

# CS 416

## Web Programming

### Java Persistence API (JPA) Data and app design

Dr. Williams  
Central Connecticut State University

# Agenda

- JDBC approach review
- Introduce Java Persistence API (JPA)
- Application design – integration of data representation
  - JPA object model
  - Underlying database structure
  - To JPA or not to JPA

**NOTE: Matches to the GitHub code demos  
Lec15DemosV2**

# Review JDBC database connectivity

- Use JDBC in servlet to select and display all people that match the specified first name
  - ReviewJDBCPeopleSearch
  - ReviewJDBCGetNamesServlet

# Java Persistence API

- Java Persistence API (JPA)
  - Encapsulates “model” part of MVC
  - Simplifies retrieval and population of objects
  - Simplifies persisting objects to the database
- JPA allows a class to be mapped to a database table
  - Records can be retrieved as the mapped objects
  - Updates to objects can easily be saved off to their respective table
- Result is rather than using record sets can use objects

# Java EE Entities

- JPA's mechanism for mapping a table to an object is to create *Entity* objects
- A entity object is a normal java bean that has annotations to indicate it is an Entity and how to map the object to a database table
- Adding @Entity above the class declaration lets the application server know the class maps to the database

# Creating the mapping

- Mapping the bean to the database is done through annotation
- At the top of the class an annotation is made to indicate the table the bean is tied to:

```
@Table (name="Person")
```

- With JPA if the bean name matches the table name this annotation isn't required unless it maps to a different table than the name of the class

## Creating the mapping cont.

- Within, the bean annotation is used to map each class attribute to the appropriate field in the table
- Attributes to columns is mapped using the annotation:

```
@Column(name = "fullName")  
private String personName;
```

Like with the table, if the attribute name matches the column name the annotation isn't needed

# Primary keys

- An annotation is required to identify the primary key of the table

@Id

```
Private Integer id;
```



# Annotations example

```
@Entity
@Table(name="Person")
public class Person implements Serializable {
    @Id
    @Column(name="ID")
    private Integer ID = null;

    @Column(name="firstname")
    private String firstName = null;

    @Column(name="lastname")
    private String lastName = null;

    @Column(name="age")
    private Integer age = null;
```

# Persisting beans

- All interaction between the Entity classes and the database is controlled by the EntityManager
- The EntityManager provides functionality to lookup, add, update, and delete Entities
- To function an Entity manager must be tied to a database connection which is specified as a persistence unit

# Persistence unit

- A persistence unit is a mapping of a persistence unit name to a connection pool
- This configuration is specified in the persistence.xml file

```
<persistence version="2.0" ...>
  <persistence-unit name="Lec17DemosPU" transaction-
type="JTA">
    <jta-data-source>jdbc/Lect16DB</jta-data-source>
    <exclude-unlisted-classes>>false</exclude-unlisted-
classes>
    <properties>
    </properties>
  </persistence-unit>
</persistence>
```

# Creating Persistence Unit in NetBeans

- On the project right-click select New
- Under category select Persistence
- Under file types select Persistence Unit
- For the data source either select the JNDI name you gave the data source or you may need to select new data source and add it (jdbc/Lect8aDB)
- Click finish

# EntityManager and UserTransaction

- Once the persistence unit has been defined it can be used to create an EntityManagerFactory that can then create a EntityManager

```
@PersistenceUnit(unitName="Lec17DemosPU")
```

```
private EntityManagerFactory  
    entityManagerFactory;
```

- Your application also needs an instance of a UserTransaction to control commits and rollbacks
  - If you are tying multiple things together you can get a generic transaction from the app server

```
@Resource
```

```
private UserTransaction userTransaction;
```

- Alternatively if just the EntityManager, just use it's transaction

```
entityManager.getTransaction()
```

# Record creation

- To create a record with JPA all that is involved is creating the object to be stored and telling the Entity Manager to persist it.

```
// Create a new Person bean
Person person = new Person();
person.setID(1);
person.setFirstName("John");
person.setLastName("Doe");
person.setAge(24);
```

```
// Persist it to the database to create
the record
entityManager.persist(person);
```

*Refer to JPAAAddPerson*

# Letting JPA do the work

- Mapping to an existing table is great if you want complete control of table creation, however you can also let JPA do the work for you
- If you persist an object that has no table, JPA will automatically create the table with the attribute names as the column names
- Rather than handling primary keys on our own. Let the system handle them for us by adding `@GeneratedValue`

# Annotations example

`@Entity`

```
public class Pet implements Serializable {
```

```
    @Id
```

```
    @GeneratedValue
```

```
    private Integer ID = null;
```

```
    private String name = null;
```

```
    private String type = null;
```

```
    private Integer age = null;
```

When class is deployed Pet table and sequence for generating value will be created



# 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

# JPA queries

There are two ways to retrieve records (or in this case objects) from the database

- By primary key

```
Person person = entityManager.find(Person.class, 1);
```

- Or by querying

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

Rather than retrieving record sets for you to process, it brings back prepopulated objects

# 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();
```

# JPA query

- Creating a JPA query is similar to a prepared statement with some minor differences

The syntax is:

```
String queryString = "Select p from Person p  
    where p.firstName = :firstName";
```

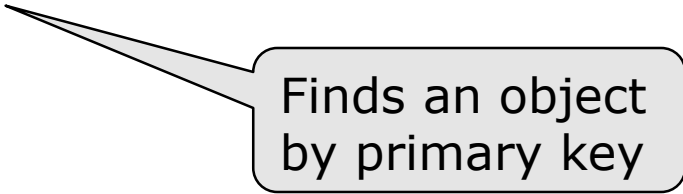
```
Query query =  
    entityManager.createQuery(queryString);  
query.setParameter("firstName", "Bob");  
List matchingPeople = query.getResultList();
```

*Refer to NamesLikeServlet*

# Record update

- To update a record a object instance that represents a record is retrieved from the database. The field is updated and the object is persisted

```
// Retrieve person from database as an object  
Person person = entityManager.find(Person.class, 1);
```



Finds an object  
by primary key

```
//update the object and write it back to the database  
person.setFirstName("Jon");  
entityManager.persist(person);
```

*Refer to JPASimpleUpdatePerson*

# Record Delete

- Like update a instance, to delete a record needs to be retrieved then EntityManager is called to remove the object

```
// Retrieve person from database as an object  
Person person = entityManager.find(Person.class, 1);
```

```
// Delete a object from the database  
entityManager.remove(person);
```

*Refer to JPADeletePerson*

# 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);`

## More examples

1. JPAAAddPet to insert the passed pet information into the database
2. JPAUpdatePerson to search for ALL people with the specified first and last name and update them all to have the new last name
3. JPAPetSearch to retrieve all pets of the specified type – it should then enhance the request and forward to the PetDisplay.jsp to display the pets. Modify that JSP to loop through and display their names and ages



# JPA summary

- 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 relationships

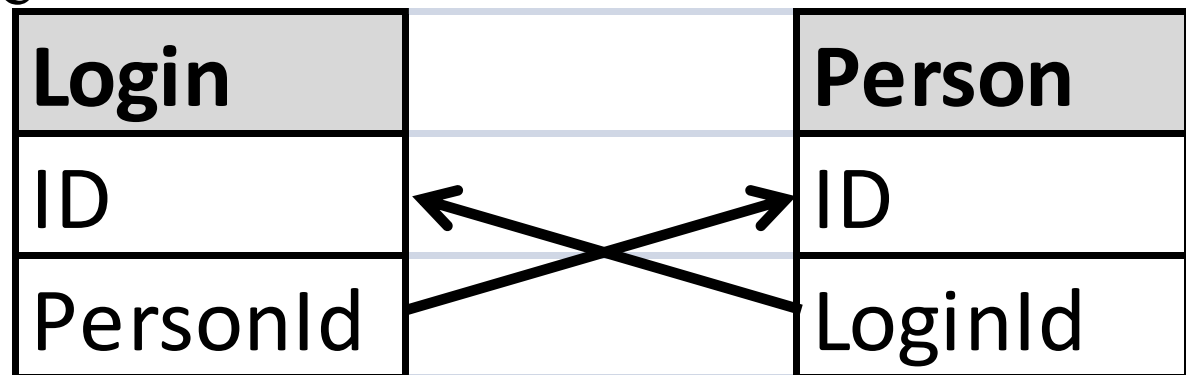
- JPA allows you to have relationships between entity classes
  - One-to-one
  - One-to-many/many-to-one
  - Many-to-many
- Once relationships mapped retrieval of objects will also bring back related entity objects
- Persistence requires persisting each object individually however

# 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



# 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.

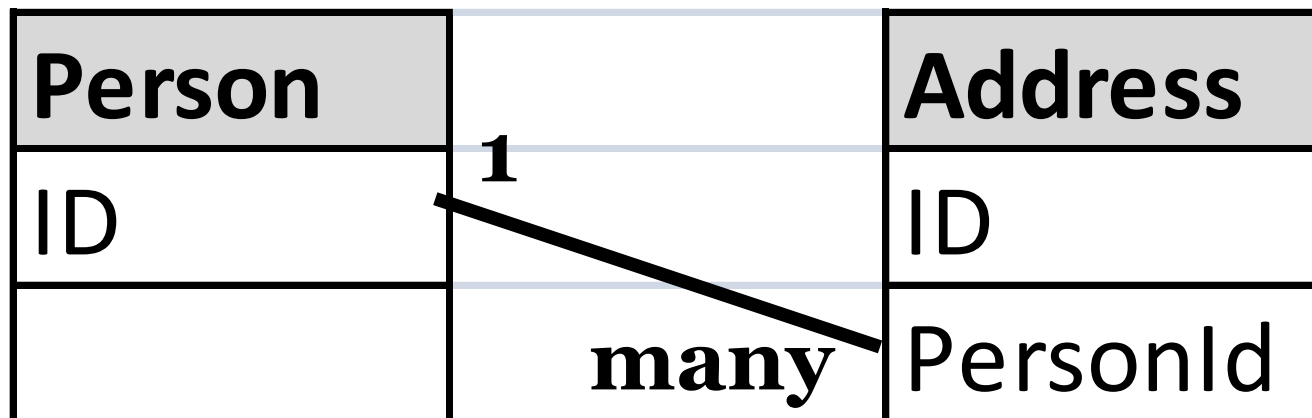
- One-to-one mapping

```
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;
```

# 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



# 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;
```



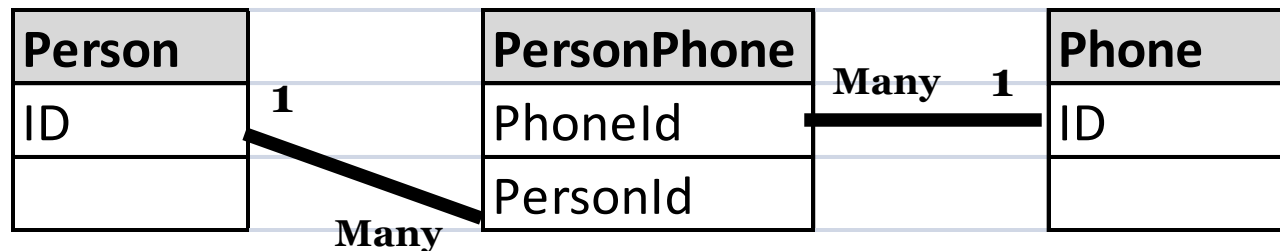
# Entity relationships one-to-many/many-to-one

- **Many-to-one mapping**

```
public class Address implements Serializable {  
    @Id  
    private Integer id;  
  
    @ManyToOne  
    @JoinColumn(name = "personId")  
    private Person person;  
}
```

# 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:



# 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;
```

# 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
- HOWEVER, 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", "Sophomore");  
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