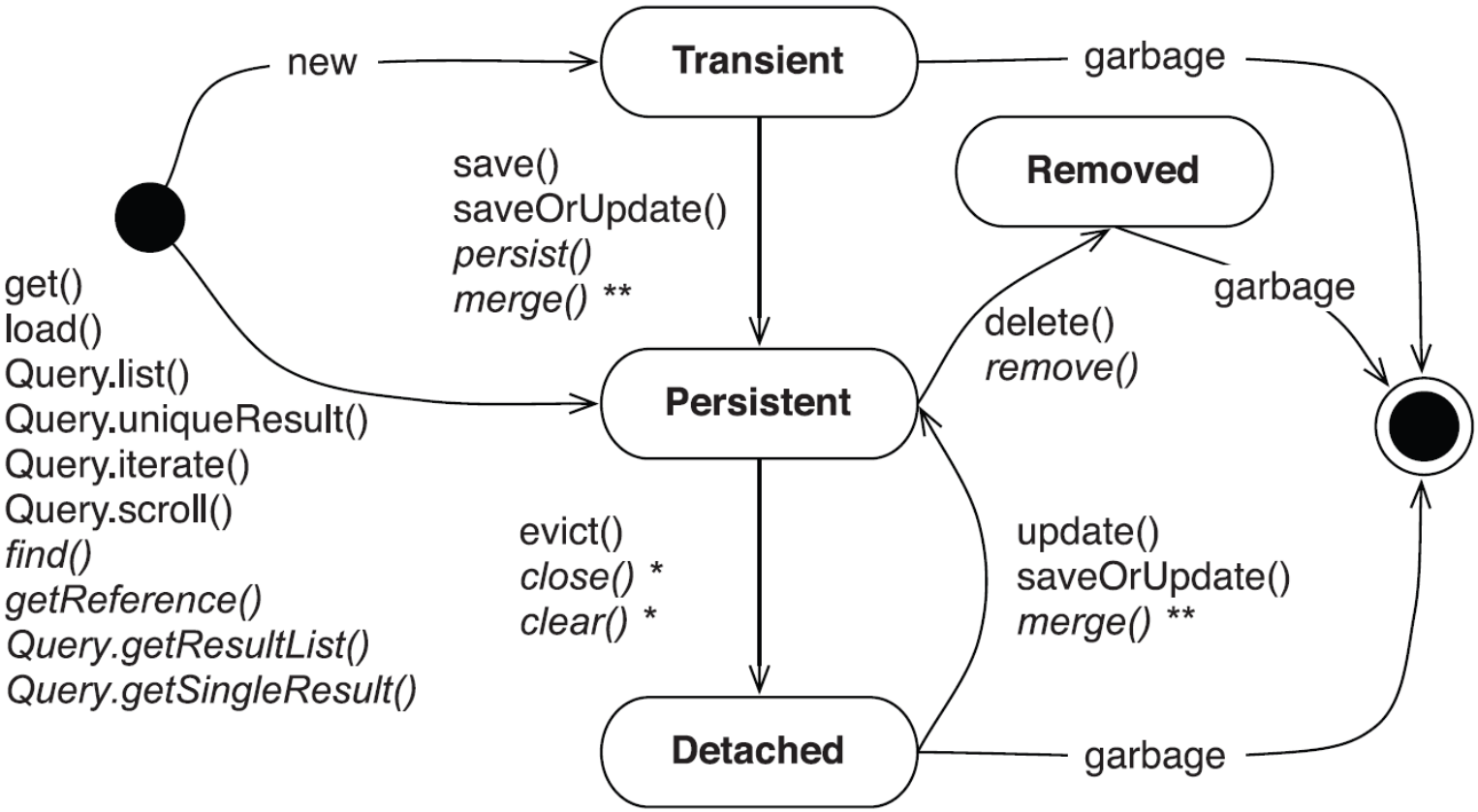
- U prvoj transakciji:
  ```
  x = em.find(User.class, "mbranko");
  ```
- U drugoj transakciji:
  ```
  y = em.find(User.class, "mbranko");
  y.setPassword("trt");
  ```
- Da li je `x.equals(y)` ?

JPA sesija

- Java identitet (i jednakost) važi za perzistentne objekte **unutar jedne sesije**!

```
EntityManager em = emf.createEntityManager();
...
em.close();
```

Životni ciklus entitija



Tipovi veza između entitija

- Posmatramo dve klase, A i B, koje su u vezi
- Veza tipa 1:1
  - klasa A sa atributom tipa B, anotacija @OneToOne
  - klasa B sa atributom tipa A, anotacija @OneToOne
- Veza tipa 1:n
  - 1-strana ima anotaciju @OneToMany, tip atributa je Set<B>
  - n-strana ima anotaciju @ManyToOne, tip atributa je A
  - n-strana obično ima i anotaciju @JoinColumn koja opisuje join uslov
- Veza tipa m:n
  - m-strana ima anotaciju @ManyToMany, tip atributa je Set<B>
  - n-strana ima anotaciju @ManyToMany, tip atributa je Set<A>
  - opciono i @JoinColumn

Jedno- i dvosmerne veze

- Jednosmerna veza: klasa A „vidi“ klasu B, a klasa B „ne vidi“ klasu A
- Dvosmerna veza: klasa A „vidi“ klasu B i obrnuto
- Prethodni slajd podrazumeva dvosmernu vezu
- Jednosmernu vezu pravimo izostavljanjem odgovarajućeg atributa u klasi

Uspostavljanje veze

- Važno pravilo: uspostavljanje veze između objekata mora da se vrši **kao da se ne koristi JPA**
- Ako je dvosmerna, veza mora da se ažurira sa obe strane
- Suprotno od EJB 2.1!

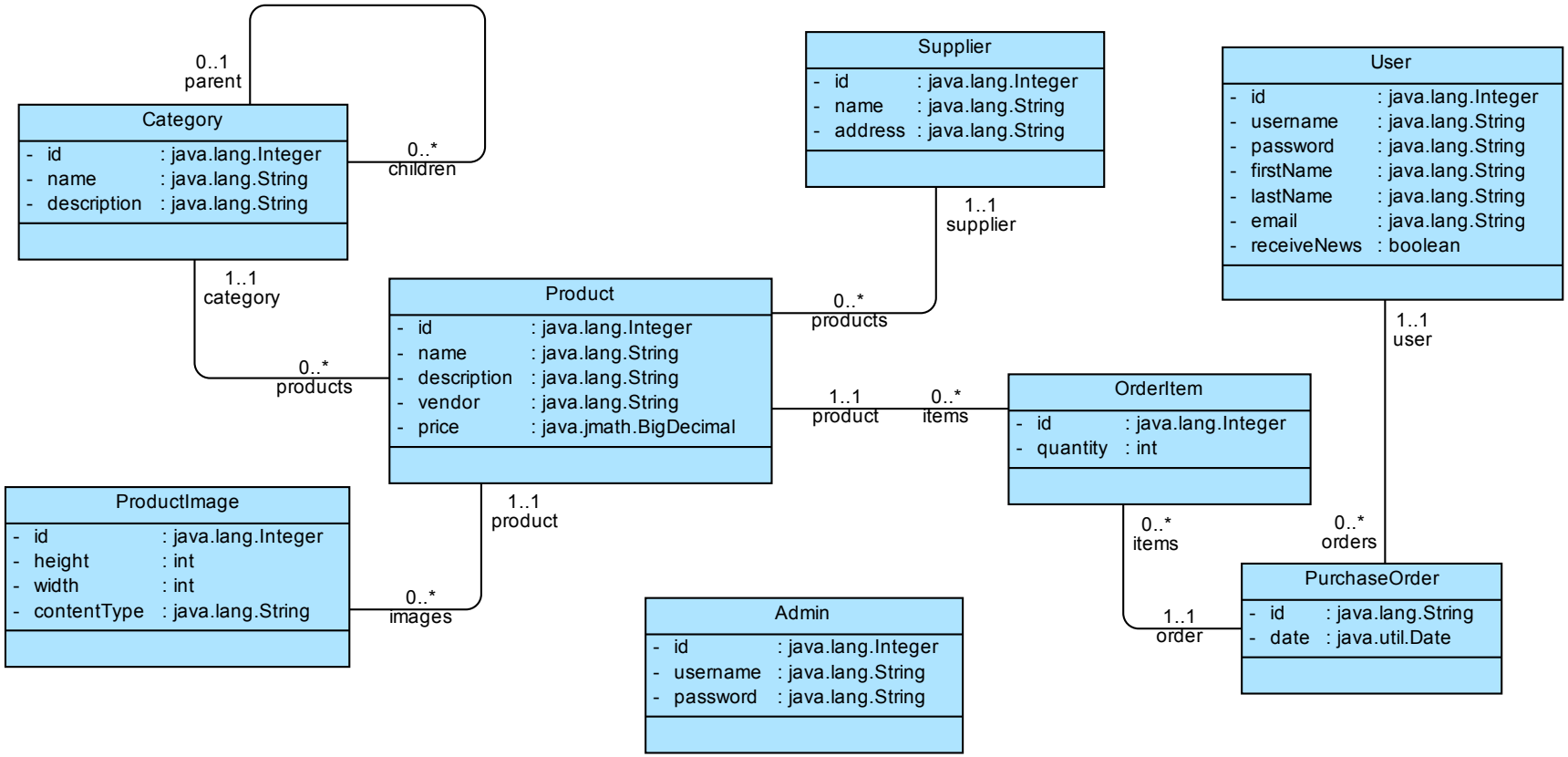
```
// oba reda su obavezna
product.setCategory(category);
category.getProducts().add(product);
```

Inicijalizacija atributa

- Atribut tipa Set<X> se mora inicijalizovati u prilikom konstrukcije objekta
- Obično se za inicijalizaciju koristi HashSet<X>
- JPA engine će taj kasnije taj objekat zameniti svojom Set implementacijom

```
class Category {
  ...
  private Set<Product> products = new HashSet<Product>();
  ...
}
```

Primer 17


Category
- id : java.lang.Integer
- name : java.lang.String
- description : java.lang.String
0..1
parent
0..*
children
1..1
category
0..*
products
Supplier
- id : java.lang.Integer
- name : java.lang.String
- address : java.lang.String
1..1
supplier
0..*
products
Product
- id : java.lang.Integer
- name : java.lang.String
- description : java.lang.String
- vendor : java.lang.String
- price : java.jmath.BigDecimal
1..1
product
0..*
items
OrderItem
- id : java.lang.Integer
- quantity : int
0..*
items
1..1
order
User
- id : java.lang.Integer
- username : java.lang.String
- password : java.lang.String
- firstName : java.lang.String
- lastName : java.lang.String
- email : java.lang.String
- receiveNews : boolean
1..1
user
0..*
orders
PurchaseOrder
- id : java.lang.String
- date : java.util.Date
ProductImage
- id : java.lang.Integer
- height : int
- width : int
- contentType : java.lang.String
1..1
product
0..*
images
Admin
- id : java.lang.Integer
- username : java.lang.String
- password : java.lang.String

Dodavanje novog elementa u Set

- Set ne prihvata duplikate
- Prilikom dodavanja novog elementa, proverava se da li je element već tamo
- Provera se oslanja na `equals()` i `hashCode()`
- ...a oni su nasleđeni iz klase `Object` i ne rade kako treba!

equals() za entitije

- `Object.equals()` $\Leftrightarrow$ `==`
- Dva različita objekta u memoriji mogu predstavljati isti red u bazi!
- Treba redefinisati `equals()` tako da koristi primarni ključ u poređenju

```
public boolean equals(Object o) {
  return this.id.equals(o.id);
}
```

equals() za entitije

- `Object.equals()` $\Leftrightarrow$ `==`
- Dva različita objekta u memoriji mogu predstavljati isti red u bazi!
- Treba redefinisati `equals()` tako da koristi primarni ključ u poređenju

```
public boolean equals(Object o) {
  return this.id.equals(o.id);
}
```

- Međutim, `id` nije definisan pre nego što se objekat snimi u bazu!

equals() za entitije

- Dodatak: ako je `id == null` za bilo koji od dva objekta, smatramo da su različiti

```
public boolean equals(Object that) {
  if (this == that)
    return true;
  if (this.id == null || that.id == null)
    return false;
  return this.id.equals(other.id);
}
```

equals() za entitije

- Dodatak: ako je `id == null` za bilo koji od dva objekta, smatramo da su različiti

```
public boolean equals(Object that) {
  if (this == that)
    return true;
  if (this.id == null || that.id == null)
    return false;
  return this.id.equals(other.id);
}
```

- Sledeći problem: ako su dva objekta jednaka, moraju imati isti `hashCode()`
- Pri tome vrednost `hashCode()` ne sme da se menja – izgubićemo objekte u Setu

hashCode() za entitije

```
private Integer hashcodeValue = null;
public int hashCode(){
  if (hashcodeValue == null) {
    if (id == null)
      hashcodeValue = new Integer(super.hashCode());
    else
      hashcodeValue = id;
  }
  return hashcodeValue.intValue();
}
```

- Kada se jednom upotrebi `hashCode()`, više se neće menjati

hashCode() za entitije

```
private Integer hashcodeValue = null;
public int hashCode(){
  if (hashcodeValue == null) {
    if (id == null)
      hashcodeValue = new Integer(super.hashCode());
    else
      hashcodeValue = id;
  }
  return hashcodeValue.intValue();
}
```

- Kada se jednom upotrebi `hashCode()`, više se neće menjati
- Problem: napravimo novi objekat, snimimo ga, zatvorimo sesiju, kasnije učitamo objekat u novoj sesiji i dobijemo dva objekta za koje važi `a.equals(b)` ali je `a.hashCode() != b.hashCode()`

Drugo rešenje za equals() i hashCode()

- Za poređenje koristimo one atribute koji su po svojoj prirodi jedinstveni
- Npr. za klasu `User` atribut `username` je jedinstven i **ne menja se nakon što se inicijalizuje** (tj. korisnik ne može da promeni svoj username kada se jednom registruje)

```
public int hashCode(){
  return username.hashCode();
}
public boolean equals(Object that) {
  if (this == that)
    return true;
  if (that == null)
    return false;
  return this.username.equals(that.username);
}
```

Drugo rešenje za equals() i hashCode()

- Za poređenje koristimo one atribute koji su po svojoj prirodi jedinstveni
- Npr. za klasu `User` atribut `username` je jedinstven i **ne menja se nakon što se inicijalizuje** (tj. korisnik ne može da promeni svoj username kada se jednom registruje)

```
public int hashCode(){
  return username.hashCode();
}
public boolean equals(Object that) {
  if (this == that)
    return true;
  if (that == null)
    return false;
  return this.username.equals(that.username);
}
```

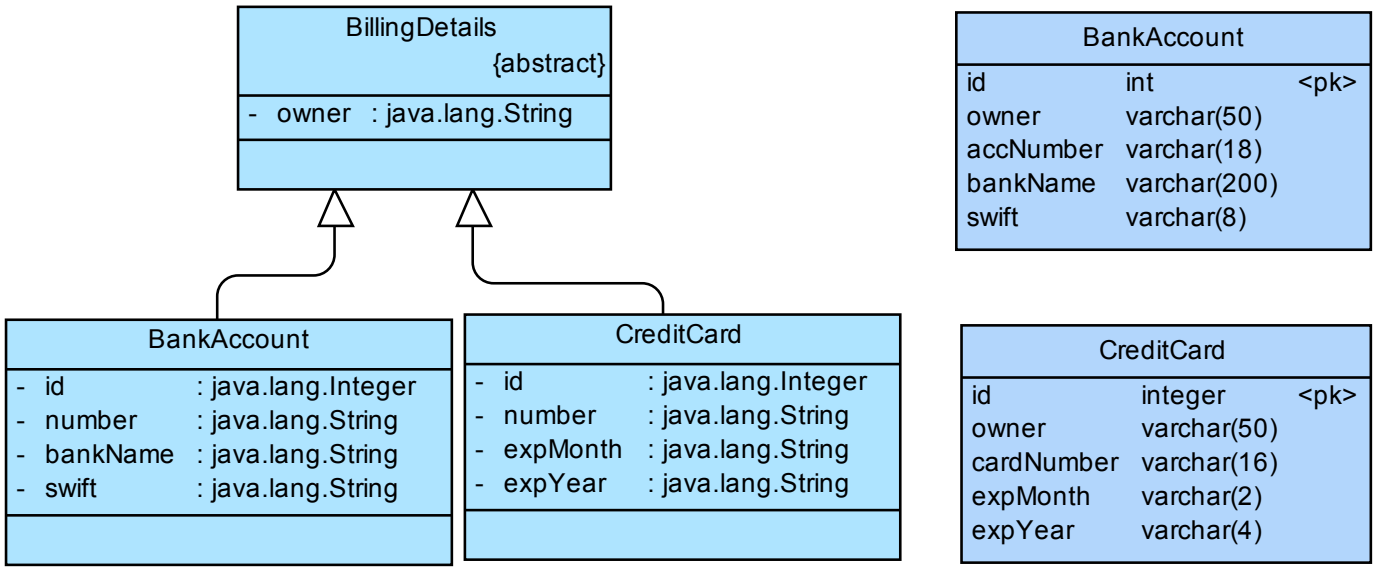
Idealno rešenje za equals() i hashCode()

- Ne postoji
- Sve zavisi od načina upotrebe entitija
- Ako imamo atribut(e) sa jedinstvenim vrednostima, druga varijanta je najbolja
- Ako ih nemamo, prva varijanta može biti dovoljno dobra
- Treća varijanta: ne redefiniši `equals()` i `hashCode()` i pazi šta radiš
- Četvrta varijanta: GUID koji se inicijalizuje kod kreiranja objekta i koristi za `equals()` i `hashCode()` – može kao dodatni atribut ili čak primarni ključ
- Dobra diskusija: `http://www.hibernate.org/109.html`

Četiri varijante mapiranja nasleđivanja

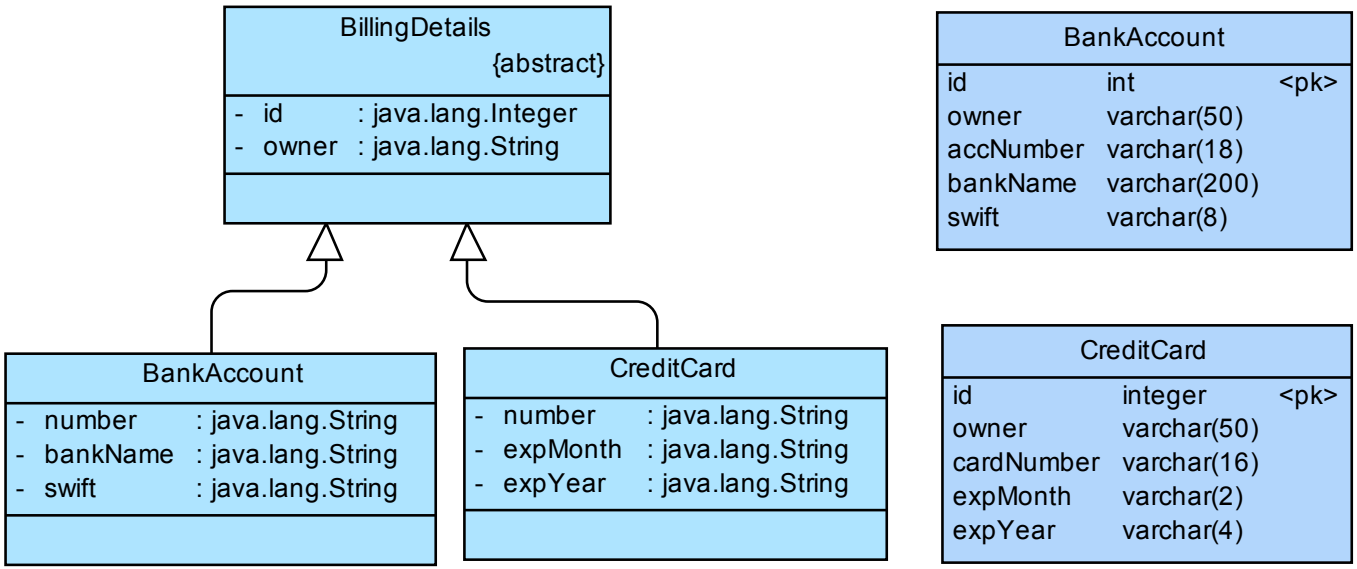
- Jedna tabela po konkretnoj klasi sa implicitnim polimorfizmom
- Jedna tabela po konkretnoj klasi
- Jedna tabela po hijerarhiji nasleđivanja
- Jedna tabela za svaku klasu

1 tabela po konkretnoj klasi sa implicitnim polimorfizmom



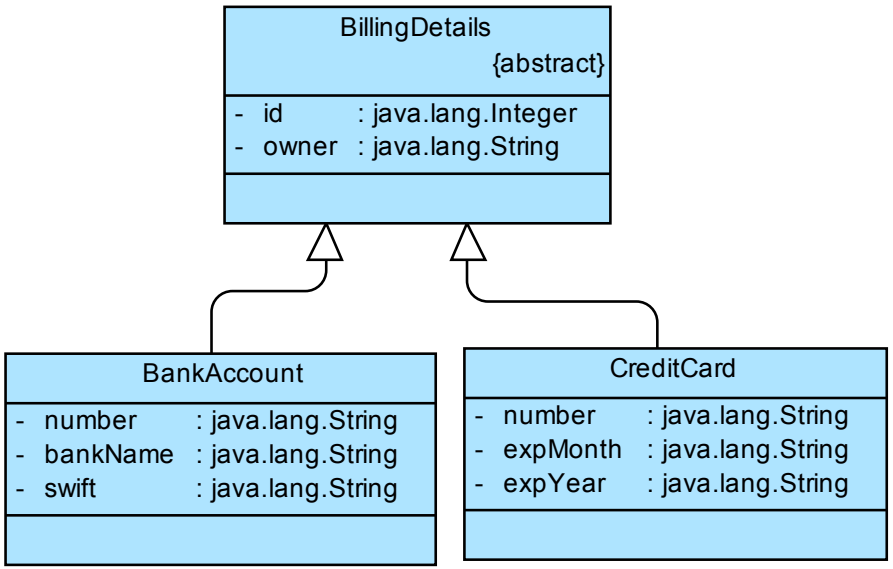
- Primer 22 – osa.pr22.v1.*

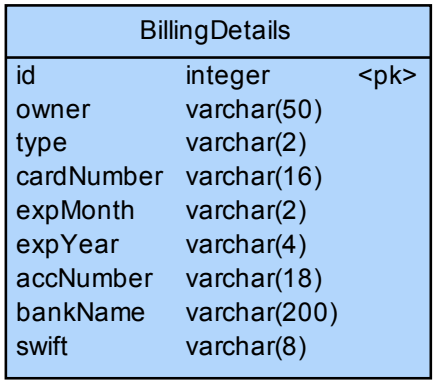
Jedna tabela po konkretnoj klasi



- Primer 22 – osa.pr22.v2.*

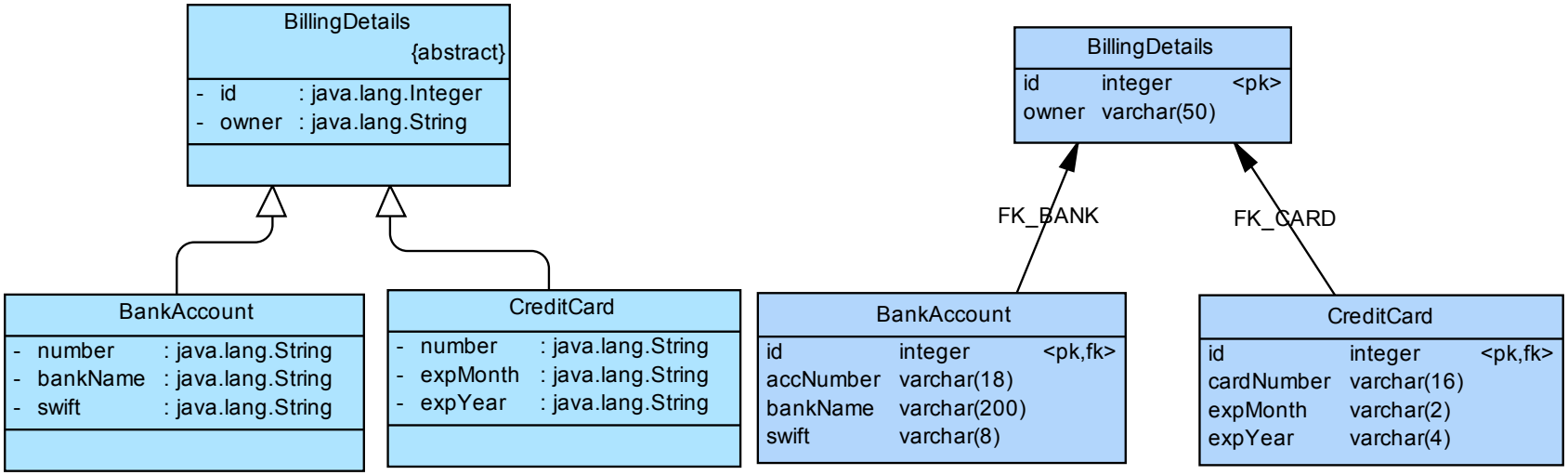
Jedna tabela po hijerarhiji nasleđivanja





- Primer 22 – osa.pr22.v3.*

Jedna tabela za svaku klasu



- Primer 22 – osa.pr22.v4.*

Šta biramo za primarni ključ?

- Neko prirodno obeležje koje je jedinstveno i nepromenljivo – prirodni ključ
  - JMBG, PIO broj, ...
  - može i grupa obeležja, npr. kontni okvir: šifra klase + šifra grupe + ...
- Veštačko obeležje koje je jedinstveno – surogatni ključ
  - integer brojač, UUID, ...
  - ključ čini uvek jedno obeležje

Prirodni vs surogatni ključ

- Prirodni ključevi

za
ne mora se izmišljati novo obeležje
protiv
obeležje nije baš nepromenljivo
ne mora biti integer tipa $\rightarrow$ manje efikasno indeksiranje

- Surogatni ključevi

za
efikasno indeksiranje
nema više od jednog obeležja u ključu
protiv
vrednost nema drugi smisao osim da bude jedinstvena

Prirodni i surogatni ključevi i JPA

- Za JPA se preporučuje upotreba surogatnih ključeva
- Znatno jednostavnije mapiranje
- Jednostavna provera da li treba raditi update (ključ $\neq$ null) ili insert (ključ = null)

- Podržani su i prirodni ključevi
- Manje efikasan rad
- Manje elegantan objektni model u slučaju kompozitnih ključeva

Generisanje vrednosti surogatnih ključeva

- Identity / auto_increment / . . . kolona u bazi
- Sekvenca
- Tabela sa brojačima

counter_name	counter_value
users	731
products	8432
. . .	. . .

- Primer 23 – osa.pr23.surrogate.*

JPA i prirodni ključevi

- Ako prirodni ključ čini jedno obeležje, on se označava sa @Id, kao i ranije
- Ako ima više obeležja u ključu, mora se napraviti posebna „PK“ klasa
- Atribut tipa PK klase se dodaje u osnovnu klasu i označava sa @EmbeddedId

- Spoljni ključ koji se sastoji iz više obeležja se opisuje @JoinColumns anotacijom
- Osim ako je spoljni ključ deo primarnog ključa – tada se izražava u PK klasi
- Objektni model više nije elegantan!

- Primer 23 – osa.pr23.natural.*

JPA - Data Access Object (DAO) sloj

- U praksi su za svaki entity potrebne uobičajene CRUD (create, retrieve, update, delete) operacije
- Njih obično implementiraju posebne DAO klase
- Jedan entity – jedan DAO
- Ima dosta "pešačkog" posla

Generički DAO: implementacija zajedničkih operacija

```
public interface GenericDao<T, ID extends Serializable> {
public Class<T> getEntityType();
public T findById(ID id);
public List<T> findAll();
public List<T> findBy(String query);
public T persist(T entity);
public T merge(T entity);
public void remove(T entity);
public void flush();
public void clear();
}
```

Generički DAO: implementacija zajedničkih operacija

```
public abstract class GenericDaoBean<T, ID extends Serializable>
implements GenericDao<T, ID> {
...

protected EntityManager em;
...
}
```

Konkretni DAO za entity User

```
public interface UserDao extends GenericDao<User, Integer> {
public User login(String username, String password);
}
```

```
public class UserDaoBean extends GenericDaoBean<User, Integer>
implements UserDao {

public User login(String username, String password) { ... }
}
```

Spring - Sloj za upravljanje podacima

- Koristi se Spring Data JPA - anotacija @Repository
- Koristi JPA specifikaciju za objektno-relaciono mapiranje
- Podrška za jednostavan razvoj sloja za pristup podacima
- Eliminiše potrebu ponovnog pisanja sličnog koda
- Programer samo specificira šta želi da dobije od podataka - samo dobavljanje će obaviti Spring Data JPA

Spring Data JPA

- Osnovni koncept je Repozitorijum
  - Predstavljen interfejsom *Repository*
  - Dobija tip entiteta sa kojim radi i obezbeđuje standardne akcije rukovanja podacima tog tipa
- Interfejs *CrudRepository*
  - Specijalizacija generičkog repozitorijuma
  - Sadrži standardne CRUD (Create, Update, Delete) operacije nad entitetom

PagingAndSortingRepository

- Specijalizacija CRUD repozitorijuma
- Omogućuje
  - Paginaciju – dobavljanje podataka u manjim grupama (stranicama)
  - Sortiranje podataka po zadatom kriterijumu

JPARepository

- Specijalizacija PagingAndSortingRepository interfejsa
- Dodatna podrška za JPA operacije - npr. pražnjenje perzistentnog konteksta (eng. *persistence context*)
- Ovaj interfejs pruža sve najčešće operacije potrebne za rad sa podacima u standardnom veb informacionom sistemu

Implementacija upravljanja podacima

- Obezbeđuje se kreiranjem interfejsa koji nasleđuje neki od repozitorijumskih interfejsa
  - Nije potrebno praviti klasu koja implementira kreirani interfejs
  - Ovim se mogu koristiti operacije predviđene nasleđenim repozitorijumskim interfejsom
- U kreirani interfejs se mogu dodavati i nove operacije nad podacima
  - Ne moraju se operacije implementirati
  - Dovoljno je u interfejsu napisati deklaraciju query metode
  - Za specifične operacije, može se kreirati klasa koja nasleđuje interfejs i u njoj implementirati metodu koja obavlja operaciju

Query metode

- Ideja je da se poštovanjem konvencije u imenovanju metode, metoda samo deklariše
  - Na osnovu deklaracije koja poštuje specificiranu formu, Spring automatski obezbeđuje implementaciju
- Moguće je i specijalnim parametrima zahtevati sortiranje i paginaciju i dobiti jednu stranicu podataka

Query metode

- Ideja je da se poštovanjem konvencije u imenovanju metode, metoda samo deklariše
  - Na osnovu deklaracije koja poštuje specificiranu formu, Spring automatski obezbeđuje implementaciju
- Moguće je i specijalnim parametrima zahtevati sortiranje i paginaciju i dobiti jednu stranicu podataka

Spring boot repository

- Spring Primer 17