CS 416
Web Programming

Java Persistance API (JPA) Cont.
Data and app design

Dr. Williams
Central Connecticut State University

Agenda

- Review highlights of JPA
- Application design – integration of data representation
  - JPA object model
  - Underlying database structure
  - To JPA or not to JPA

Java Persistence API - JPA

- Tool for linking object model to database representation
- Allows Beans to be mapped to tables
- Allows objects to be persisted rather than fields (add, update, delete)
- Allows queries to return lists of objects rather than records
- Allows object relationships to be mapped
- **Partially automated mapping does not mean you don’t need to know DB design**

Entity highlights

- Entity beans
  - Must have @Entity annotation
  - No mapping specified will map (or create) bean directly to table and columns of same names
  - Must annotate primary key, possible to have compound primary key through primary key class
  - Option to have primary key generated on insert

DB persistence interaction highlights

- EntityManager is controller of all things JPA
  - Controls persistence and retrieval
- All persistence must be enclosed in transaction
- Insert of bean
  - Create new instance of bean class populate `entityManager.persist(person);`
- Update of bean
  - Must retrieve bean from DB prior to update then `entityManager.persist(person);`
- Deletion
  - Must retrieve bean from DB prior to deletion then
  - `entityManager.remove(person);`

DB retrieval by primary key

- Retrieval by primary key requires find including class of entity being retrieved + primary key

```
Person p =
entityManager.find(Person.class, id);
```

- With composite primary keys
  - Requires entity to overide equals and hashcode
  - Specify key class at top of entity:

  ```
  @IdClass(value = EmailPK.class)
  ```

  - Find requires key class

```
Email e = entityManager.find(Email.class,
emailPK);
```

DB query interaction highlights

- SQL query syntax:

```
select p from Pet p where p.type = :type
```

  - Table of entities given symbolic name like *p* then selection return is that name
  - Parameter syntax slightly different from JDBC
  - Setting parameters

```
    query.setParameter("type", type);
```

- Retrieval of list of objects

```
List matchingPets = query.getResultList();
```

Entity relationships

- JPA allows you to have relationships between entity classes
  - One-to-one
  - One-to-many/many-to-one
  - Many-to-many
- Once relationships mapped retrieval of objects will also bring back related entity objects
- Persistence requires persisting each object individually however

Entity relationships one-to-one

- One-to-one mapping

```
public class Login implements Serializable{
    @Id
    private String id;

    // No foreign key attribute declaration
    // Integer personId;

    // Maps to foreign key column
    @OneToOne
    @JoinColumn(name="personid")
    private Person person;
```

Entity relationships one-to-one cont.

- One-to-one mapping

```
public class Person implements Serializable
{
    @Id
    private Integer id = null;

    // Maps to the name the class is called
on the Login class
    @OneToOne(mappedBy="person")
    private Login login;
```

Entity relationships one-to-many/many-to-one

- One-to-many mapping

```
public class Person implements Serializable{
    @Id
    private String id;

     // Add the mapping to many addresses
     // specifying the field
    // that maps back to this bean
    @OneToMany(mappedBy="person")
    private Set<Address> addresses;
```

Entity relationships one-to-many/many-to-one

- Many-to-one mapping

```
public class Address implements Serializable {
    @Id
     private Integer id;

    @ManyToOne
    @JoinColumn(name = "personId")
    private Person person;
```

Entity relationships many-to-many

- many-to-many mapping – the ugly

```
public class Phone implements Serializable{
    @Id
    private String number;

    @ManyToMany
    @JoinTable(name="PersonPhone",
    joinColumns=@JoinColumn(name="phoneId",
    referencedColumnName="number"),
inverseJoinColumns=@JoinColumn(name="personId",
referencedColumnName="id"))
    private Set<Person> persons = new HashSet();
```

Entity relationships many-to-many

- many-to-many mapping – the clean

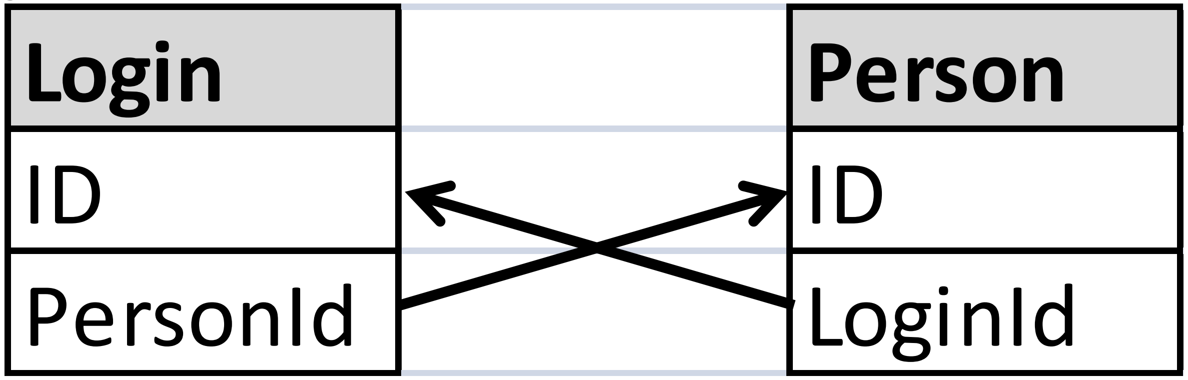
```
public class Person implements Serializable{
    @Id
    private String id;

    @ManyToMany(mappedBy="persons")
    private Set<Phone> phones;
```

JPA relationship design

- Important to notice while JPA will manage mapping once it is established it is up to you to figure out DB structure for mapping!
- To create a good JPA model often requires more considerations than just building quick object model
  - Consider whether relationship is strong enough to warrant DB connection or just the object(s) are used in a calculation in which case they should just be parameters

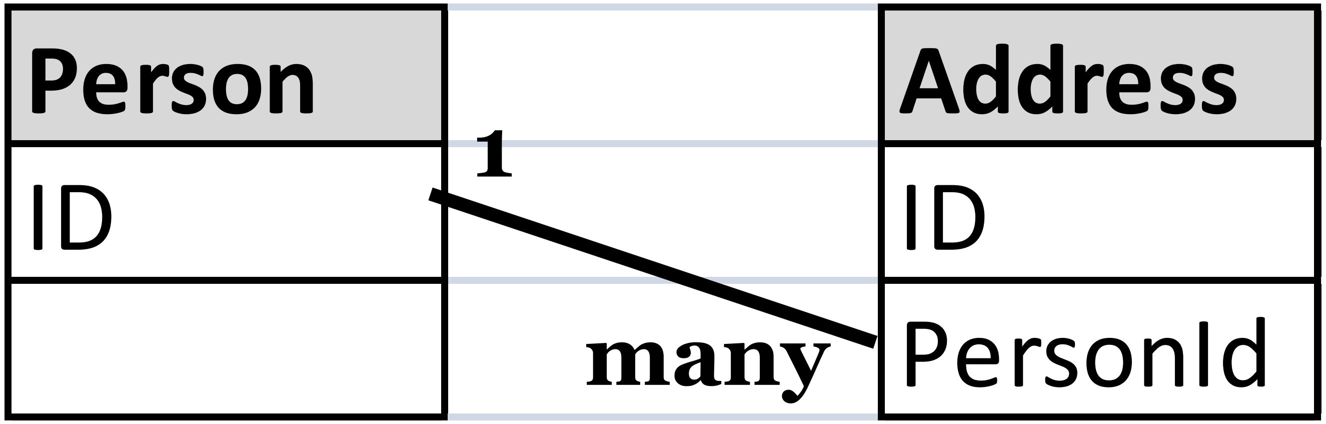
JPA relationship design – one-to-one

- For one-to-one relationships – do they really need to be separate objects/tables?
- If relationship is needed in order to enforce DB referential integrity of one-to-one you will need foreign key on both sides of the relationship
- In practice it is common in DB design to have FK on only one side



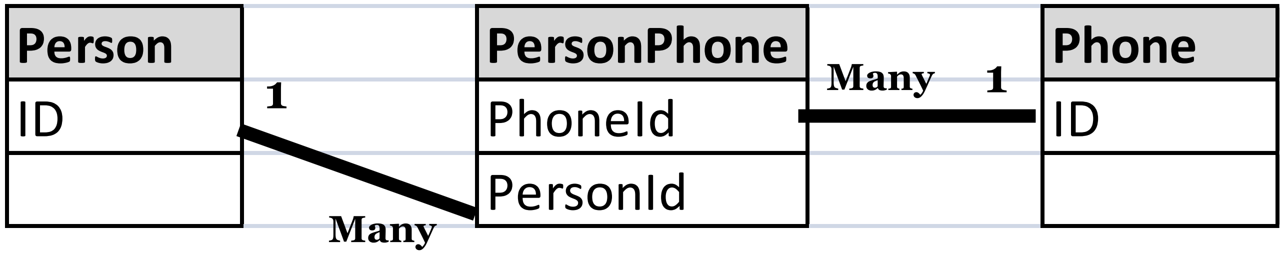
JPA relationship one-to-many/many-to-one

- Unlike a basic object model, for entity object model the “many” need to be in a Set and each must be uniquely defined
- Also for full power of JPA object relationship is bidirectional, which is often avoided in normal object model design
- From DB design perspective relationship is completely specified on many side through FK



Many-to-Many

- Like one-many, means every object in Sets on both side must be uniquely identified
- To create a many to many relationship a **join table** is required. The table structure to represent this type of relationship is:



XML considerations

- As was discussed earlier when we covered XML structure, XML design needs to take into account purpose
- Just because we retrieve full JPA objects with their relationships doesn’t mean they all belong in the XML
- Further same objects may have many different XML presentations
  - Remember based on perspective of interest
  - Only information needed
- *Note also applies to JSON, but particularly relevant if using @XmlRootElement annotations*

To JPA or not to JPA

- For the most part using JPA in your application (insert, update, delete, finds) will make your code cleaner and easier to maintain
  - Also as we will see with JSF using JPA has HUGE advantages
- <u>HOWEVER</u>, using the JPA Entity Manager object operations isn't always the best choice
  - Mass updates/deletes
  - Complex queries

To JPA or not to JPA cont.

- Mass updates/deletes

```
Query query = entityManager.createQuery("update
MyEntity set year = :newyear where year = :oldyear");
query.setParameter(“newyear", “Sophmore”);
query.setParameter(“oldyear", “Freshman”);
int recordsAffected = query.executeUpdate();
```

- Goes against spirit of JPA, but performs MUCH better
- Alternatively could do same through JDBC

- **Either way** – these invalidate JPA’s optimistic locks so all objects affected **must** be re-retrieved through Entity Manager

To JPA or not to JPA cont. When to use JDBC

- Complex queries
  - JPA is designed for simple queries and bringing back object model
  - JPA is not designed for
    - Complex joins where there are criteria on multiple tables
    - Queries that involve calculations, ordering, grouping
- Use of results simpler without object model
  - Ex. Selection of fields from multiple tables for a report – easy to do with SELECT statement, but requires work to accomplish same thing traversing object model