Hibernate
Entity Life-Cycle

Lesson Outline

- Entity Life-Cycle (Object States)
- Session Operations

Lesson Outline

- Entity Life-Cycle (Object States)
- Session Operations

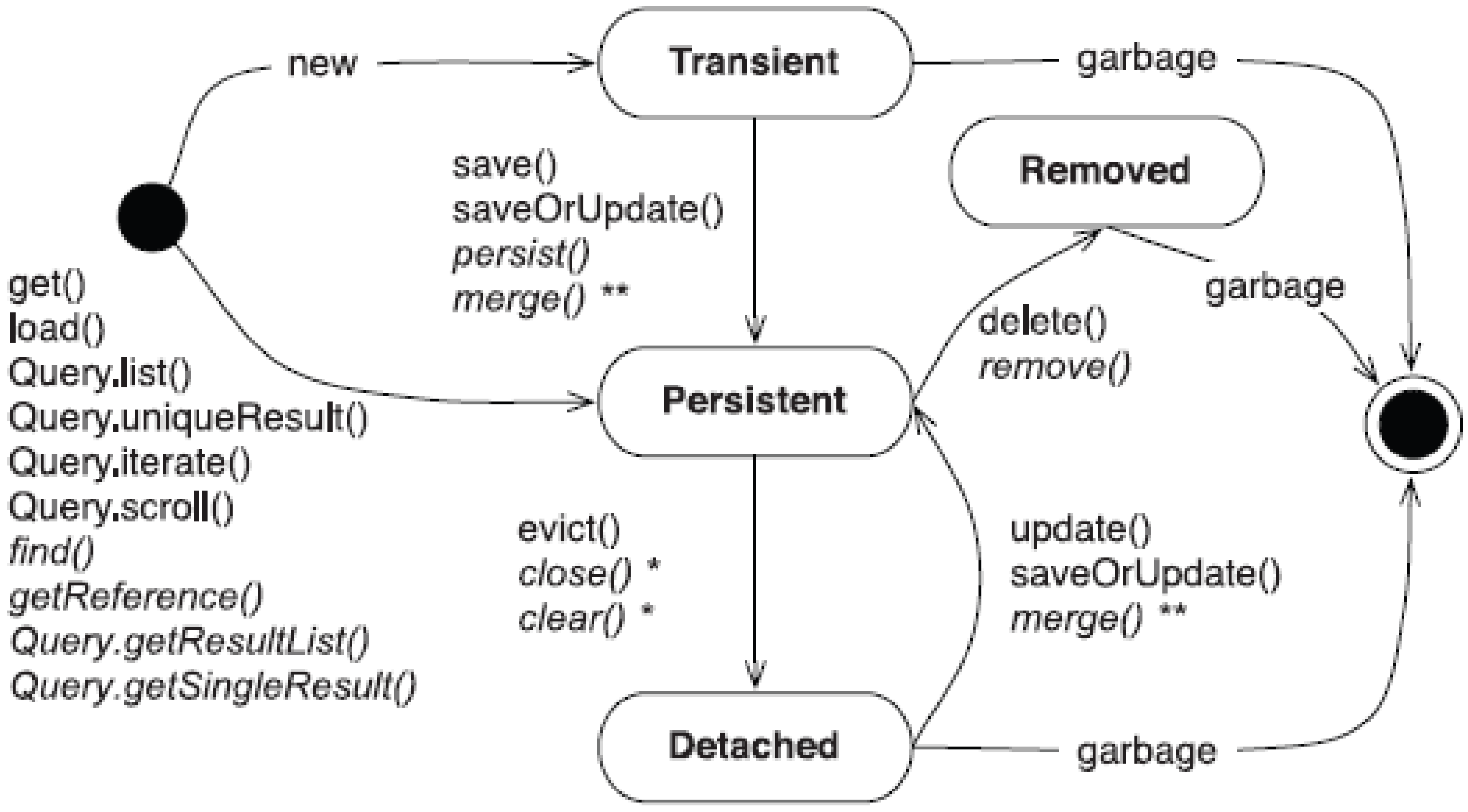
Entity Life-Cycle (Object states)

- Any application with persistent state must call Hibernate interfaces to store and load objects.

- it’s necessary for the application to concern itself with the state and lifecycle of an object with respect to persistence.

  – We refer to this as the ***Persistence lifecycle***

Entity Life-Cycle (Object states) Diagram



* Hibernate & JPA, affects all instances in the persistence context
** Merging returns a persistent instance, original doesn't change state

Transient Objects

- Objects instantiated using the new operator **aren’t** immediately persistent.
  - Their state is *transient*, which means they aren’t associated with any database table row
- Hibernate consider all transient instances to be non transactional;
  - any modification of a transient instance isn’t known to Session and doesn't propagated to DB.
- Hibernate doesn’t provide any roll-back functionality for transient objects.

Persistent Objects

- A persistent instance is an entity instance with a database identity
- That means a persistent and managed instance has a primary key value set as its database identifier
- Hibernate caches them and can detect whether they have been modified by the application.
  - And changes are reflected to the database tables.

Detached Objects

- When Session closed
  - All *persistent* instance become detached instance.
- Which means that their state is no longer guaranteed to be synchronized with database state;
  - they're no longer attached to a Session . They still contain persistent data (which may soon be stale).
- You can continue working with a detached object and modify it.

Detached Objects (Ex.)

- However, at some point you probably want to make those changes persistent
- In other words, bring the detached instance back into persistent state.
- Hibernate offers two operations, ***reattachment*** and ***merging***, to deal with this situation.

Removed Objects

- An object is in the *removed* state if it has been scheduled for deletion at the end of a unit of work.

Lesson Outline

- Entity Life-Cycle (Object States)
- Session Operations

Session Operations

- Session interface provides methods for these operations :
  - Saving objects

```
Person p = new Person();
session.save(p); or session.persist(p);
```

  - Loading objects

```
load(Class theClass, Serializable id)
```

    - This method will **throw an exception** if the unique id is not found in the database
  - Getting objects

```
get(Class theClass, Serializable id)
```

    - This method will **return null** if the unique id is not found in the database

Session Operations (Ex.)

- Refreshing objects
  - When Your Hibernate application is not the only application working with this data you can use it

    `refresh(Object object)`
- Updating objects
  - **`update(Object object)`**
  - **`saveOrUpdate(Object)`**
  - **`merge(Object object);`**
  - saveOrUpdate and merge methods create a new one if the object is not persisted
- Deleting objects
  - Removes an object from the database
  - **`delete (Object object)`**
- Querying objects

Useful Tips …

- Generating Schema from xml configuration files :
  - Configuration configuration = **new** Configuration();
  - configuration = configuration.configure (*CONFIG_FILE_LOCATION*);

    SchemaExport schemaExport = **new** SchemaExport(configuration);

    schemaExport.create (**false**, **true**);
- Printing the generated SQL :
  - <property name="hibernate.format_sql">true</property>
  - <property name="hibernate.show_sql">true</property>
- Enable the getCurrentSession() :
  - <property name = "hibernate.current_session_context_class">

      thread

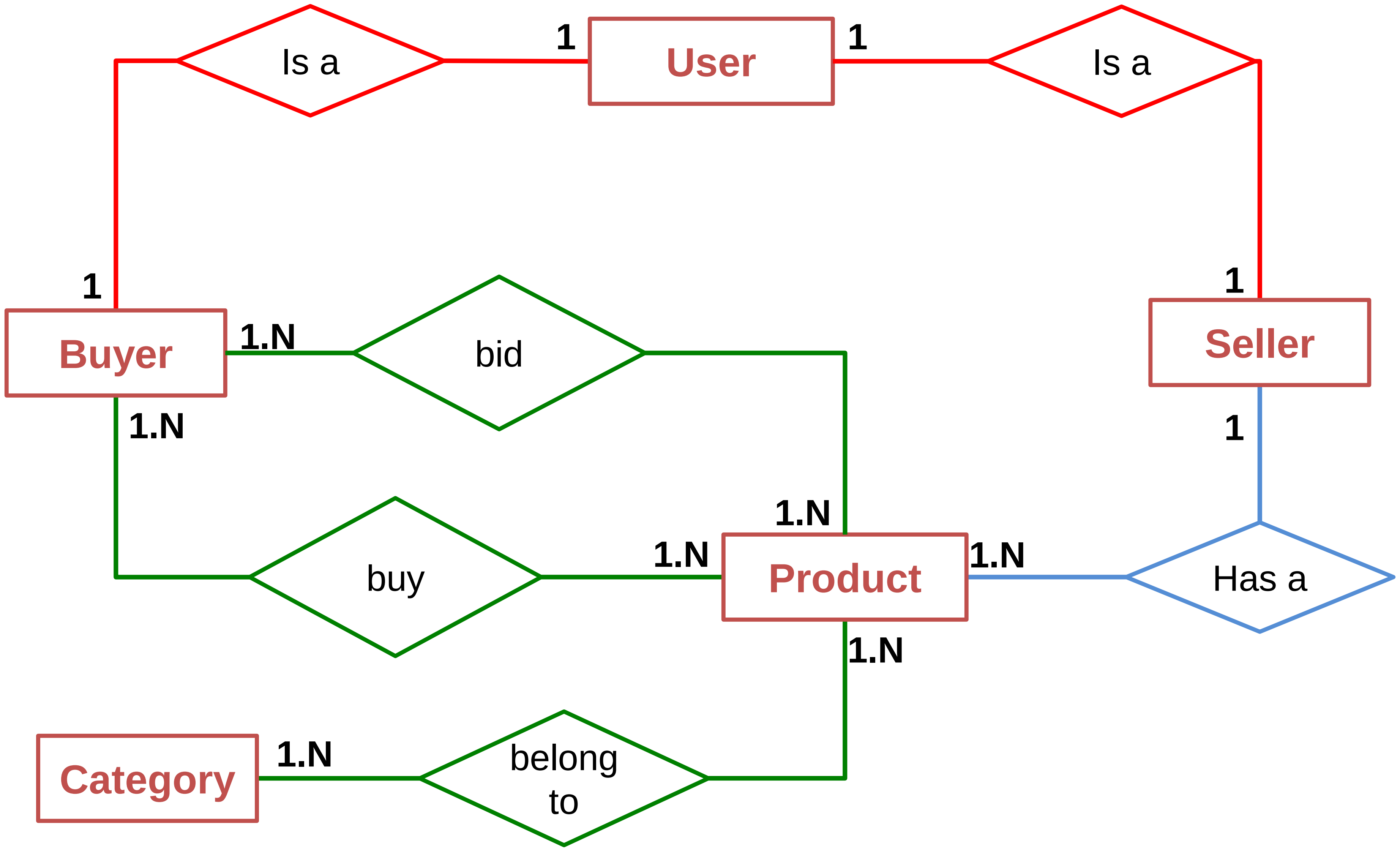
    </property>

Entities Associations

Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

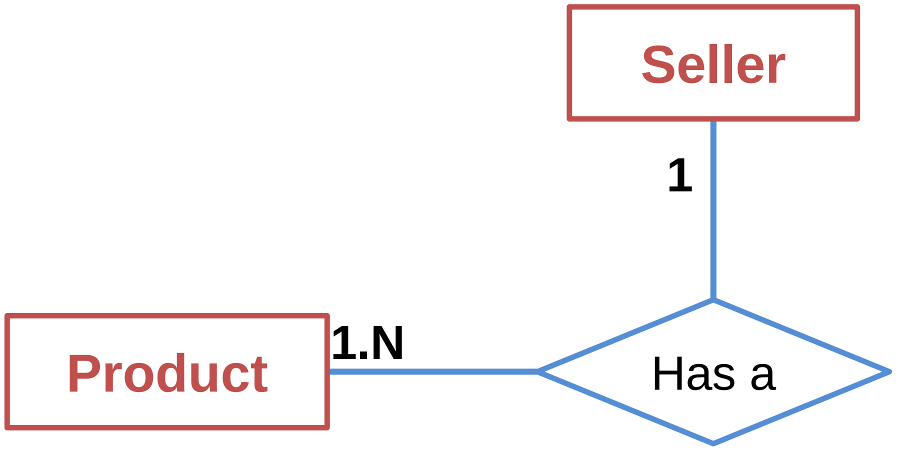
DB Diagram


1
User
1
Is a
Is a
1
1
Buyer
Seller
1.N
bid
1.N
1
1.N
buy
1.N
Product
1.N
Has a
1.N
1.N
Category
belong to

Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

DB Diagram (Ex.)



Many-to-one Association

- Association from product to seller is a **many-to-one** association.
- Since associations are directional(Uni-directional), you classify the inverse association from seller to product as: a **one-to-many** association, And it's called bidirectional
- To map association from student to group,
  - We need two properties in two classes.
    - One is a collection of references, and
    - The other a single reference.

Many-to-one Uni-directional

```
@Entity
@Table(name="product",catalog="biddingschema")
public class Product{
    ...
    @ManyToOne
    @JoinColumn(name="seller_id")
    private Seller seller;
    public Seller getSeller() {
        return seller;
    }
    public void setSeller(Seller seller) {
        this.seller = seller;
    }
    ...
}
```

Product.java

Many-to-one Uni-directional (Ex.)

```
<hibernate-mapping>
   <class name="dao.Product" table="product"
     catalog="biddingschema">
     ...
     <many-to-one name="seller" class="dao.Seller">
        <column name="seller_id"/>
     </many-to-one>
     ...
   </class>
</hibernate-mapping>
```

Product.hbm.xml

Many-to-one Uni-directional (Ex.)

- If you need the seller instance for which a particular product was selected,
  - call prodcutObject.getSeller(), utilizing the entity association you created.
  - On the other hand, if you need all products that have been offered by a specific seller, you can write a query (in whatever language Hibernate supports).
- One of the reasons you use a tool like Hibernate
  - is, of course, that you don’t want to write that query.

Many-to-one Bi-directional

- You want to be able to fetch all products offered by a particular seller without an explicit query,
  - By : sellerObject.getProducts().iterator()**

Many-to-one Bi-directional (Ex.)

```java
@Entity
@Table(name="seller",catalog="biddingschema")
public class Seller{
        ...
      @OneToMany(mappedBy="seller")
        private set<Product> products = new HashSet();

        public set getProducts() {
               return products;
        }
        public void setProducts(set products) {
               this.products = products;
        }
        ...
}
```

Seller.java

Many-to-one Bi-directional (Ex.)

```
<hibernate-mapping>

    <class name="dao.Seller" table="seller"
      catalog="biddingschema">

      ...

      <set name="products" table="product"
          inverse="true">

          <key>   <column name="seller_id" />   </key>

        <one-to-many class="dao.Product" />

      </set>

      ...

    </class>

</hibernate-mapping>
```

Seller.hbm.xml

Many-to-one Bi-directional (Ex.)

- The content of the collection is mapped with element, <one-to many>.
- The column mapping defined by the <key> element is the foreign key column seller_id of the product table,
  - The same column you already mapped on the other side of the relationship
- The inverse attribute tells Hibernate that
  - the collection is a mirror image of the <many-to-one> association on the other side.

Many-to-one Bi-directional (Ex.)

- Without the inverse attribute,
  - Hibernate tries to execute two different SQL statements, both updating the same foreign key column,
- when use inverse="true",
  - you explicitly tell Hibernate which end of the link it should not synchronize with the database.
    - In this example, you tell Hibernate that it should propagate changes made at the product end of the association to the database,
    - ignoring changes made only to the products collection.

Many-to-one Bi-directional (Ex.)

```
public class Seller{
    ...
    private set products = new HashSet();

    public set getProducts() {
      return products;
    }
    public void setProducts(set products) {
      this.products = products;
    }
    public void addProduct(Product product) {
      product.setSeller(this);
      products.add(product);
    }
    ...
}
```

Seller.java

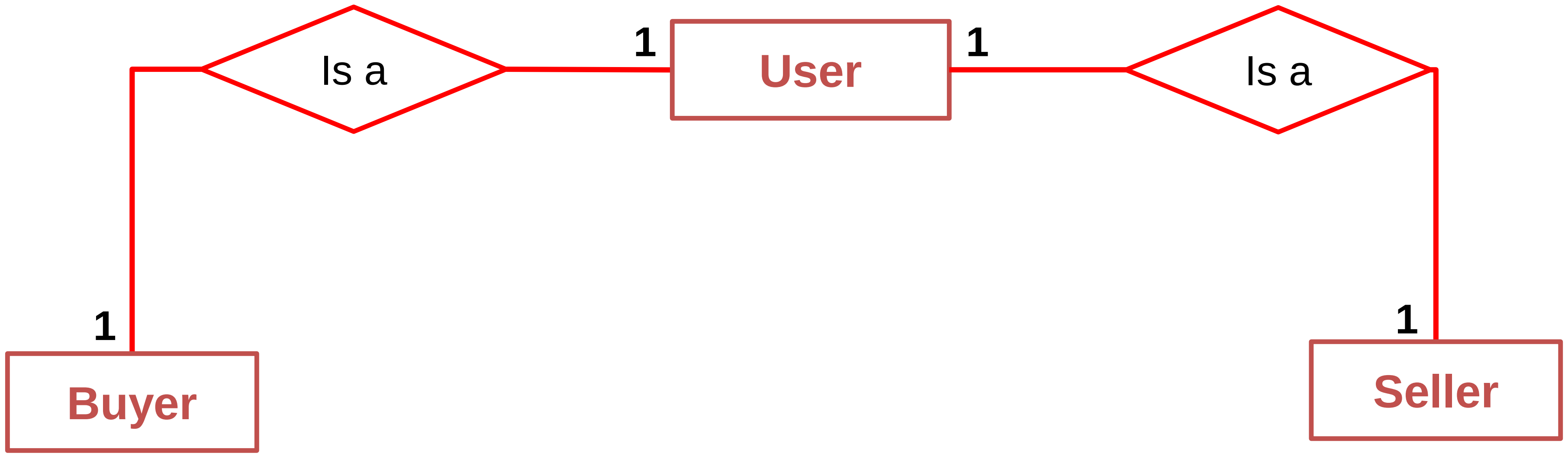
Many-to-one Bi-directional (Ex.)

- If you only call sellerObject.getProducts().add(Product),
  - no changes are made persistent!
- You get what you want only if the other side,
  - productObject.setSeller(sellerObject), is set correctly.
- It’s the primary reason why you need convenience methods such as
  - addStudent()
    - they take care of the bi-directional references in a system without container- managed relationships.

Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

DB Diagram (Ex.)



One-to-One

- Rows in two tables related by a primary key association.
  - share the same primary key values.

- The main difficulty with this approach is
  - ensuring that associated instances are assigned the same primary key value when the objects are saved.

One-to-One (Ex.)

```
@Entity
@Table(name="user",catalog="biddingschema")
public class User{
    ...
   @OneToOne(mappedBy="user")
    private Seller seller;

    public Seller getSeller() {
        return seller;
    }
    public void setSeller(Seller seller) {
        this.seller = seller;
    }
    ...
}
```

User.java

One-to-One (Ex.)

```
<hibernate-mapping>
    <class name="dao.User" table="user"
      catalog="biddingschema">
      ...
      <one-to-one name="seller" class="dao.Seller"/>
      ...
    </class>
</hibernate-mapping>
```

User.hbm.xml

One-to-One (Ex.)

- When save User and its Seller
  - Hibernate inserts a row into the User table and a row into the Seller table.
  - But How can Hibernate possibly know that the record in the Seller table needs to get the same primary key value as the User row?
  - To do that
    - You need to enable a special identifier generator.
- So, If a Seller instance is saved, it needs to get the primary key value of a User object
  - You can’t enable a regular identifier generator
    - Like database sequence

One-to-One (Ex.)

```
public class Seller{
    ...
   @OneToOne
   @PrimaryKeyJoinColumn
    private User user;
   @GenericGenerator(name="SellerIdGenerator", strategy="foreign",
parameters=@Parameter(name="property", value="user"))
    @Id
    @GeneratedValue(generator="SellerIdGenerator")
    @Column(name="id", unique=true, nullable=false)
    private int id;
    public int getId()            {   return id;   }
    public void setId(int id) {   this.id = id; }
    public User getUser() {
         return user;
    }
    public void setUser(User user) {
         this.user = user; }}
```

Seller.java

One-to-One (Ex.)

```
<hibernate-mapping>
   <class name="dao.Seller" table="seller"
     catalog="biddingschema">
     <id name="id" column="id" type="int" >
          <generator class="foreign">
               <param name="property">user</param>
          </generator>
     </id>
     <one-to-one name="user" class="dao.User"
          constrained="true"/>

   </class>
</hibernate-mapping>
```

Seller.hbm.xml

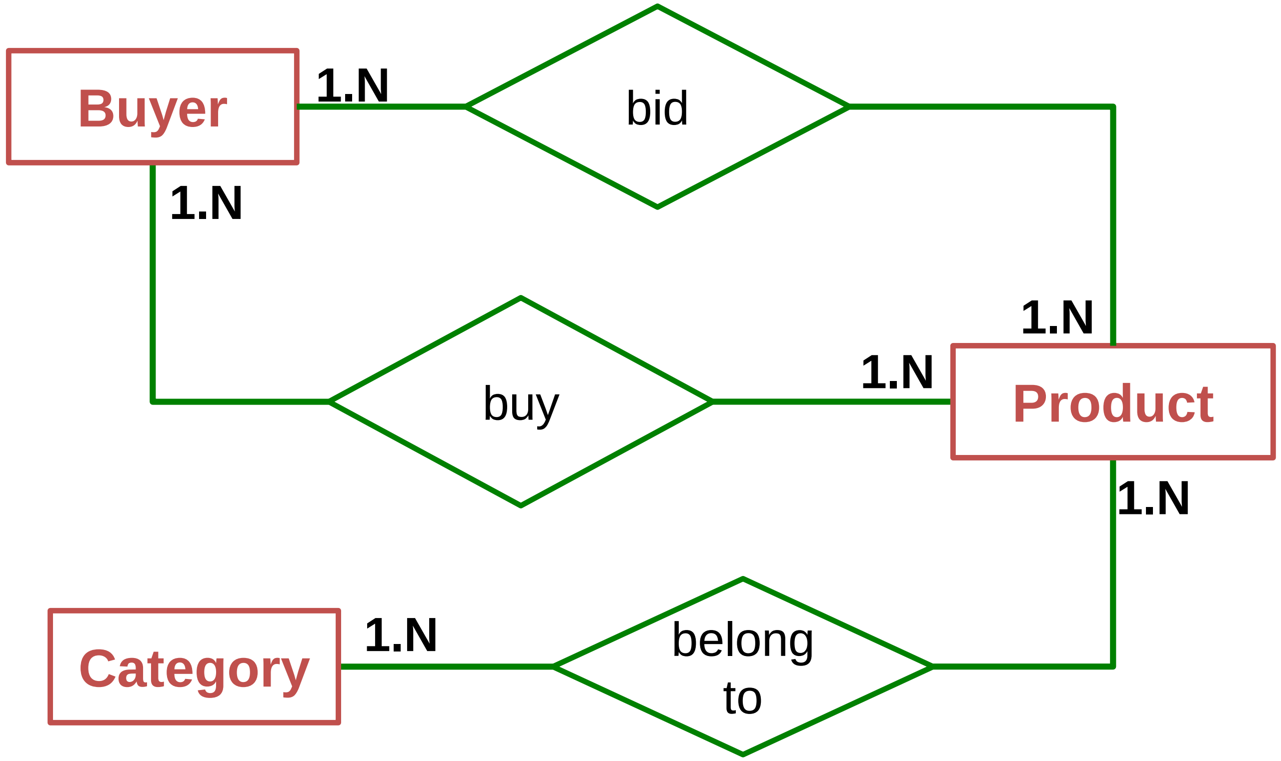
One-to-One (Ex.)

- With constrained = "true",
  - adds a foreign key constraint linking the primary key of the Seller table to the primary key of the user table.
- You can now use the special foreign identifier generator for Seller objects.
- When a Seller is saved, the primary key value is taken from the user property.
  - The user property is a reference to a User object;
  - hence, the primary key value that is inserted is the same as the primary key value of that instance.
- Note: The other way to handle relation one-to-one in db does supported in hibernate as many-to-one relation.

Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

DB Diagram (Ex.)



Many-to-Many

- A many-to-many association may always be represented as two many-to-one associations to an intervening class.
  - This model is usually more easily extensible,
  - so we tend not to use many-to-many associations in applications

- Also, remember that you don't have to map *any collection of entities,*
  - *You* can always write an explicit query instead of direct access through iteration.

Many-to-Many (Ex.)

- The join table *has two columns: the foreign* keys of the Product and Category tables.
- The primary key is a composite of both columns.
- Creating a link between a Product and a Category is easy:
  - productObject.getCategories().add(categoryObject);
  - categoryObject.getProducts().add(productObject);

Many-to-Many Uni-directional

```
public class Product{
    private set categories = new HashSet();
@ManyToMany(fetch=FetchType.LAZY)
@JoinTable(name="product_has_category",catalog="biddingschema",
joinColumns = {@JoinColumn(name="product_id", nullable=false,
updatable=false) }, inverseJoinColumns = {
@JoinColumn(name="category_id", nullable=false, updatable=false)
})
public set getCategories() {
        return categories;
    }
    public void setCategories(set categories) {
        this.categories = categories;
    }
}
```

Product.java

Many-to-Many Uni-directional (Ex.)

```
<hibernate-mapping>
   <class name="dao.Product" table="product"
     catalog="biddingschema">
     ...
  <set name="categories" table="product_has_category"
         lazy="true" fetch="select">
      <key>  <column name="product_id" />   </key>
      <many-to-many entity-name="dao.Category" >
        <column name="category_id" not-null="true"/>
      </many-to-many>
</set>
</class>
</hibernate-mapping>
```

Product.hbm.xml

Many-to-Many Bi-directional

- An association between a Product and a Category is represented in memory by
  - the Category instance in the categories collection of the Product,
  - And the Product instance in the products collection of the Category.

- The code to create the object association also changes:
  - productObject.getCategories().add(categoryObject);
  - categoryObject.getProducts().add(productObject);

Many-to-Many Bi-directional (Ex.)

```
public class Category{

    private set products = new HashSet();
 @ManyToMany(fetch=FetchType.LAZY)
 @JoinTable(name="product_has_category", catalog="biddingschema",
joinColumns = {@JoinColumn(name="category_id", nullable=false,
updatable=false) }, inverseJoinColumns = {
@JoinColumn(name="product_id", nullable=false, updatable=false)
})
    public set getProducts() {
        return products;
    }
    public void setProducts(set products) {
        this.products = products;
    }}
```

Category.java

Many-to-Many Bi-directional (Ex.)

```
<hibernate-mapping>
   <class name="dao.Category" table="category"
     catalog="biddingschema">
     ...
     <set name="products" table="product_category"
         lazy="true" fetch="select" inverse="true">
       <key>  <column name="category_id" />   </key>
       <many-to-many entity-name="dao.Product" >
         <column name="product_id" not-null="true"/>
       </many-to-many>
     </set>
</class>
</hibernate-mapping>
```

Category.hbm.xml

Many-to-Many Bi-directional

- If the Many-to-Many relation have an attribute or more on it, the relation will be represented by two tables that have a One-to-Many relation with the third one
- All three tables will be mapped to entities.
- And the third table will have a composite primary key

Many-to-Many Bi-directional (Ex.)

```
public class Buyer{

@OneToMany(fetch=FetchType.LAZY, mappedBy="buyer")
private Set<BuyerBuyProduct> buyerBuyProducts = new HashSet();
…..
 }
---------------------------------------------------------------
public class Product{
@OneToMany(fetch=FetchType.LAZY, mappedBy="product")
 private Set<BuyerBuyProduct> buyerBuyProducts = new HashSet();
}
```

Category.java
Product.java

Many-to-Many Bi-directional (Ex.)

```java
public class BuyerBuyProduct{
     @EmbeddedId
     private BuyerBuyProductId id;

     @ManyToOne(fetch=FetchType.LAZY)
     @JoinColumn(name="product_id)
     private Product product;

    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="buyer_id)
    private Buyer buyer;}
```

BuyerBuyProduct.java

Many-to-Many Bi-directional (Ex.)

```
@Embeddable
public class BuyerBuyProduct{
 @Column(name="product_id")
     public int getProductId() {
         ………
     }
@Column(name="buyer_id")
     public int getBuyerId() {…….
     }
}
```

BuyerBuyProductId.java

Many-to-Many Bi-directional (Ex.)

```
<hibernate-mapping>
    <class name="BuyerBuyProduct"
table="buyer_buy_product">
        <composite-id name="id" class="BuyerBuyProductId">
            <key-property name="buyerId" type="int">
                <column name="buyer_id" />
            </key-property>
            <key-property name="productId" type="int">
                <column name="product_id" />
            </key-property>
        </composite-id>
```

BuyerBuyProductId.java