

JPA

Vještina: Osnove programskog jezika Java

Zagreb, lipanj 2018.

Što je JPA?

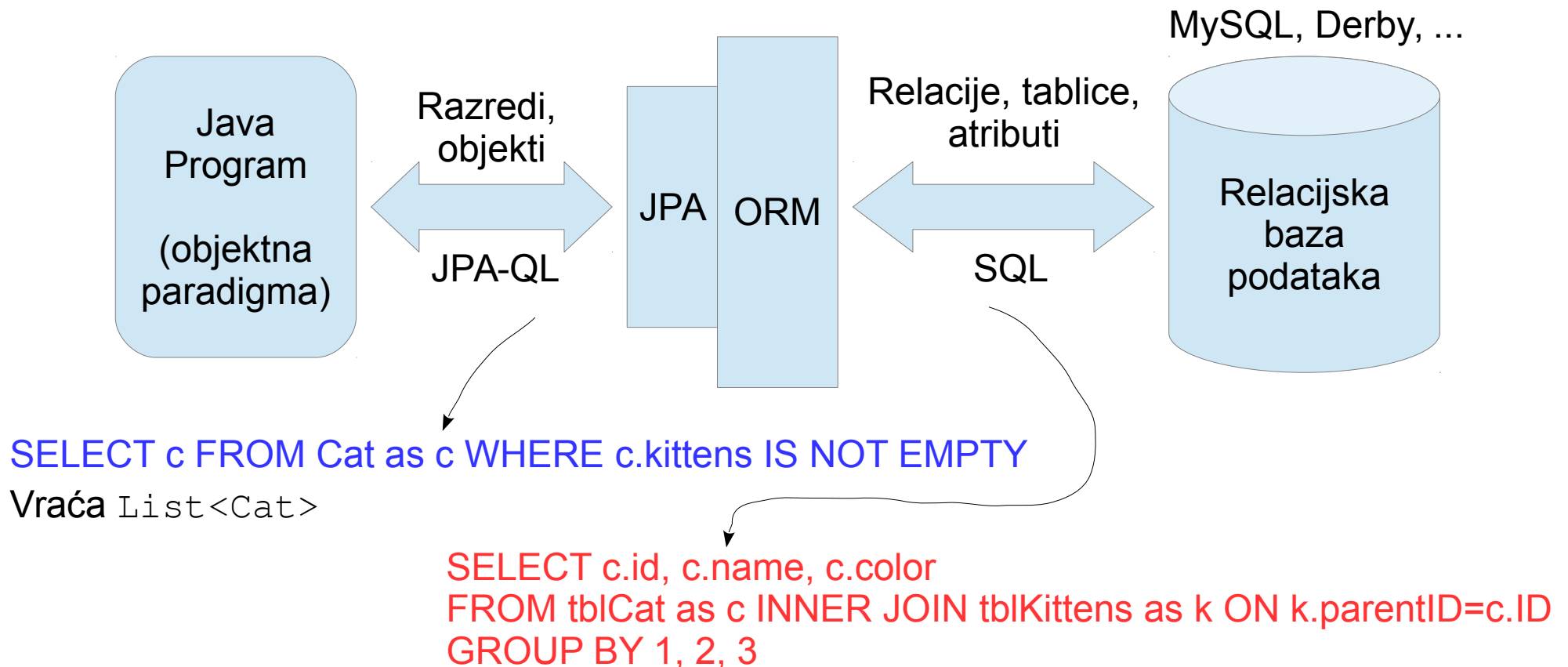
- Java Persistence API
- Specifikacija koja omogućava Java programu da iz relacijske baze podataka čita/zapisuje objekte
- Temelji se na uporabi ORM-a
 - *Object Relation Mapper*
 - Podsustav koji sjedi između klijentskog koda i relacijske baze podataka

Što je JPA?

- ORM je biblioteka koja nudi objektnu apstrakciju podataka kroz sučelje prema klijentu
- Nudi metode za stvaranje, dohvat, izmjenu i brisanje objekata koji se pohranjuju u relacijskoj bazi podataka
- Nudi upitni jezik više razine od SQL-a koji omogućava postavljanje upita na razini objekata i relacija između njih

Što je JPA?

- Interno generira potrebne SQL-upite koje razmjenjuje s relacijskom bazom podataka



Što je JPA?

- Postoji mnoštvo različitih implementacija ORM-ova koji klijentima nude vlastiti specifični API
 - Hibernate
 - Oracle TopLink
 - EclipseLink
 - ...
- JPA specifikacija prema klijentima nudi unificiran API a navedene biblioteke ga implementiraju

Što je JPA?

- ORM-biblioteke očekuju da baza podataka već postoji, kao i korisnik koji se na nju može spajati
- Većina popularnih biblioteka sve ostalo zna raditi sama
 - Uključivo i stvaranje tablica ako ne postoje
 - Ažuriranje strukture tablica ako se model podataka u programu promijeni
 - ...

Što je JPA?

- Ponekad ORM-biblioteka treba pomoć kako bi ispravno protumačila strukturu objektnog modela
 - Na razini OOP jezika nema pojma “identifikator objekta”/”jedinostveno za objekt”/...
 - Nema načina da se pobliže specificiraju domenska ograničenja (String ne dulji od 5 znakova; ne-null vrijednost, ...)

Što je JPA?

- Stoga ORM-biblioteka očekuje:
 - Ili u zasebnoj datoteci detaljniji opis modela s ograničenjima (engl. *Mapping-file*)
 - Ili prikladno obilježen (anotiran) izvorni kod razreda za to predviđenim anotacijama

Usporedba s JDBC-om

- Vezu prema bazi podataka u JDBC-u modelira `Connection`
 - Kod JPA to je `EntityManager`
- Za stvaranje veza prema bazi u JDBC-u je zadužen `DriverManager` ili `DataSource` (koji veze može ili ne mora "pool"ati)
 - Kod JPA to je `EntityManagerFactory`
 - On interno treba izravnu JDBC vezu prema bazi pa prima sve potrebne podatke pri konfiguriranju

Usporedba s JDBC-om

- Početni objekt `EntityManagerFactory` stvara se preko statičke metode razreda

`Persistence.createEntityManagerFactory("naziv")`
kojoj se predaje naziv konfiguracije

- Ista očekuje da postoji datoteka `META-INF/persistence.xml` koja sadrži `<persistence-unit>` tog imena sa svom potrebnom konfiguracijom (URL do baze, podatci za autentifikaciju/autorizaciju, konfiguraciju connection-poola, keševa itd.)

Uporaba

```
// Na pocetku programa:  
EntityManagerFactory emf =  
Persistence.createEntityManagerFactory("ime");  
  
// Za svaki posao:  
EntityManager em = emf.createEntityManager();  
em.getTransaction().begin();  
odradiPosao(em);  
em.getTransaction().commit();  
em.close();  
  
// Na kraju programa  
emf.close();
```

Uporaba

- Važno: JPA implementacija prati sve objekte dohvaćene kroz `EntityManager` i pri komitanju transakcije automatski šalje potrebne naredbe `UPDATE` u relacijsku bazu ako je objekt modificiran
 - Posljedica: ne postoji naredba “`save()`”!
 - Imamo `em.persist(noviObj)` kojeg treba pozvati samo ako treba pohraniti novostvoreni objekt (ekvivalent SQL-naredbi `UPDATE`)
 - Imamo `em.delete(objekt)` za brisanje objekata

Uporaba

- Automatsko praćenje može biti problematično u određenim primjenama
 - Web-aplikacija, stranica nudi formular za ažuriranje podataka o korisniku
 - Dohvatimo korisnika iz baze, ažuriramo ime prema formularu, ažuriramo prezime prema formularu, nađemo grešku u sintaksi e-maila...
 - Problem: klijentu ćemo vratiti formular na doradu ali ne želimo parcijalno ažuriranje poslati u bazu...
 - `transaction.rollback()`? Nismo sretni...
 - Vodi na uporabu objekata koji modeliraju formulare

Uporaba

- Razred `Korisnik` i `KorisnikFormular`
 - `KorisnikFormular` ima sva svojstva kao `String`ove te za svako svojstvo i zastavicu `error` te polje `errorMsg`

- Stvaranje novog korisnika:



- 1) Zakvači `new KorisnikFormular()` u attribute
- 2) `forward` JSP-u na crtanje

- 1) Stvori `kf = new KorisnikFormular()`
- 2) popuni ga iz primljenih parametara
- 3) validiraj; ako nešto ne valja, evidentiraj greške u formular, zakvači ga u attribute i `forward` JSP-u na crtanje
- 4) inače stvori `k = new Korisnik()`, napuni ga iz `kf` i pohrani u bazu; redirect na `uspjeh`

Uporaba

- Uređivanje postojećeg korisnika:



- 1) Zakvači new KorisnikFormular() u attribute
- 2) Dohvati iz baze traženog korisnika:
`Korisnik k = em.find(Korisnik.class, 13)`
- 3) Napuni formular kf iz podataka k
- 2) forward JSP-u na crtanje

- 1) Stvori kf = new KorisnikFormular()
- 2) popuni ga iz primljenih parametara
- 3) validiraj; ako nešto ne valja, evidentiraj greške u formular, zakvači ga u attribute i forward JSP-u na crtanje
- 4) inače dohvati iz baze
`k = em.find(Korisnik.class, 13)`, napuni ga iz kf; redirekt na uspjeh (automatski će biti pohranjen)

Razlika između /new i /edit: prvi u formular za ID upisuje null, drugi šalje valjani ID pa /update zna treba li ažurirati korisnika ili ga stvoriti prema predanim podacima

Uporaba

- Opisana organizacija web-aplikacije ilustrirana je primjerom adresara
 - Dostupno u dodatcima uz ovo predavanje kao adresar.zip
 - Za domaću zadaću ovo je potrebno skinuti, upogoniti i proučiti kako je implementirano
 - Posebnu pažnju posvetiti upravo trima servletima koji rade /new, /edit i /update
 - Koristi se jednostavna simulacija baze podataka
 - Nema veze s tehnologijom JPA

JPA

Ukratko o označavanju entiteta

Entitet?

- **Entitet** je razred čiji se objekti pohranjuju u bazu podataka i koji imaju vlastiti identitet (imaju primarni ključ po kojem je dohvativ)
 - Npr. Student, Kolegij, Semestar, Racun, Proizvod, ...
- Pri anotiranju razreda "obična" svojstva te veze prema drugim entitetima označavaju se različitim anotacijama

Entitet?

- Objekt može imati:
 - Primitivna svojstva modelirana tipovima poput `String`, `long`, `double`, ...
 - Kolekcije primitivnih svojstava
 - Reference prema nekom drugom entitetu
 - Kolekciju referenci prema nekom drugom entitetu
- Svaki od ovih slučajeva drugačije se označava

Veze prema više entiteta

- Ukoliko između jedne vrste entiteta i druge vrste entiteta postoji veza tipa *xToMany* (npr. `Korisnik` ima više `Posudba`), JPA dozvoljava da se ta veza u kodu iskaže uporabom raznih sučelja (npr. `Set`, `List`, `Map`)
- Međutim, ne poštuje se semantika tih sučelja (primjerice, `List` ne garantira poredak!)

Primarni ključ entiteta

- Svaki entitet mora imati primarni ključ prema kojem se primjerci tog entiteta razlikuju
- Ključ može biti jednostavan (jedan `Long`, `String`, ...) ili složen (kompozitan)
- Preporuča se uporaba jednostavnih ključeva
- Mi ovdje nećemo govoriti o kompozitnim ključevima i načinima njihovog definiranja (opisano je u specifikaciji)

Označavanje entiteta

@Entity

@Table(name="customer records")

public class CustomerRecord {

@Id @GeneratedValue

private Long id;

@Column(nullable=false, length=30)

private String firstName;

@Column(nullable=false, length=50)

private String lastName;

@Column(nullable=true, length=20, unique=true)

private String nickName;

@Temporal(TemporalType.TIMESTAMP)

private Date lastVisit;

}

Dvosmjerna 1-na-1

```
@Entity
public class Employee {
    private Cubicle assignedCubicle;
```

Vlasnik relacije

Ima strani ključ na Cubicle

```
    @OneToOne
    public Cubicle getAssignedCubicle() {
        return assignedCubicle;
    }
    public void setAssignedCubicle(Cubicle cubicle) {
        this.assignedCubicle = cubicle;
    }
    ...
}
```

```
@Entity
public class Cubicle {
    private Employee residentEmployee;
```

```
    @OneToOne(mappedBy="assignedCubicle")
```

```
    public Employee getResidentEmployee() {
        return residentEmployee;
    }
```

```
    public void setResidentEmployee(Employee employee) {
        this.residentEmployee = employee;
    }
    ...
}
```

**Inverzni
kraj**

Dvosmjerna 1-na-1

- U prethodnom primjeru razred `Cubicle` ima svojstvo (*property*) `residentEmployee` ali je ono označeno s `mappedBy=x`, gdje je `x` naziv svojstva kojim objekt s druge strane pokazuje na ovaj objekt
- Posljedica: u relacijskoj bazi podataka, druga strana će u svojoj tablici imati strani ključ na objekt tipa `Cubicle`, ali `Cubicle` neće imati strani ključ na pripadni `Employee`

Dvosmjerna 1-na-1

- U dvosmjernim vezama bilo kojeg tipa, samo jedan kraj smije biti vlasnik relacije
 - Drugi kraj nužno mora biti proglašen inverzним krajem što se radi označavanjem s `mappedBy`

Dvosmjerna n-na-1 / 1-na-n

```
@Entity
public class Employee {
    private Department department;
```

Vlasnik relacije
Ima strani ključ na Department

```
    @ManyToMany
    public Department getDepartment() {
        return department;
    }
    public void setDepartment(Department department) {
        this.department = department;
    }
    ...
}
```

```
@Entity
public class Department {
    private Collection<Employee> employees = new HashSet();

    @OneToMany(mappedBy="department")
    public Collection<Employee> getEmployees() {
        return employees;
    }

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

Inverzni kraj



Jednosmjerna 1-na-1

```
@Entity
public class Employee {
    private TravelProfile profile;

    @OneToOne
    public TravelProfile getProfile() {
        return profile;
    }
    public void setProfile(TravelProfile profile) {
        this.profile = profile;
    }
    ...
}
```

Vlasnik relacije
Ima strani ključ na TravelProfile



```
@Entity
public class TravelProfile {
    ...
}
```

Ovaj razred ne spominje Employee;
Ako imamo primjerak ovog razreda,
ne možemo izravno doći do vlasničkog
objekta tipa Employee.

U bazi podataka, strani ključ Employee::profileID će imati
nad sobom ograničenje UNIQUE

Jednosmjerna n-na-1

```
@Entity
public class Employee {
    private Address address;
```

Vlasnik relacije
Ima strani ključ na Address

```
    @ManyToMany
    public Address getAddress() {
        return address;
    }
    public void setAddress(Address address) {
        this.address = address;
    }
    ...
}
```

```
@Entity
public class Address {
    ...
}
```

U bazi podataka, strani ključ Employee::profileID neće imati nad sobom ograničenje UNIQUE – legalno je da više zaposlenika živi na istoj adresi.

Dvosmjerna n-na-n

```
@Entity
public class Project {
    private Collection<Employee> employees; Vlasnik relacije
```

```
    @ManyToMany
    public Collection<Employee> getEmployees() {
        return employees;
    }

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

```
@Entity
public class Employee {
    private Collection<Project> projects;
```

```
    @ManyToMany(mappedBy="employees")
    public Collection<Project> getProjects() {
        return projects;
    }
```

```
    public void setProjects(Collection<Project> projects) {
        this.projects = projects;
    }
    ...
}
```

**Par stranih
ključeva ide u
zasebnu (spojnu)
tablicu**

**Inverzni
kraj**

Ograničenja na relaciju

```
@Entity
@Table(name="customer_records")
public class CustomerRecord {

    ...

    @ManyToOne
    @JoinColumn(nullable=false, unique=false)
    private Product favoriteProduct;

    ...


}
```

- x “Obična” svojstva (engl. *Property*): @Column
- x Relacija xToOne: @JoinColumn

Ograničenja na razini entiteta

```
@NamedQueries({
    @NamedQuery(name="Group.findAllSemUsers",query="select distinct ug.user from Group
as g, IN(g.users) ug where g.compositeCourseID LIKE :compositeCourseID AND relativePath
LIKE '0/%'"),
    @NamedQuery(name="Group.findForUser",query="select g from Group as g, IN(g.users) ug
where g.compositeCourseID LIKE :compositeCourseID AND relativePath LIKE '0/%' AND
ug.user=:user")
})
@Entity
@Table(name="groups",uniqueConstraints={
    // Ne mogu postojati dvije grupe s istim compositeCourseID i relativePath

    @UniqueConstraint(columnNames={"compositeCourseID","relativePath"}),
    // Roditelj ne može imati dva djeteta koja se zovu isto
    @UniqueConstraint(columnNames={"parent_id","name"})
})
@Cache(usage=CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Group implements Serializable {
    ...
}
```



Od svojstva “User parent”, gdje je primarni ključ od razreda User nazvan “id” (hibernate sam generira ovakav naziv – trebamo ga stoga koristiti)

Gdje pisati anotacije

- Anotacije mogu ići:
 - Nad definicijom članskih varijabli
 - Nad getterima
- Jednom kada ste odlučili, NE smijete ih miješati: u razredu sve mora biti anotirano na isti način

Više...

- Pogledati specifikaciju JPA, posebice sekcije 2.9 i 2.10

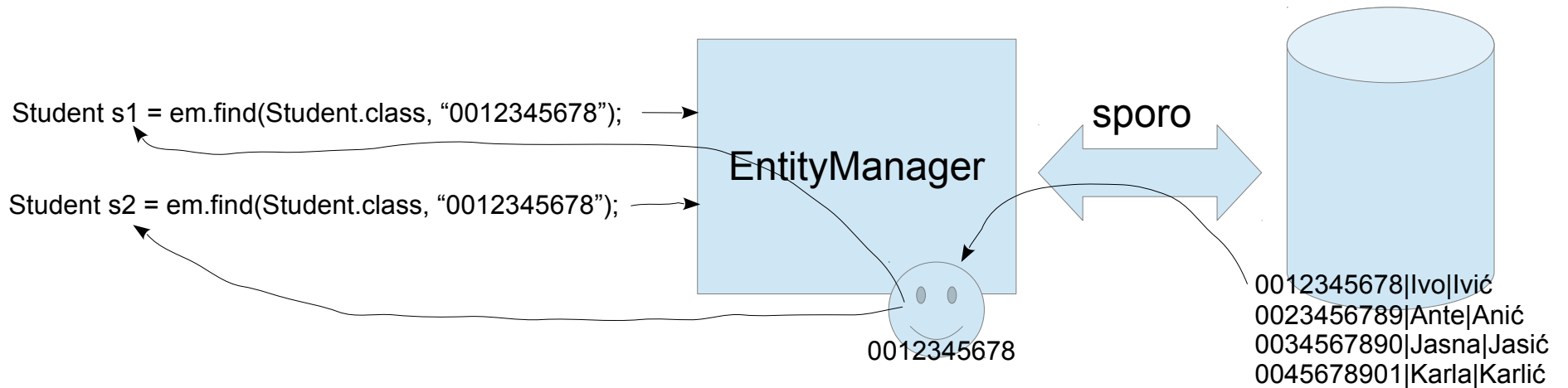
JPA

2nd-level keširanje

Keševi u JPA

- Da bismo radili s ORM-om, otvaramo “vezu”
 - Kod JPA to je primjerak razreda `EntityManager`
 - Preko njega učitavamo nove objekte, šaljemo upite i slično
 - On se ponaša kao 1st-level keš: objekt koji je jednom učitao čuva i vraća referencu na njega

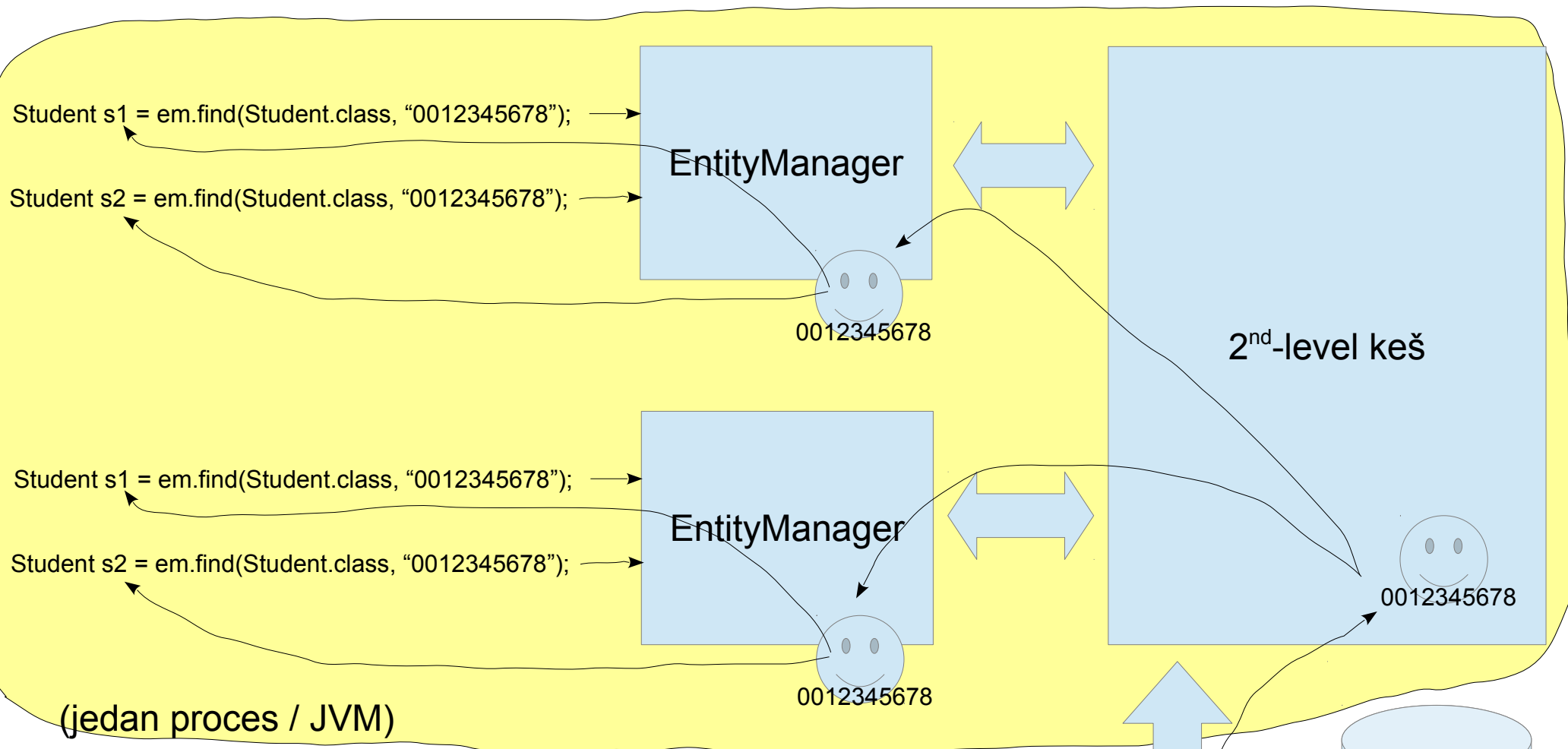
Keševi u JPA



Keševi u JPA

- Ako imamo više klijenata (primjerice, više dretvi u programu koje trebaju pristup objektima)
 - Svaka dretva koristi zaseban primjerak EntityManagera
 - Svaki EntityManager iz baze dohvaća podatke (koje je potencijalno neki drugi EntityManager već dohvatio) → neefikasno
 - Želimo na razini procesa jedan dijeljeni keš koji koriste svi EntityManageri: 2nd-level keš

Keševi u JPA



Ako klijenti samo čitaju objekte, nema problema. Ako ih žele mijenjati, 2nd-level keš se mora brinuti za sinkronizaciju.

0012345678|Ivo|Ivić
0023456789|Ante|Anić
0034567890|Jasna|Jasić
0045678901|Karl|Karlić

Keševi u JPA

- Moguća je i složenija situacija u kojoj više procesa, svaki iz više dretvi pristupa bazi
- U tom slučaju, 2nd-level keševe svih procesa treba međusobno sinkronizirati kako bi se izbjeglo da jedan proces promijeni objekt u bazi, a njegova (tada zastarjela) kopija sjedi u 2nd-level kešu nekog drugog procesa
 - Distribuirani 2nd-level keševi
 - Nećemo ih ovdje razmatrati; ali budite svjesni problematike

Keševi u JPA

- Što se može keširati u 2nd-level kešu:
 - Objekti: primjerice, objekt Cat čiji je ID=1, čiji je ID=2, čiji je ID=3, ...
 - XtoMany relacije: primjerice, relacija *kittens*, konkretno, Cat ID=1 ima u kittens objekte tipa Cat čiji su ID-evi 2, 3 i 4; Cat ID=5 ima u kittens objekte tipa Cat čiji su ID-evi 6 i 7; ...
 - Uočite da se pamte samo “vezne tablice” (terminologijom relacijske paradigme)

Keševi u JPA

- Popularna biblioteka koja nudi uslugu 2nd-level keša:
 - ehcache
 - Nju ćemo koristiti u primjerima

JPA

Demonstracijska web-aplikacija

Web-aplikacija

- Ista ideja kao i s prethodnom web-aplikacijom:
 - DAO: sučelje koje definira operacije nad slojem za perzistenciju podataka
 - JPADAOTmpl: implementacija tog sučelja tehnologijom JPA
 - DAOProvider: vezni razred prema višim slojevima
 - Odabire i stvara konkretnu implementaciju sučelja DAO
 - Nudi referencu na stvoreni objekt klijentima

Web-aplikacija

- Ista ideja kao i s prethodnom web-aplikacijom:
 - `JPAEMProvider: ThreadLocal` mapa koja nudi vezu prema ORM-u
 - Vezu otvara tek na prvi zahtjev → servleti koji je ne trebaju nikada neće inicirati stvaranje veze
 - Da bi radilo, ovaj razred treba znati kako doći do primjerka `EntityManagerFactoryja`; kako tu nije vidljiv `ServletContext`, imamo pomoćni razred `JPAEMFProvider` koji nudi referencu na objekt `EntityManagerFactory`

Web-aplikacija

- Ista ideja kao i s prethodnom web-aplikacijom:
 - JPAEMFProvider: čuva referencu na EntityManagerFactory koji će JPAEMFProvider koristiti za lijenu uspostavu veze
- Inicijalizacija: ServletContextListener koji stvara EntityManagerFactory, pohranjuje ga u globalne attribute i u JPAEMFProvider
- JPAFilter: zatvara EntityManager kad je obrada zahtjeva gotova