

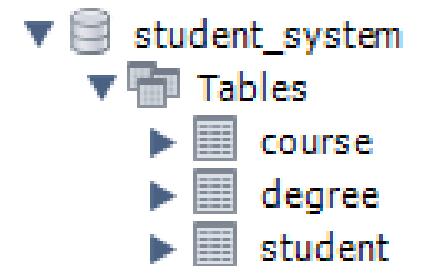
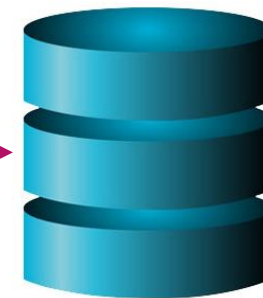
# Objektno-relaciono mapiranje

# Problem

- ▶ Kada se radi sa objektno-orijentisanim sistemima, postoji neslaganje između objektnog modela i relacione baze
- ▶ Kako izvršiti mapiranje?

```
public class Student
{
    private String name;
    private String address;
    private Set<Course> courses;
    private Set<Degree> degrees;
}
```

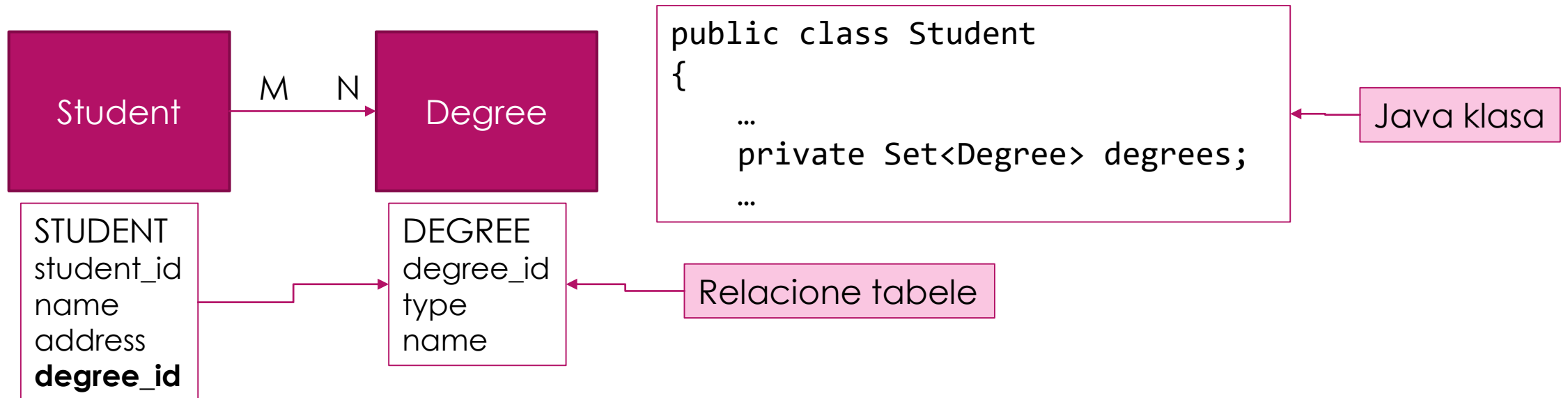
Java POJO sa atributima i asocijacijama



Relaciona baza sa tabelama i kolonama

# Problem

- ▶ Kako mapirati asocijacije između objekata?
  - ▶ Strani ključevi ne mogu da reprezentuju M:N (više na više) veze



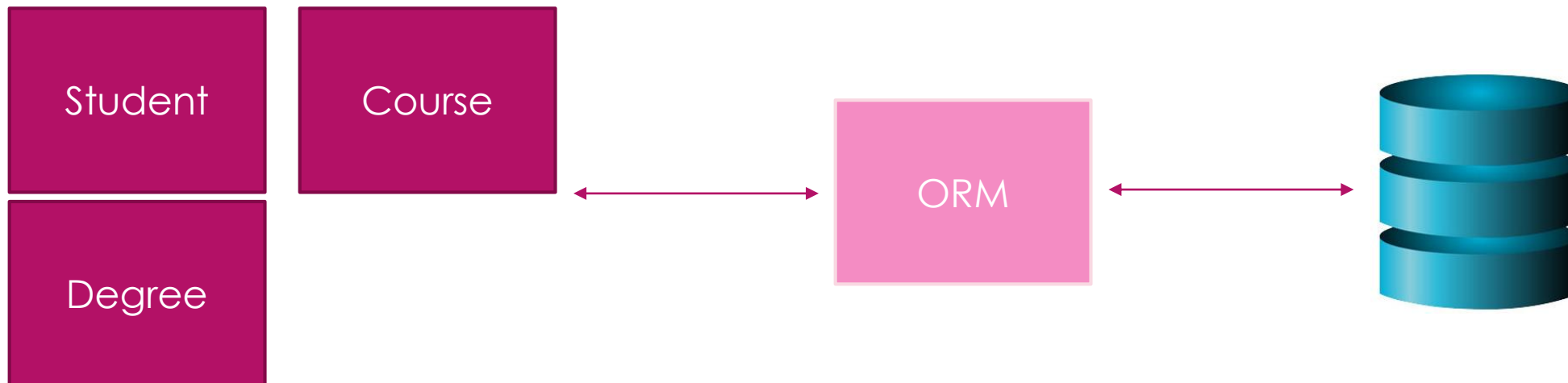
# Potreba za ORM

- ▶ Pisanje SQL metoda za konverziju pomoću JDBC
  - ▶ Zamorno i zahteva mnogo koda
  - ▶ Podložno greškama
  - ▶ Nestandardni SQL upiti vezuju aplikaciju za specifičnu bazu
  - ▶ Promene u modelu zahtevaju promene u upitima
  - ▶ Teško je predstaviti asocijacije između objekata

```
public void addStudent( Student student )
{
    String sql = "INSERT INTO student ( name, address ) VALUES ( '"' +
        student.getName() + "', '" + student.getAddress() + "' )";
    // Initiate a Connection, create a Statement, and execute the query
}
```

# Rešenje

- ▶ Korišćenje sistema za objektno-relaciono mapiranje (npr. Hibernate)
- ▶ Pružaju API za skladištenje i očitavanje Java objekata u i iz baze
- ▶ Non-intrusive: nema potrebe da se prate posebna pravila ili šabloni dizajna
- ▶ Transparentni: objektni model nije svestan konverzije

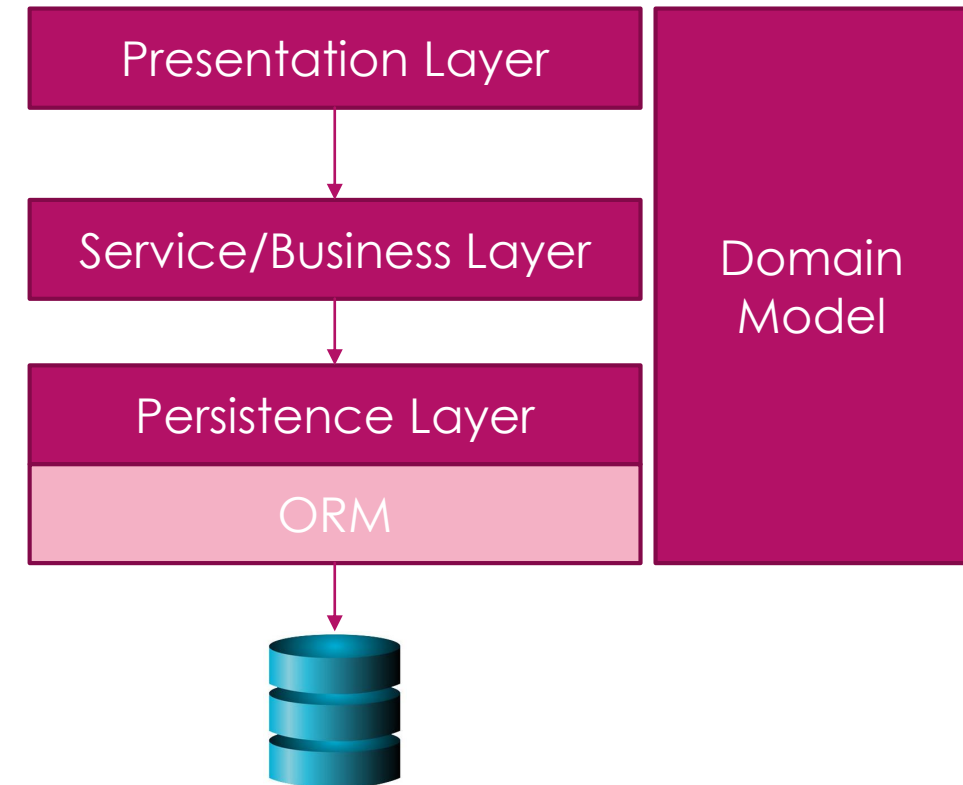


# Prednosti ORMa

- ▶ Poslovna logika pristupa objektima, a ne tabelama u bazi
- ▶ Sakriveni su detalji SQL upita od OO logike
- ▶ U pozadini se zasniva na JDBC
- ▶ Nema potreba za detaljnijom komunikacijom sa specifičnom relacionom bazom
- ▶ Entiteti su prilagođeni poslovnoj logici a ne strukturi baze
- ▶ Automatsko upravljanje transakcijama i generisanje ključeva
- ▶ Brži razvoj aplikacija

# ORM i arhitektura aplikacije

- ▶ Postoji sloj koji upravlja perzistencijom
- ▶ Nudi sloj apstrakcije između modela i baze
- ▶ JPA (Java Persistence API) omogućava snimanje POJO objekata u relacionu bazu
- ▶ API implementira neka konkretna biblioteka – Hibernate, TopLink, ...



# JPA vs Hibernate

- ▶ JPA definiše specifikaciju
  - ▶ Kako se definišu entiteti?
  - ▶ Kako se mapiraju atributi?
  - ▶ Kako se mapiraju veze između entiteta?
  - ▶ Ko upravlja entitetima?
- ▶ Hibernate je najpopularnija implementacija JPA
  - ▶ Razume mapiranja koja se prave između objekata i tabela
  - ▶ Obezbeđuje da se podaci čuvaju/čitaju iz baze na osnovu mapiranja
  - ▶ Nudi dodatne funkcionalnosti čijim korišćenjem se korisnik vezuje za jednu implementaciju i ne može da se prebaci na druge poput TopLinka



# O/R operacije: CRUD

- ▶ Uobičajene O/R operacije su:
- ▶ **C**reate – čuva (perzistuje) novi objekat u bazu
- ▶ **R**etrieve (Read) – čita objekat iz baze
- ▶ **U**ppdate – ažurira objekat već sačuvan u bazi
- ▶ **D**eleate – briše objekat iz baze

# Elementi JPA

- ▶ Entity je POJO klasa sa anotacijom `@Entity`
- ▶ Mora imati konstruktor bez parametara
- ▶ Dodatno se može iskoristiti anotacija `@Table("naziv_tabele_u_bazi")` kojom se specificira tačan naziv tabele u bazi, šema kojoj pripada
- ▶ Ako se izostavi ova anotacija, dovoljno je imati anotaciju `@Entity` i u bazi će se kreirati tabela sa nazivom klase.
- ▶ Najčešće se mapira jedna klasa na jednu tabelu
- ▶ Atributi klase mapiraju se na kolone tabele
- ▶ Parametri mapiranja opisuju se anotacijama (`@Column`)
- ▶ Enumeracije i datumi mogu se konvertovati u odgovarajući tip anotacijama `@Enumerated` i `@Temporal`
- ▶ Anotacije se vezuju za attribute ili get metode

# Elementi JPA

- ▶ Svaki entitet ima svoj surogat ključ (**@Id**) (ili prirodni ključ (**@NaturalId**))
- ▶ Strategija generisanja ključeva može se eksplicitno podesiti (**@GeneratedValue**):
  - ▶ AUTO - generisanje ključeva se oslanja na perzistencionog provajdera da izabere način generisanja (ako je u pitanju Hibernate, selektuje tip na osnovu dijalekta baze, za najpopularnije baze izabraće IDENTITY)
  - ▶ IDENTITY - inkrementalno generisanje ključeva pri svakom novom insertu u bazu
  - ▶ SEQUENCE - koriste se specijalni objekti baze da se generišu id-evi
  - ▶ TABLE - postoji posebna tabela koja vodi računa o ključevima svake tabele
  - ▶ Više informacija na [Java Persistence/Identity and Sequencing](#)

# Elementi JPA

- ▶ 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!

# Tipovi veza između entitija

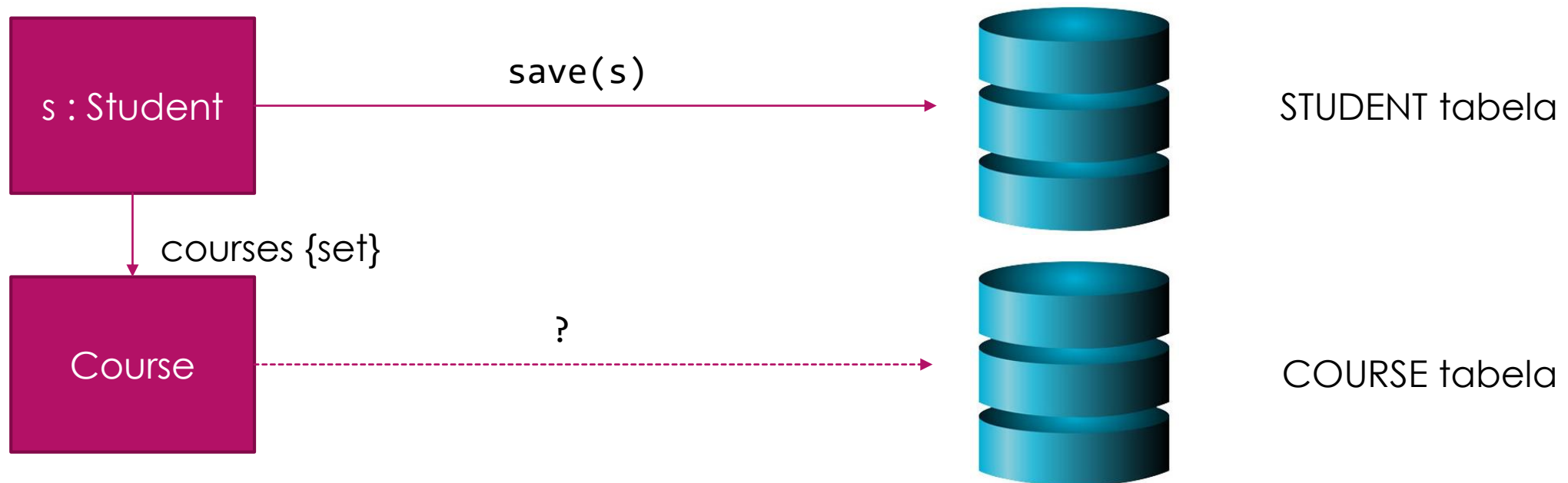
- ▶ Posmatramo dve klase, A i B, koje su u vezi
- ▶ Veza tipa 1:1
  - ▶ klasa A sa atributom tipa B, anotacija `@OneToOne`
  - ▶ klasa B sa atributom tipa A, anotacija `@OneToOne`
- ▶ Veza tipa 1:N
  - ▶ 1-strana ima anotaciju `@OneToMany`, tip atributa je `Set<B>` ili `List<B>`
  - ▶ N-strana ima anotaciju `@ManyToOne`, tip atributa je A
  - ▶ N-strana obično ima i anotaciju `@JoinColumn` koja opisuje uslov povezivanja
  - ▶ Opciono se može iskoristiti atribut `mappedBy` da se naznači ko je vlasnik veze
  - ▶ U bazi će se u tabeli B kreirati dodatna kolona koja će sadržati id objekata tipa A kao strani ključ
  - ▶ Ako se izostavi `mappedBy` kreiraće se međutabela koja će sadržati 2 kolone - id klase A i id klase B
- ▶ Veza tipa M:N
  - ▶ M-strana ima anotaciju `@ManyToMany`, tip atributa je `Set<B>` ili `List<B>`
  - ▶ N-strana ima anotaciju `@ManyToMany`, tip atributa je `Set<A>` ili `List<A>`
  - ▶ Kako u ovom slučaju nastaje treća tabela, mora se naznačiti unutar `@JoinTable` koje kolone će se referencirati pomoću `@JoinColumn`

# Unidirekcione i bidirekcione veze

- ▶ Unidirekcionalna (jednosmerna) veza: klasa A „vidi“ klasu B, a klasa B „ne vidi“ klasu A
- ▶ Bidirekcionalna (dvosmerna) veza: klasa A „vidi“ klasu B i obrnuto
- ▶ Jednosmerna veza se pravi izostavljanjem odgovarajućeg atributa u klasi

# Šta su kaskadne operacije?

- ▶ Kada se čuvaju/ažuriraju/brišu objekti iz baze, da li se i povezani objekti isto čuvaju/ažuriraju/brišu?




# Cascading u JPA

- ▶ JPA kod anotacija za veze nudi i atribut **cascade**
  - ▶ CascadeType podešen na **ALL** dozvoljava da se prilikom svakog čuvanja, izmene ili brisanja studenta čuvaju, menjaju ili brišu i kursevi
  - ▶ To znaci da ne moraju unapred da se čuvaju kursevi pa onda povezuju sa studentom
  - ▶ **orphanRemoval** podešen na **true** će obezbediti da se kursevi izbrišu iz baze kada se izbrišu iz kolekcije kurseva kod studenta

```
@Entity
public class Student {

    @OneToMany(mappedBy=„student“, cascade=ALL)
    private List<Course> courses;
```

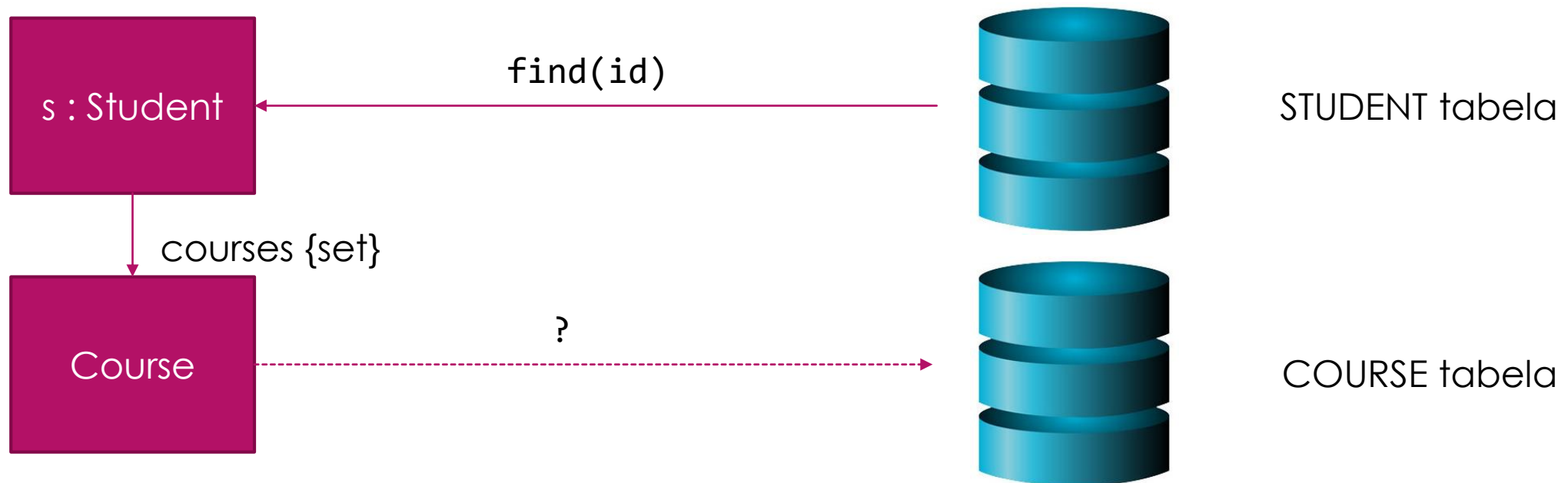
NONE  
PERSIST  
REFRESH  
REMOVE  
ALL





# Šta je Eager i Lazy Fetching?

- Pri očitavanju objekata iz baze, kada se kreiraju povezani objekti?



# Fetching u JPA

- ▶ Atributom **fetch** može se podešavati način dobavljanja povezanih entiteta
- ▶ Opcije su **EAGER** i **LAZY**
- ▶ **FetchType** odlučuje da li će se učitati i sve veze sa odgovarajućim objektom čim se inicijalno učitava sam objekat ili neće
- ▶ Ako je FetchType EAGER, učitavaće se sve veze sa objektom odmah, a ako je FetchType LAZY, učitavaće se tek pri eksplicitnom traženju povezanih objekata (pozivanjem npr. metode `getCourses`)
- ▶ Više informacija na [Hibernate Eager vs Lazy Fetch Type](#)

```
@Entity  
public class Student {
```

```
@OneToMany(mappedBy="student", fetch=LAZY)  
private List<Course> courses;
```

LAZY  
EAGER



# Generisanje šeme baze

- ▶ Na osnovu anotacija u modelu, Hibernate može da izgeneriše i popuni šemu baze bez eksplicitnog pisanja SQL naredbi
- ▶ U application.properties fajlu je potrebno podesiti način generisanja:
  - ▶ `spring.jpa.hibernate.ddl-auto`
    - ▶ `none` – ništa se neće generisati
    - ▶ `validate` – validira šemu baze
    - ▶ `update` – ažurira šemu baze
    - ▶ `create` – dropuje šemu pa je kreira
    - ▶ `create-drop` – kreira šemu i dropuje je kada se aplikacija ubije

# Spring Data Repository

- ▶ Drugi deo JPA, pored sistema za mapiranje, je EntityManager API za pristup objektima i izvršavanje upita
- ▶ Kako bi se ceo proces uprostio uveden je Repository koncept
- ▶ Repository funkcioniše kao adapter
- ▶ Cilj Spring Data Repository apstrakcije je da značajno smanji količinu koda koji je potreban za implementaciju DAO sloja i osnovnih CRUD operacija
- ▶ Potrebno je samo kreirati interfejse koji implementiraju postojeće Spring Data interfejse
- ▶ Na taj način Spring kontejner će naći odgovarajuće interfejse i dodati ih kao Spring beanove kreiranjem proxy objekata i delegirati pozive originalnoj implementaciji

# Spring Data Repository

- ▶ Repository je interfejs koji dozvoljava Spring Data infrastrukturi da prepozna korisnički definisane repozitorijume
  - ▶ sam interfejs anotira se sa `@Repository`
  - ▶ `CrudRepository` dodaje osnovne metode poput čuvanja, brisanja i pronalaženja entiteta
  - ▶ `PagingAndSortingRepository` nasleđuje `CrudRepository` i dodaje metode za pristup entitetima stranicu po stranicu i njihovo sortiranje
  - ▶ `JpaRepository` nasleđuje `PagingAndSortingRepository` i dodaje JPA specifične funkcionalnosti poput `flush` i `deleteInBatch`.
- ▶ Različiti interfejsi koji se mogu iskoristiti dozvoljavaju manipulaciju različitim vrstama metoda koje trebaju biti podržane
  - ▶ npr. repozitorijum treba da bude samo *readonly* ili treba da ima `findAll()` metodu koja pritom treba da vraća samo deo rezultata ograničen pomoću *Pageable* interfejsa

# Spring Data Repository

- ▶ Spring radni okvir se oslanja na filozofiju konvencija ispred konfiguracije
- ▶ Kreiranjem fajlova na odgovarajućim lokacijama i odgovarajućeg naziva rešice deo konfiguracije
- ▶ Takođe, pisanje klasa i metoda na odgovarajući način aktiviraće željene operacije bez pisanja suvišnog koda
- ▶ Repository interfejsi mogu imati samo interfejsne metoda napisane u odgovarajućem formatu bez dodatne konfiguracije koje će predstavljati upit
  - ▶ `findAll()` će formirati upit sličan ovome: `SELECT * FROM table_name`
  - ▶ `findById(long id)` će formirati upit sličan ovome: `SELECT * FROM table_name WHERE table_name.id = id`

# Spring Data Repository

- ▶ Ako su upiti malo složeniji, prosto navođenje naziva metoda može da postane prilično nečitljivo
- ▶ Alternativa je pisanje stvarnih upita korišćenjem JPQL/HQL sintakse
- ▶ Metoda može da ima proizvoljan naziv ali se mora anotirati anotacijom `@Query`

```
@Query("select h.city as city, h.name as name, avg(r.rating) as averageRating "  
      + "from Hotel h left outer join h.reviews r where h.city = ?1 group by h")  
Page<HotelSummary> findByCity(City city, Pageable pageable);
```

# Spring Data Repository

- ▶ Treći način je pisanje upita unutar klase modela u sklopu anotacije `@NamedQuery`
- ▶ U repozitorijumu se navodi samo interfejs metode proizvoljnog naziva

```
@NamedQuery(name="Product.findByPrice", query="select name, origin, price from  
Product p where p.price = ?1")  
public class Product implements Serializable {  
...  
}
```

```
public interface ProductRepository extends Repository<Product, Long> {  
    //Primer NamedQuery koji se nalazi iznad klase Product  
    List<Product> findByPrice(long price);  
}
```