Java persistence

The Next Generation

Chris Maki
chrismaki@mac.com

Agenda

- JPA Overview
- Entities
- Packaging
- EntityManager
- Java EE in a Java SE World
- JPQL

About Me

- Working with persistence technologies since 1992
  - NeXT DBKit
  - Apple/NeXT Enterprise Objects Framework
  - Built my own ORM solution using C++
  - Hibernate, JPA
- Technical editor, *Mastering JavaServer Faces (Wiley)*
- Author, *JPA 101 - Java Persistence Explained*
- President, Utah Java User Group

Agenda

- **JPA Overview**
- Entities
- Packaging
- EntityManager
- Java EE in a Java SE World
- JPQL

JPA Overview

Do we really need another standard?

JPA History

- JSR-220 (Enterprise JavaBeans 3.0)
  - Goal: simplify entity beans
  - Java Community requested Java EE and Java SE support
- JSR-220 became to specifications
  - JSR 220: EJB 3.0 Simplified API
  - JSR 220: Java Persistence API
  - Reference Implementation under Project GlassFish
- Leverages Hibernate, TopLink, and JDO communities

Why Another Standard?

- Many successful proprietary solutions, both commercial (e.g. TopLink) and open source (e.g. Hibernate)
- There is no community accepted persistence standard (JDO? EJB Entity Beans?)
- With a standard you have:
  - Vendor independence
  - Creates competition
  - More choices for the user

JPA Features

- Supports a POJO persistence model
- Rich inheritance support
- Polymorphic associations and queries
- Support for annotation or XML based mapping
- Java SE and Java EE support
- Supports pluggable persistence providers

Agenda

- JPA Overview
- **Entities**
- Packaging
- EntityManager
- Java EE in a Java SE World
- JPQL

Entities

good-bye XDoclet

Entity Characteristics

- Are POJO's identified as an Entity
- Can be a concrete or abstract class
- Supports mixed inheritance, superclass and/or subclass does not have to be an entity
- Has a persistent id
- Serializable - if passing as a detached object (no need for a DTO)
- Can have persistent and transient state
- Supports polymorphic relationships

Entity State

- Persistent fields/properties can be simple types
  - Java primitives and Wrapper classes, Serialized types (e.g. BigDecimal, or your own) or enums
- or complex types
  - arrays (e.g. byte[]), embedded types, or other entities either as a reference or a collection
  - support Lazy or Eager fetching
- Transient fields, with either the `transient` keyword or `@Transient` annotation.

Annotations

- If you choose to use annotations, they may be placed on a field or property
- You cannot mix field level and property level annotations within the same class hierarchy

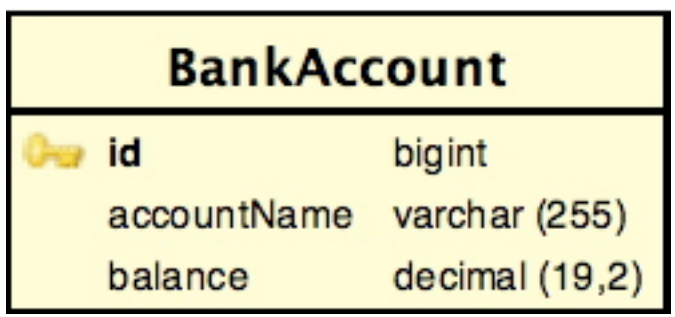
Examples

POJO

```
public class BankAccount {


    private Long id;
    private String accountName;
    private BigDecimal balance;
}
```

Simple Entity

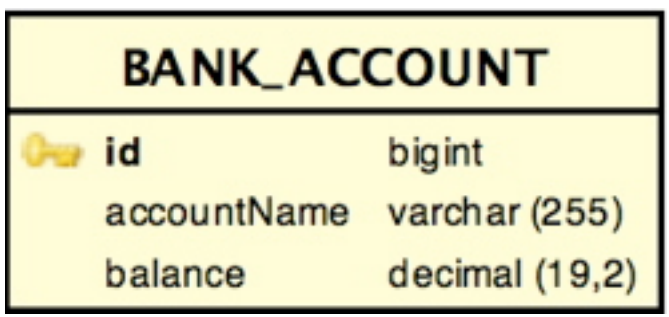
```
@Entity
public class BankAccount {

    @Id
    private Long id;
    private String accountName;
    private BigDecimal balance;
}
```



A class is mapped to a table, fields are mapped to columns

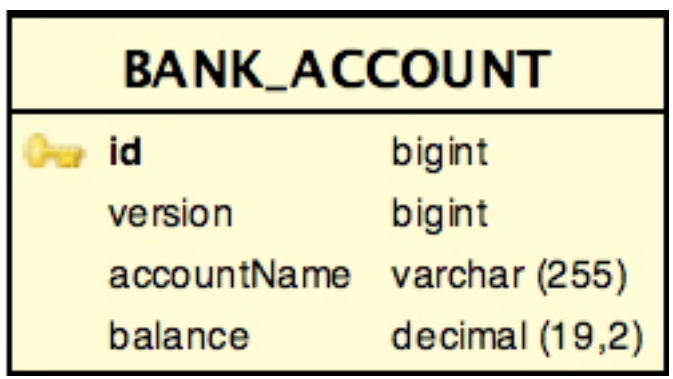
@Table

```
@Entity
@Table(name="BANK_ACCOUNT")
public class BankAccount {

    @Id
    private Long id;
    private String accountName;
    private BigDecimal balance;
}
```



@Version

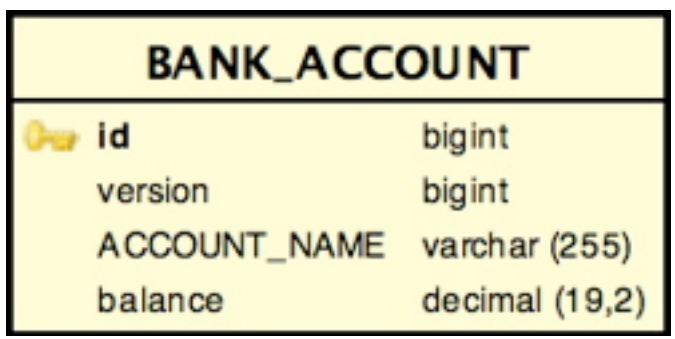
```
@Entity
@Table(name="BANK_ACCOUNT")
public class BankAccount {

    @Id
    private Long id;

    @Version
    private Long version;
    private String accountName;
    private BigDecimal balance;
}
```



You can place the `Version` annotation on an int, short, long, or timestamp field

@Column

```
@Entity
@Table(name="BANK_ACCOUNT")
public class BankAccount {
    @Id
    private Long id;
    @Version
    private Long version;

    @Column(name="ACCOUNT_NAME")
    private String accountName;
    private BigDecimal balance;
}
```



The column annotation allows you to define the length, precision, scale, etc.

@Lob

```
@Entity
@Table(name="BANK_ACCOUNT")
public class BankAccount {
    @Id
    private Long id;
    @Version
    private Long version;
    @Column(name="ACCOUNT_NAME")
    private String accountName;
    private BigDecimal balance;

    @Lob
    private String description;
}
```

BANK_ACCOUNT	
id	bigint
version	bigint
ACCOUNT_NAME	varchar (255)
balance	decimal (19,2)
description	text (65535)

Default database column size for a String field/property is 255

@Lob

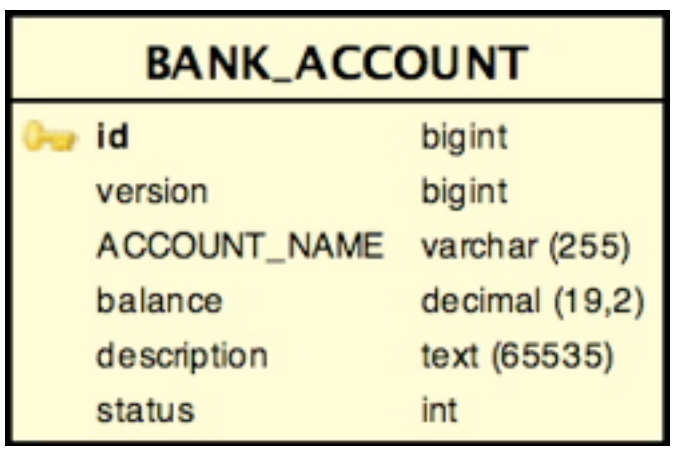
```
@Entity
@Table(name="BANK_ACCOUNT")
public class BankAccount {
    @Id
    private Long id;
    @Version
    private Long version;
    @Column(name="ACCOUNT_NAME")
    private String accountName;
    private BigDecimal balance;

    @Lob @Basic(fetch=LAZY)
    private String description;
}
```

BANK_ACCOUNT	
id	bigint
version	bigint
ACCOUNT_NAME	varchar (255)
balance	decimal (19,2)
description	text (65535)

Default database column size for a String field/property is 255

@Enumerated

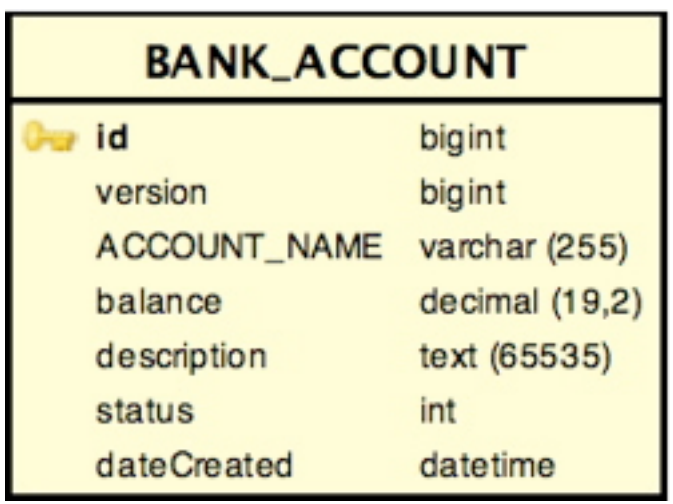
```
@Entity
@Table(name="BANK_ACCOUNT")
public class BankAccount {
    @Id
    private Long id;
    // ...

    @Enumerated(EnumType.ORDINAL)
    private AccountStatus status;
}

public enum AccountStatus {
   PENDING, APPROVED, ACTIVE;
}
```



EnumType definition: public enum EnumType {ORDINAL, STRING}

@Temporal

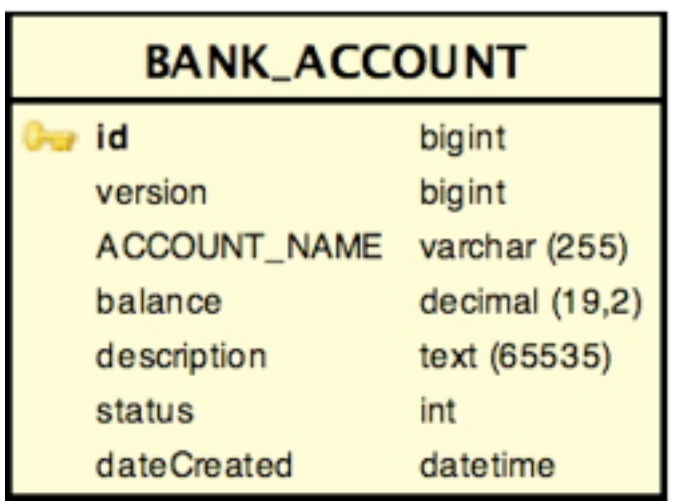
```
@Entity
@Table(name="BANK_ACCOUNT")
public class BankAccount {
    @Id
    //...
    @Enumerated(EnumType.ORDINAL)
    private AccountStatus status;

    @Temporal(TemporalType.TIMESTAMP)
    private Date dateCreated;
}
```



TemporalType definition: public enum TemporalType {DATE, TIME, TIMESTAMP}

@Transient

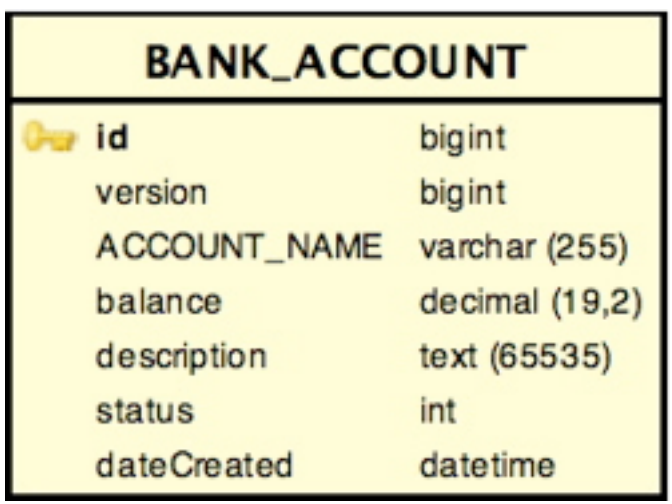
```
@Entity
@Table(name="BANK_ACCOUNT")
public class BankAccount {
    @Id
    //...
    @Enumerated(EnumType.ORDINAL)
    private AccountStatus status;
    @Temporal(TemporalType.TIMESTAMP)
    private Date dateCreated;

    @Transient
    private BigDecimal calculatedValue;

}
```



You can use either the annotation @Transient, of the Java keyword transient

transient keyword

```
@Entity
@Table(name="BANK_ACCOUNT")
public class BankAccount {
    @Id
    //...
    @Enumerated(EnumType.ORDINAL)
    private AccountStatus status;
    @Temporal(TemporalType.TIMESTAMP)
    private Date dateCreated;


    private transient BigDecimal calculatedValue;
}
```



You can use either the annotation @Transient, of the Java keyword transient

Identification

Entity Identification

- All entities must have a persistent id (database primary key)
- JPA supports:
  - simple types (single field)
  - composite primary key (multiple columns in the database)
- Must be defined on the root entity or mapped superclass of the hierarchy

Persistent ID

- Define a primary key field/property with the @Id annotation
- Use @EmbeddedId to map one entity field/property to a composite primary key
- Use @IdClass to map multiple entity fields/properties to a composite primary key
- Composite id classes must be `Serializable`

Single Primary Key

```
@Entity
public class BankAccount {

    @Id
    private Long id;

    // ...
}
```

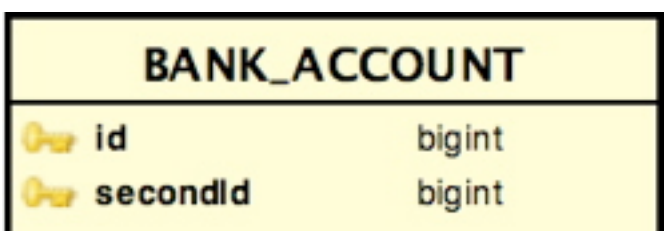
Composite Primary Key

1) Add @IdClass to BankAccount

```
@Entity
@Table(name="BANK_ACCOUNT")
@IdClass(BankAccountKey.class)
public class BankAccount {
  @Id
  private Long id;
  @Id
  private Long secondId;
}
```

2) Create Key Class (POJO)

```
public class BankAccountKey
  implements Serializable {

  private Long id;
  private Long secondId;
}
```



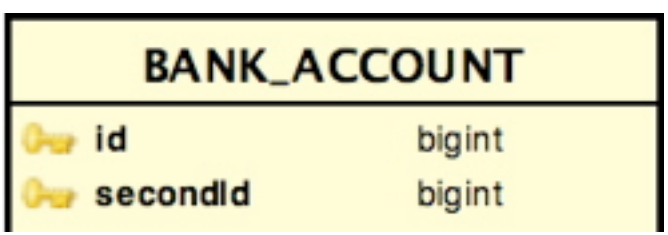
Composite Primary Key

1) Add @EmbeddedId to BankAccount

```
@Entity
@Table(name="BANK_ACCOUNT")
public class BankAccount {
  @EmbeddedId
  private BankAccountKey id;
}
```

2) Create Key Class (POJO) add Embeddable

```
@Embeddable
public class BankAccountKey
  implements Serializable {

    private Long id;
    private Long secondId;
}
```



Primary Key Generation

- Use @GeneratedValue to define primary key generation
- Generation strategies available:

```
public enum GenerationType {
  TABLE, SEQUENCE, IDENTITY, AUTO
};
```

- For database portability, use AUTO if possible

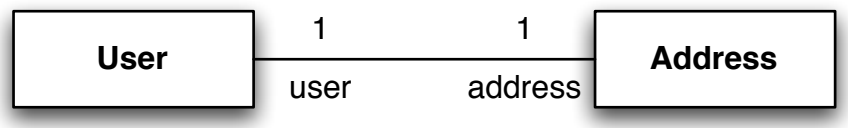
Relationships

JPA Relationship Support

- Supports composition and aggregation relationships
- Relationships: one-to-one, one-to-may, many-to-one, many-to-many
- A relationship may be unidirectional or bidirectional
- Bidirectional relationships are managed by the application (not the persistence provider)
  - Bidirectional relationships have an owning side and inverse side

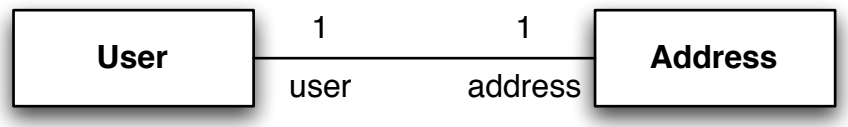
Bidirectional Realtionships

Bidirectional Relationship



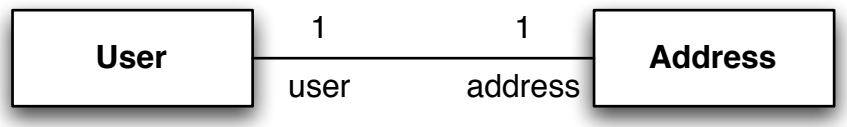
- Owning side: entity whose database table contains a foreign key column for the target entities primary key
- For this example, Address will be the owning side

Entity/Table




```java
@Entity
@Table(name="address")
public class Address {
    @Id
    private Integer id;

    @OneToOne
    private User user;

    private String streetAddress;
}


@Entity
public class User {
    @Id
    private Integer id;

    @OneToOne(cascade=CascadeType.ALL, mappedBy="user")
    private Address address;
}
```

Relationship Management

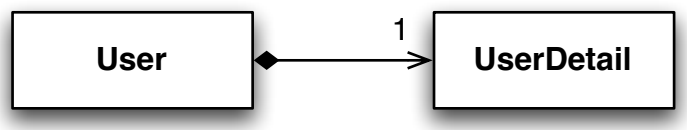



```
@Entity
public class User {

  @Id
  private Integer id;

  @OneToOne(cascade=CascadeType.ALL, mappedBy="user")
  private Address address;

  public void setAddress(Address address) {
    this.address = address;
    address.setUser (this);
  }
}
```

Composition

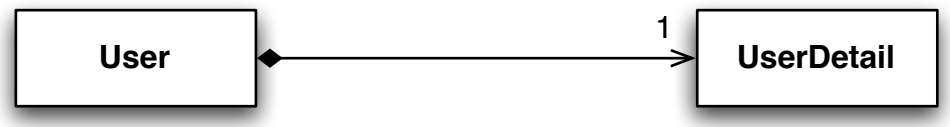
Composition

- Composition defines a strong relationship between two objects, one object "owns" the other, including its lifetime



- Use @Embedded and @Embeddable for composition relationships
- Defines a logical separation of entities
- Both entities are stored in the same db table

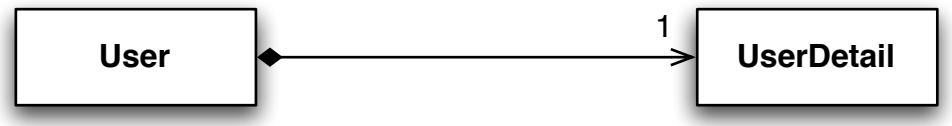
Entity

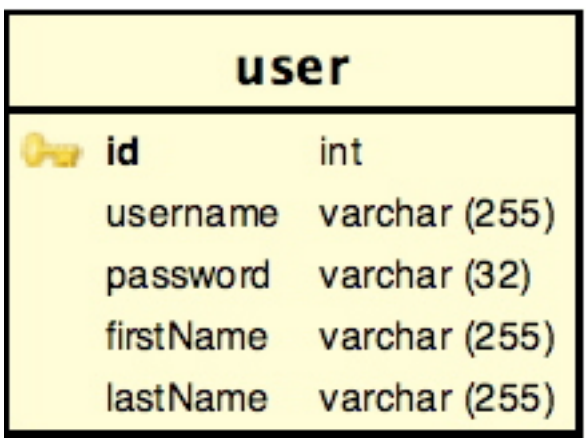



```
@Entity
@Table(name="user")
public class User {

  @Embedded
  private UserDetail userDetail;

  private String username;
  private String password;
}
```

```
@Embeddable
public class UserDetail {

  private String firstName;
  private String lastName;
}
```

Database View





Aggregation

Aggregation Relationship

- An aggregation relationship represents a part-of or has-a relationship
  - each entity may have its own independent lifetime
  - and may associated with other entities
- Defined using @ManyToOne, @OneToMany, @ManyToMany, and @OneToOne annotations

Cascade & FetchType

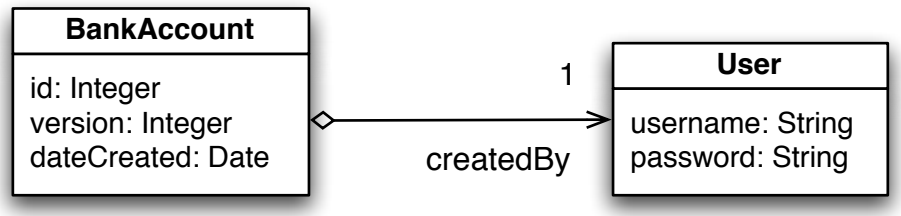
- JPA provides support for cascading operations across relationships, via the CascadeType enum

```
public enum CascadeType {
  ALL, PERSIST, MERGE, REMOVE, REFRESH
};
```

- JPA also provides support for lazy or eager fetching of relationships

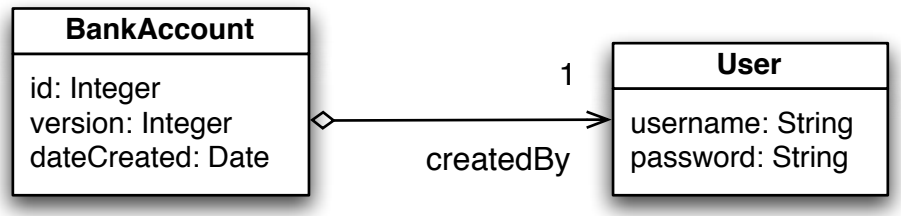
```
public enum FetchType { LAZY, EAGER };
```

Many-to-One



- Use @ManyToOne for the many-side of a many-to-one relationship.
- @ManyToOne represents an object reference
- The many-side of a many-to-one relationship is always the owning side
- This is this owning side
- Use @JoinColumn for foreign key column definition

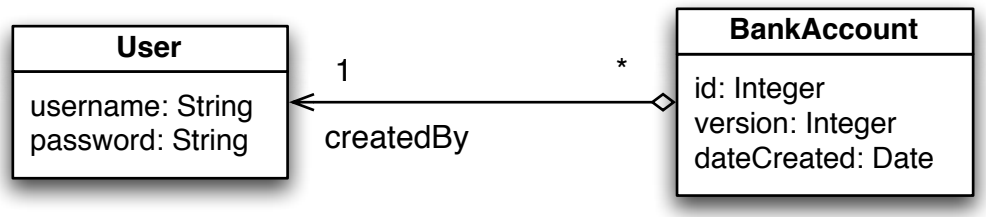
Entity Definition Many-to-One




```
@Entity
public class BankAccount implements Serializable {
  // ...

  @ManyToOne(optional=false, fetch=FetchType.LAZY)
  @JoinColumn(name="created_by_user_id")
  private User createdBy;
}
```

Use the `optional` element to indicate the column is `nullable` or not

One-to-Many



- Use the @OneToMany annotation for the "one" side of a one-to-many relationship
- One-to-many represents a collection
- This is not the owning side of the relationship
- Use @JoinColumn for foreign key column definition
- Use @OrderBy to specify the collections order

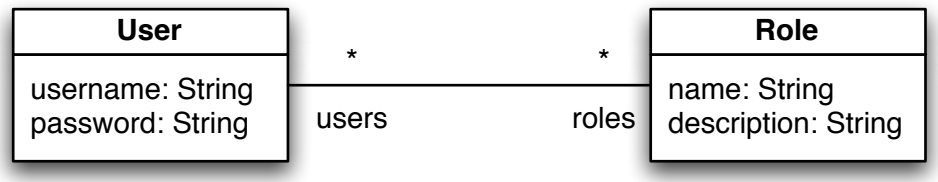
Entity Definition One-to-Many


```
@Entity
@Table(name="user")
public class User {

   @OneToMany(mappedBy="createdBy")
   @OrderBy("dateCreated DESC")
  private List<BankAccount> accounts;

}
```

- The BankAccount table has a foreign key column, `created_by_user_id`

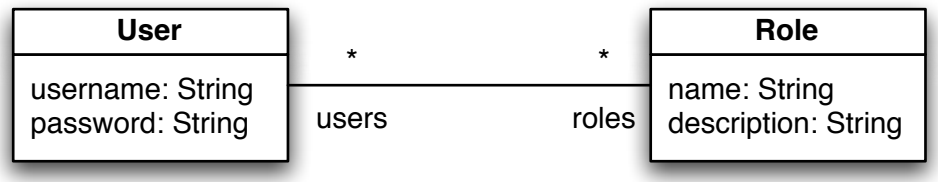
The `OneToMany` annotation does not have an `optional` element, it is not the owning side

Many-to-Many



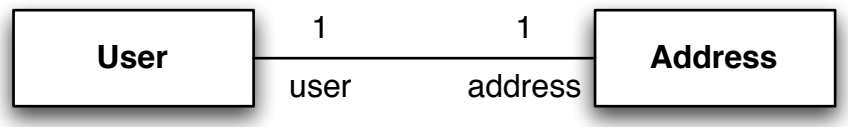
- Use the @ManyToMany annotation for both sides of the relationship
- Represents collections on both sides
- Either side may be the owning side of the relationship
- Use @JoinTable on owning side to define the join table
- Use @OrderBy to specify the collections order
- Shares the same elements as @OneToMany

Entity DefinitionMany-to-Many




```
@Entity
public class Role {

  @ManyToMany(cascade=CascadeType.ALL, fetch=FetchType.LAZY)
  @JoinTable(name = "user_role",
     joinColumns = @JoinColumn(
        name = "user_id", referencedColumnName = "id"),
     inverseJoinColumns = @JoinColumn(
        name = "role_id", referencedColumnName = "id"))
    protected Set<User> users = new HashSet<User>();
}
```

One-to-One



- Use the @OneToOne annotation for both sides of the relationship
- Represents a object reference
- Either side may be the owning side of the relationship
- Shares the same elements as @ManyToOne

Inheritance

JPA Inheritance Support

- The JPA provides two annotations for inheritance support: @MappedSuperclass and @Inheritance
- You can have mixed inheritance hierarchies, entities and non-entities, either concrete or abstract
- Use @MappedSuperclass to define common fields/properties used in your model
- Use @Inheritance for traditional inheritance hierarchies

MappedSuperclass

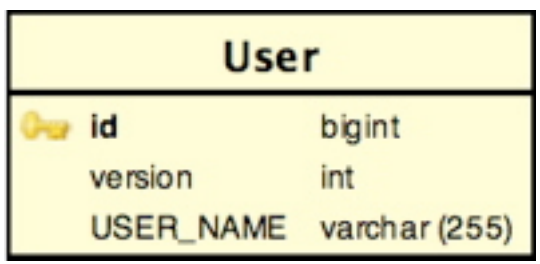
MappedSuperclass

- A MappedSuperclass does not map to any database table
- The table an entities is mapped to contains the MappedSuperclass attributes (non-normalized)
- A MappedSuperclass cannot be the target of a relationship
- You cannot query a MappedSuperclass

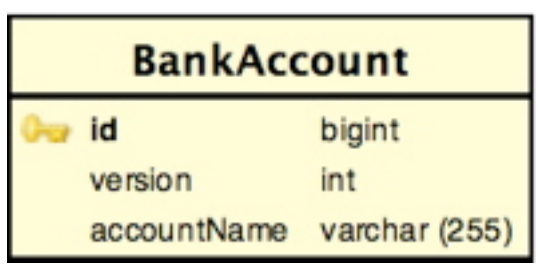
MappedSuperclass Example

```
@MappedSuperclass public abstract class ModelBase  {
  @Id private Integer id;
  @Version private Integer version;
}
```

```
@Entity public class User extends ModelBase {
  @Column(name="USER_NAME")
  private String username;
}
```




```
@Entity
public class BankAccount extends ModelBase {
  private String accountName;
}
```



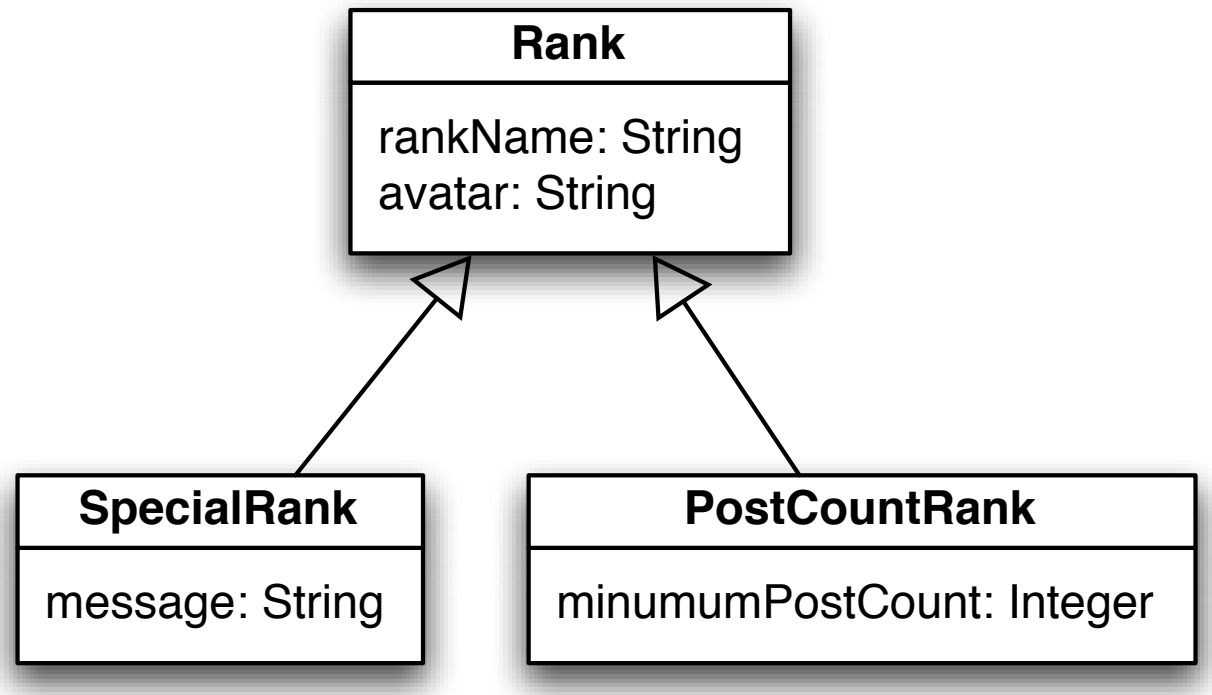
@Inheritance

@Inheritance

- An entity can subclass:
  - concrete or abstract non-entity class
  - concrete or abstract entity class
- The JPA supports three inheritance strategies:
  - SINGLE_TABLE – one table for the hierarchy
  - JOINED – each entity (abstract or concrete) in the hierarchy gets its own table
  - TABLE_PER_CLASS – (optional) one table per concrete class

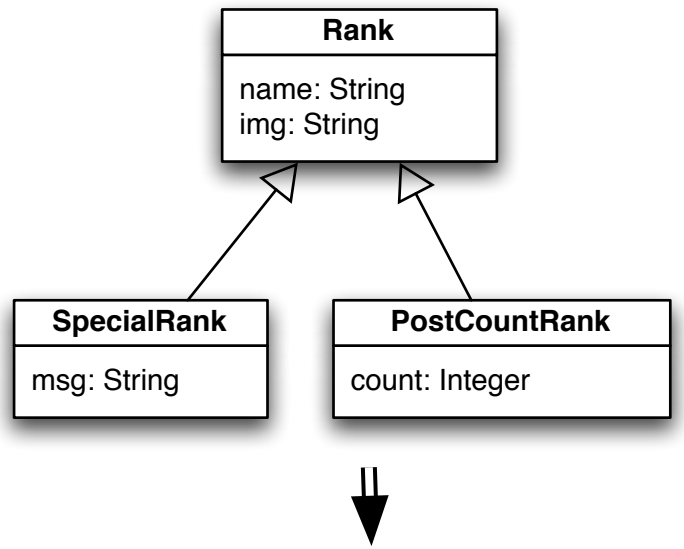
```
public enum InheritanceType  { SINGLE_TABLE, JOINED,TABLE_PER_CLASS };
```

Example Class Diagram



SINGLE_TABLE

```
@Entity
@Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorColumn(name="type",
  discriminatorType=STRING)
public abstract class Rank {
  protected String name;
  protected String img;
}

@Entity
@DiscriminatorValue("SPECIAL_RANK")
public class SpecialRank
    extends Rank {
  protected String msg;
}

@Entity
@DiscriminatorValue("POST_COUNT")
public class PostCountRank extends Rank {
  protected Integer count;
}
```

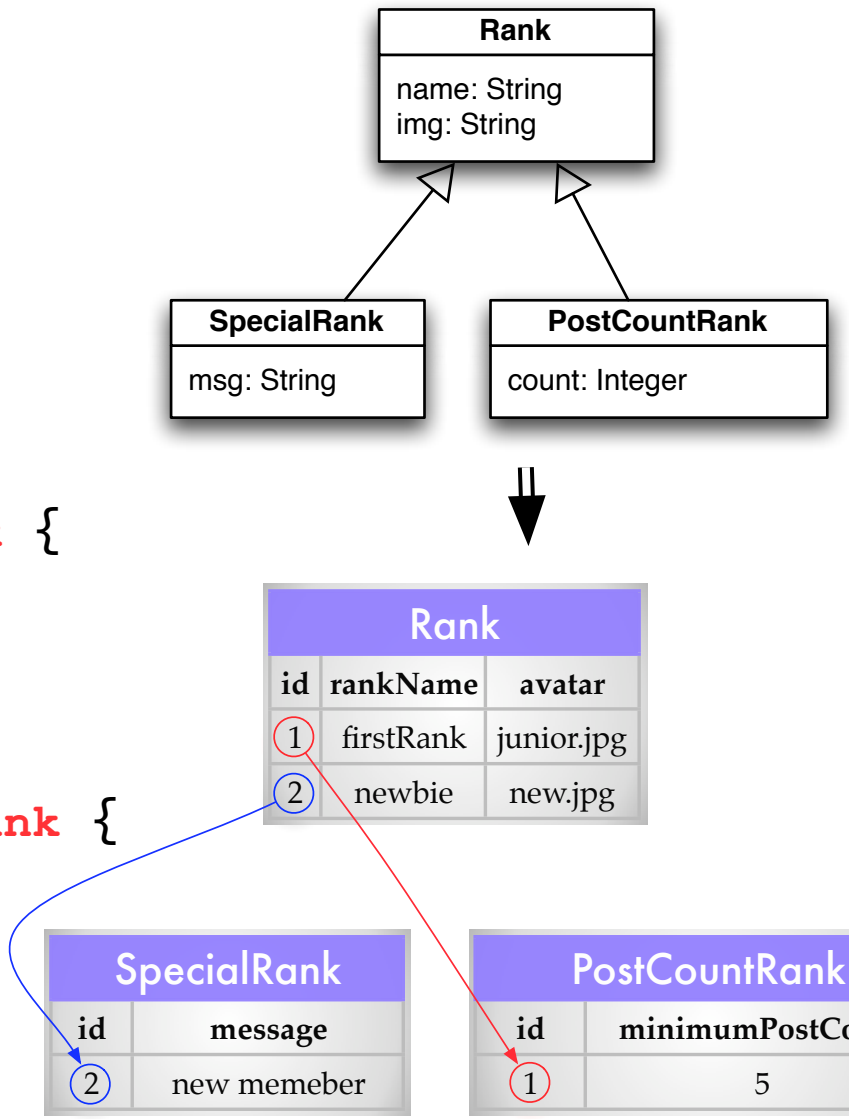


Rank

id	type	rankName	avatar	message	minimum PostCount
1	POST_COUNT	firstRank	junior.jpg		5
2	SPECIAL_RANK	newbie	new.jpg	new member	

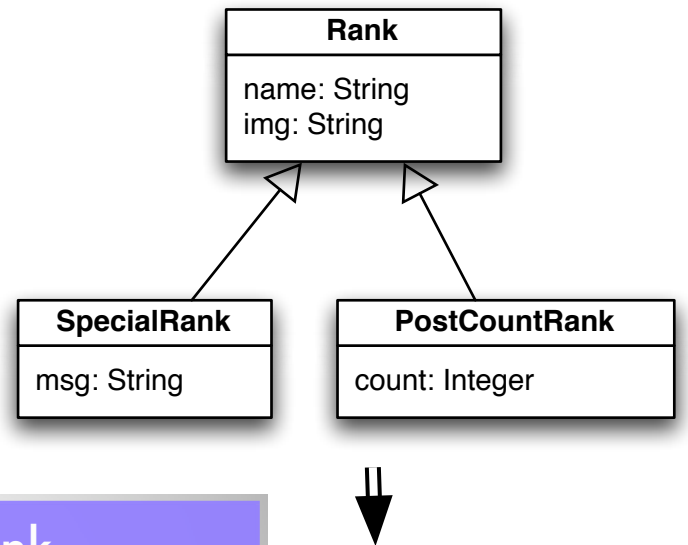
JOINED

```
@Entity
@Inheritance(strategy=JOINED)
public abstract class Rank {
  protected String rankName;
  protected String avatar;
}
@Entity
public class SpecialRank extends Rank {
  protected String message;
}
@Entity
public class PostCountRank extends Rank {
  protected Integer minimumPostCount;
}
```



TABLE_PER_CLASS

```
@Entity
@Inheritance(strategy=TABLE_PER_CLASS)
public abstract class Rank {
  @Id @GeneratedValue(
    strategy=GenerationType.TABLE)
  private Long id;

}

@Entity
public class SpecialRank
    extends Rank {
  protected String msg;
}

@Entity
public class PostCountRank extends Rank {
  protected Integer count;
}
```



SpecialRank

id	rankName	avatar	message
2	newbie	new.jpg	new member

PostCountRank

id	rankName	avatar	minimum PostCount
1	firstRank	junior.jpg	5

You cannot use the identity generation strategy if using Hibernate

Agenda

- JPA Overview
- Entities
- **Packaging**
- EntityManager
- Java EE in a Java SE World
- JPQL

Packaging

The only XML you really need to know

Packaging your Entities

- JPA requires one XML file:

  `META-INF/persistence.xml`

  - Defines the name of the persistence unit
  - Defines transaction strategy for EntityManager
  - Identifies entities contained within a persistence unit
  - Define persistence provider configuration
- You can also create O/R mapping files
  - similar to Hibernate's .hbm.xml filels
  - can complement, or override your annotations
  - identified in `persistence.xml`

persistence.xml

```
<persistence ...>
  <persistence-unit name="djbug" transaction-type="RESOURCE_LOCAL">

    <provider>
      org.hibernate.ejb.HibernatePersistence
    </provider>

    <mapping-file>META-INF/orm-mapping.xml</mapping-file>

    <class>org.ujug.jpa.model.BankAccount</class>

    <properties>
      <property name="hibernate.show_sql" value="false" />
    </properties>
  </persistence-unit>
</persistence>
```

- `trasnaction-type` can be: `JTA` or `RESOURCE_LOCAL`

JPA Terminology

- Persistence Unit – a named logical grouping of entities, defined using persistence-unit element in `persistence.xml`

```
<persistence-unit name="djbug" ... >
```

- Persistence Context – a collection of entities at runtime, managed by an EntityManager

Agenda

- JPA Overview
- Entities
- Packaging
- **EntityManager**
- Java EE in a Java SE World
- JPQL

EntityManager

Where simplicity meets complexity

Entity Manager

- Difference between Java SE and Java EE apparent when using EntityManager
- Similar to Hibernate's Session interface
- Your gateway to the Persistence Context
- Provides CRUD services, `persist`, `remove`, `find`, `merge`, etc.
- Interface to the Transaction API
- Factory for Query Instances

EM & Persistence Context

- Two types of EntityManagers:
  - Container-managed EntityManager – intended for Java EE, use @PersistenceContext annotation
  - Application-managed EntityManager – intended for Java SE/EE, use EntityManagerFactory to get reference, application responsible for EM lifecycle
- Two types of persistence context:
  - transaction-scoped persistence context
  - extended persistence context (new to java ORM)

Container Managed EM

- Container-managed, transaction-scoped EM:
  - define transaction-scoped in `persistence.xml`

```
<persistence-unit ... transaction-type="JTA">
```

  - Create a container-managed EM:

```
// Default type
@PersistenceContext(type=TRANSACTION)
protected EntityManager entityManager;


@PersistenceContext(type=EXTENDED)
protected EntityManager entityManager;
```

Application Managed EM

- Can be used in Java EE and Java SE environments
- Application required to close the EM when finished
- Application decides when to join existing transactions
- Obtain an EM through EntityManagerFactory
- Uses an ExtendedPersistenceContext

An application-managed PC is extended because the application controls its lifetime

Application Managed EM

- In a container environment you can use the `@PersistenceUnit` annotation:

```
@PersistenceUnit
EntityManagerFactory emf;
```

- In any environment you can use the `Persistence` class:

```
EntityManagerFactory emf =
  Persistence.createEntityManagerFactory("djbug");
```

Using an Application Managed EM

```
@PersistenceUnit
EntityManagerFactory emf;

  public User createUser(String username, String password, String email) {
    User u = null;


    EntityManager em = emf.createEntityManager();
    try {
      u = new User();
      u.setUsername(username);
      u.setPassword(password);
      u.getUserDetail().setEmailAddress(email);

      em.getTransaction().begin();
      em.persist(u);
      em.getTransaction().commit();

    } finally {
      em.close();

    }
    return u;
  }
```

`@PersistenceUnit` is applicable if you have a Java EE 5 container

Using an Application Managed EM

```
public User createUser(String username, String password, String email) {
  User u = null;

  EntityManagerFactory emf = Persistence.createEntityManagerFactory("ujug");
  EntityManager em = emf.createEntityManager();
  try {
    u = new User();
    u.setUsername(username);
    u.setPassword(password);
    u.getUserDetail().setEmailAddress(email);

    em.getTransaction().begin();
    em.persist(u);
    em.getTransaction().commit();

  } finally {
    em.close();
    emf.close();
  }
  return u;
}
```

Callbacks and Listeners

- Callback methods are defined on an entity

```
void methodName()
```

- Listener methods are defined on non-entity classes

```
void methodName(Object entity)
```

- The callback/listener methods can be registered for the following:
  - @PrePersist/@PostPersist
  - @PreUpdate/@PostUpdate – persistence provider dependent
  - @PreRemove/@PostRemove
  - @PostLoad

Example Callback

- To register for a callback, just create a method and add a lifecycle event annotation:

```
@MappedSuperclass
public abstract class ModelBase  {

  @PrePersist
  public void setDates() {

    Date now = new Date();
    if (dateCreated == null) {
      dateCreated = now;
    }
    dateUpdated = now;
  }
}
```

Example Entity Listener

- Register the listener class:

```
@MappedSuperclass
@EntityListeners({ModelListener.class})
public abstract class ModelBase  {
  // fields and methods removed for readability
}
```

- Implement the listener:

```
public class ModelListener {
  @PrePersist
  public void setDates(ModelBase modelBase) {

    Date now = new Date();
    if (modelBase.getDateCreated() == null)
       modelBase.setDateCreated(now);
    modelBase.setDateUpdated(now);
  }
}
```

Agenda

- JPA Overview
- Entities
- Packaging
- EntityManager
- **Java EE in a Java SE World**
- JPQL

Java EE in a Java SE

Spring to the rescue

Use Spring to bridge the divide

- Use Spring 2.0 JPA Support
- Spring provides JPA integration, implementing JPA Container responsibilities
- Works with your persistence provider (e.g. Hibernate or TopLink)
- Enables Java EE only annotations in a Java SE applications (e.g. @PersistenceContext, etc.)
- See Spring 2.0 documentation for more

Agenda

- JPA Overview
- Entity
- Packaging
- EntityManager
- Java EE in a Java SE World
- **JPQL**

JPQL

SQL done write?

JPQL Overview

- JPQL executes over the *abstract persistence schema* (the entities you've created) defined by your persistence unit.
- SQL executes over the physical database schema
- JPQL supports a SQL like syntax

```
select from [where] [group by] [having] [order by]
```

- JPA supports dynamic and static (named) queries

JQPL Overview

- Supports projection into non-entity classes
- Aggregate functions, e.g. max, min, avg, etc.
- Bulk updated and delete support
- Subquery support
- Supports both inner and outer joins

Query API

- Use EntityManager to obtain a Query instance:

```
Query q = em.createQuery("query string");
```

- Query allows you to:
  - control pagination
  - define the maximum results returned
  - bind parameters by name or position

Creating a Query

- EntityManager query related methods:

```
public Query createQuery(String qlString);
public Query createNamedQuery(String name);
public Query createNativeQuery(String sqlString);
public Query createNativeQuery(String sqlString, Class resultClass);
public Query createNativeQuery(String sqlString, String resultSetMapping);
```

```
@PersistenceContext
protected EntityManager entityManager;

public List findAll() {
  Query q = entityManager.createQuery("select user from User u");

  return q.getResultList();
}
```

Static and Dynamic Queries

- Static queries are defined using @NamedQuery, and retrieved via `EntityManager.createNamedQuery()`
- Dynamic queries are defined using `EntityManager createQuery()`
- Use static queries over dynamic, they can be precompiled

Example Dynamic Query

```
Query q = em.createQuery("SELECT u FROM User u " +
                         “WHERE u.username = ?1");

q.setParameter(1, "mac");
q.setFirstResult(10); // numbered from 0
q.setMaxResults(10); // only fetch 10

List results = q.getResultList();
```

Example Static Query

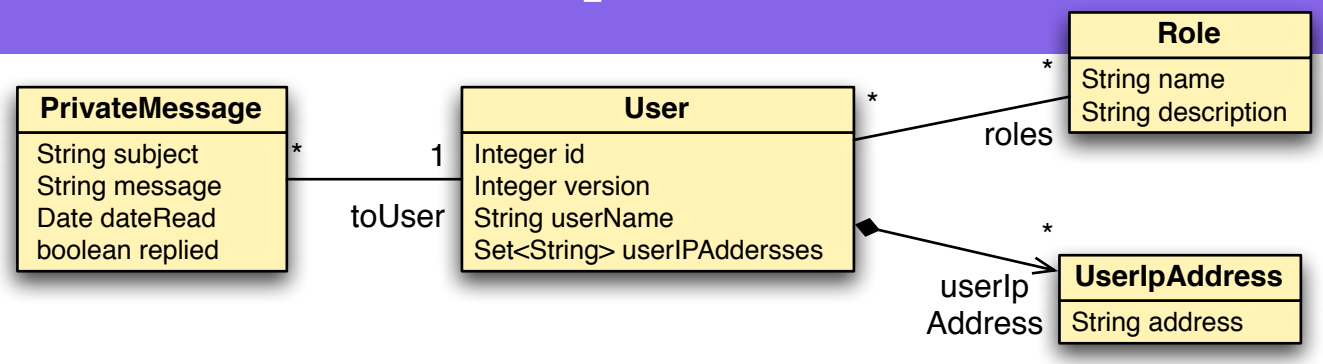
- Define a named query:

```
@Entity
@NamedQuery(name="user.updatedAtIP",
      query="select u from User u where u.updatedAtIP like :ip")
public class User extends ModelBase {
  // removed for readibility...
}
```

- Using the above named query:

```
Query q = em.createNamedQuery("user.updatedAtIP");

q.setParameter("ip", "127.0.0.1");
q.setFirstResult(9); // numbered from 0
q.setMaxResults(10); // only fetch 10

List results = q.getResultList();
```

Path Expressions



- JPQL allows you to navigate entity relationships in the select clause:

```
select p.toUser.userIPAddresses from PrivateMessage p
```

- You can use the **navigation operator** (.) to access single-valued types (not collections)
- Use the join keyword to access fields in a collection type

```
select r.name from User u join u.roles r
```

Summary

Summary

- JPA is the standard for Java Persistence
- POJO based domain model
- Rich annotation support for O/R mapping, fits in well with XML mapping
- EntityManager has a simple interface, easy to learn and use
- Relationship management is kept simple, put the power in the developers hands
- Provides support for dynamic and static queries

Q & A

JPA 101 now available at SourceBeat.com

Blog: www.jroller.com/page/cmaki