Persistence

Filip Nguyen

Agenda

- Persistent storage
- Java Patterns
- Java Persistence API
- Support from Java middleware

Motivation

- Important for Information System
- Project in this course
- Important in the industry, any Java EE interview
- Wide adoption

Persistence I

- Persistent storage
- Java Patterns
- JPA
  - Configuration
  - Basic Entity Mappings

Resources

- JPA 2.1 specification
- Hibernate documentation http://hibernate.org/orm/documentation/
- Hibernate in Action
- JEE Development Without EJB: chapter 3, chapter 10

Persistent storage

- Data Storage Types (RDBMS, NoSQL)
- Java EE Standards JCR, JDO, JPA
- Java Libraries (Spring JDBC Template, iBatis SQL Map, JPA implementations, NoSQL API practices)

Data Storage Types

- RDBMs
  - Structured, allows constraints
  - JDBC drivers
- NoSQL
  - Unstructured
  - Proprietary drivers

Advantages of RDBMs

- Good theoretical model (Relational Algebra)
- High performance (optimizations)
- Well established and standardized (SQL)
- Supported on many platforms and programming languages

Disadvantages of RDBMs

- Not possible to horizontally scale
- Adding more RDMBs servers doesn't increase performance
- RDBMs is usually the bottleneck

Java Libraries for RDBMs

- JDBC
  - version 3, version 4
- Spring JDBC Template
- iBatis SQL Map

Java Patterns

- Persistence Layer
- Data Access Object
- Object Relational Mapping

Persistence Layer

- Encapsulates data access
- Lowest level, doesn't use any other layer underneath
- Shouldn't contain business logic
- Usually implemented using Data Access Object (DAO) design pattern

Entity

- Represents an object from reality that we model
- Usually a simple POJO class
- In Java the Entity is standardized in JPA. Word 'Entity' in JPA defines very strict Java class as shown later

Data Access Object Pattern

- Has clear interface, usually Create, Update, Read, Delete operations
- Usually 1 to 1 relationship to Entity classes

DAO - Example

- PersonDao
  - find(long id)
  - findByName(String name)
  - delete(Person p)
  - create(Person p)

DAO Transactions

- Methods of the DAO are fine grained
- DAO should participate in a transaction but does not demarcate the transactions!
- What do you know about JDBC transaction control?

DAO Exceptions

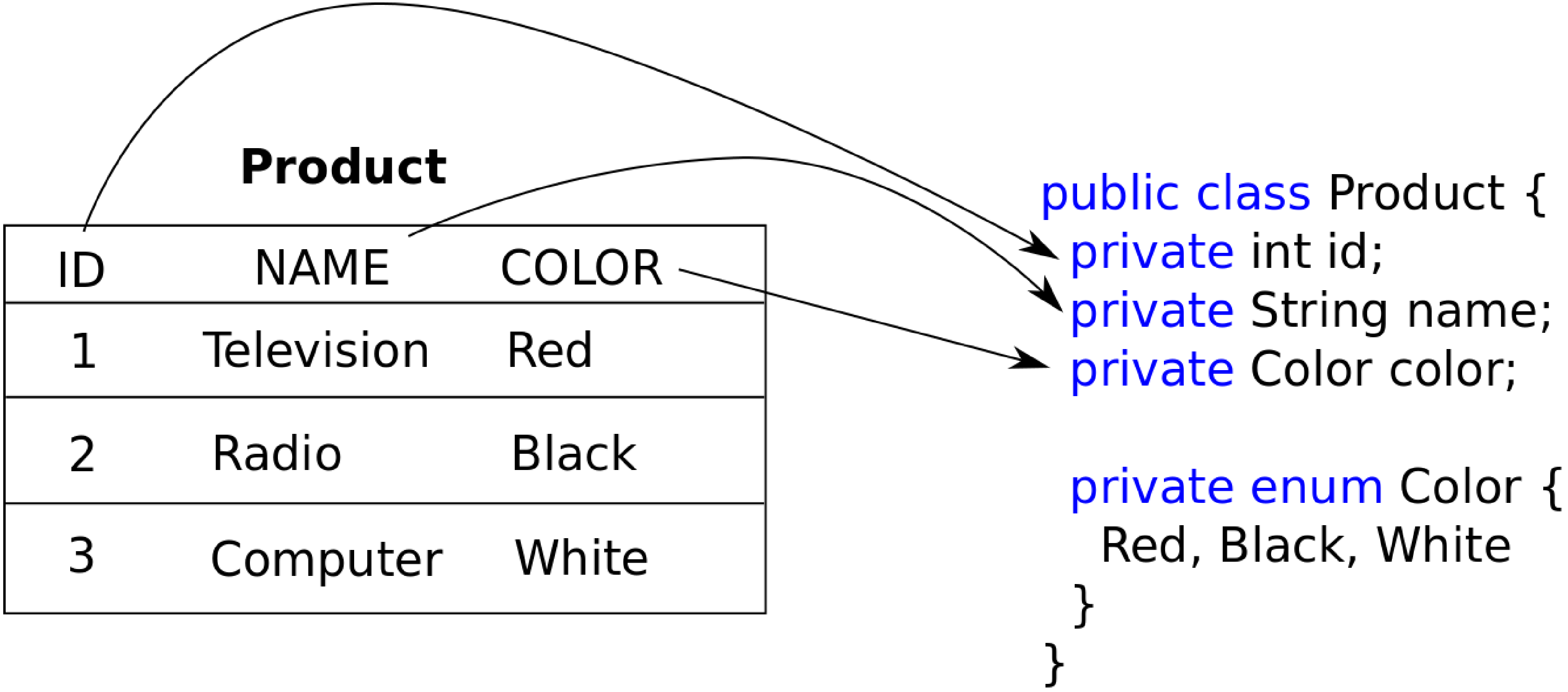
- Errors on Data Layer should be propagated
- Encapsulation applies

DAO Exceptions - Example

Naive Exception Translation:

```
public void create(Person p) {
  try {
     ….
  } catch (SqlException ex) {
    throw new DataAccessException(....);
  }
}
```

ORM Mapping



Product

ID	NAME	COLOR
1	Television	Red
2	Radio	Black
3	Computer	White

```
public class Product {
 private int id;
 private String name;
 private Color color;

 private enum Color {
  Red, Black, White
 }
}
```

Java ORM

- JPA (versions 2, 2.1)
- Hibernate
- Toplink

JPA vs Hibernate

- Java Persistence API (JPA)
  - Standard, set of Interfaces
  - PDF file with the specification
- Hibernate
  - Implementation of the Interfaces
  - Additional features on top of the JPA

JPA Entity

- Must have @Entity annotation
- Must have public/protected no-arg constructor
- Must have field acting as unique identifier annotated with @Id

JPA Entity equals/hashCode

- Use instanceof instead of getClass()
- Prefer business key instead of getId()
- Use getters on “other object” do not take advantages of visibility of other object's private field

Supported Data Types

- Java primitive types and wrappers (Integer, etc.)
- java.lang.String;
- Enums
- java.math.BigInteger; java.math.BigDecimal,
- java.util.Date; java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp,
- byte[], Byte[], char[], Character[],
- user-defined types that implement the Serializable interface;

```
@Entity
public class Product {
 @Id
 @GeneratedValue(strategy=GenerationType.IDENTITY)
 private Long id;

 @Column(nullable=false,unique=true)
 private String name;

 @Enumerated
 private Color color;
 public enum Color{ BLACK, WHITE, RED}

 public Product(Long productId) {
  this.id = productId;
 }
 public Product() {
 }

 public String getName() {
  return name;
 }

 public void setName(String name) {
  this.name = name;
 }

 public Long getId() {
  return id;
 }
}
```

Supported Data Types

- Java primitive types and wrappers (Integer, etc.)
- java.lang.String;
- Enums
- java.math.BigInteger; java.math.BigDecimal,
- java.util.Date; java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp,
- byte[], Byte[], char[], Character[],
- user-defined types that implement the Serializable interface;

Basic Field Annotations

- @Lob – large object, usually maps to database as BLOB
- @Basic(fetch=LAZY)
- @Column
- Convention over configuration
- @Transient

Basic Entity Annotations

- @Table – override the table name, to override the case of the name use \”\”
- @NamedQuery
- @SecondaryTable

Primary Key

- Entity must have primary key
- Supported Data Types: any primitive type, java.lang.String, java.util.Date, java.sql.Date, java.math.BigDecimal, java.math.BigInteger
- May be composite!
- Can be either assigned by the developer or autogenerated by the database

Primary Key - Generation

- @GeneratedValue(type=TABLE|SEQUENCE|IDENTITY)
- TABLE – a database table is created and new primary key values are taken from it
- IDENTITY – database identity column is used
- SEQUENCE – database sequences are used (some DB systems do not support this)

Primary Key - Generation

```
public class Product {
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private Long id;
```

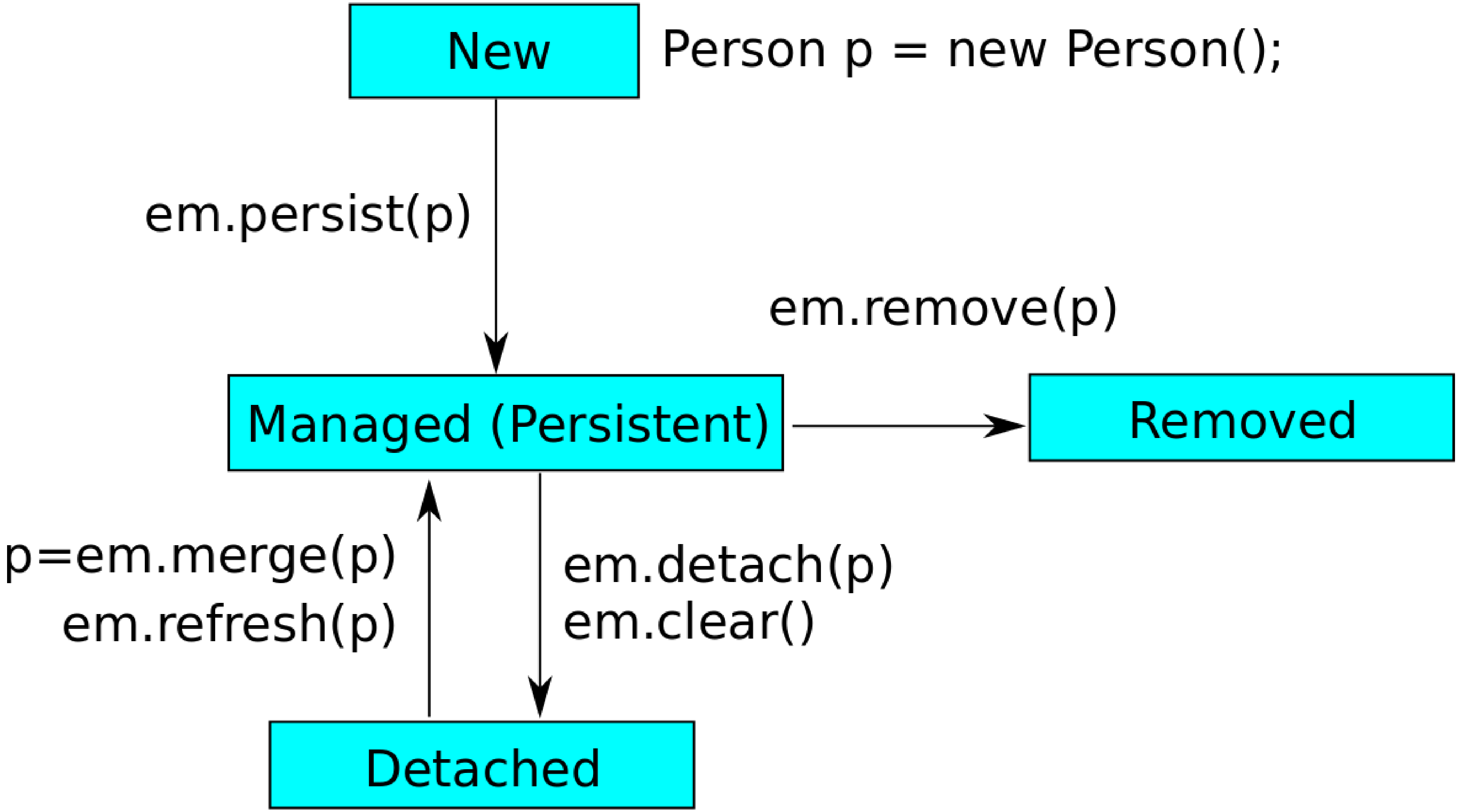
Persistence Context

- Memory area with instances of Entities
- Usually one instance of EntityManager corresponds to this memory area – entities are tied to this EntityManager

Persistence Context

A persistence context is a set of managed entity instances in which for any persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their lifecycle are managed by the entity manager.

```
EntityManager em = emf.createEntityManager();
```



New

Person p = new Person();

em.persist(p)

em.remove(p)

Managed (Persistent)

Removed

p=em.merge(p)
em.refresh(p)

em.detach(p)
em.clear()

Detached

Database Synchronization

- Persistence Context is synchronized
  - on flush() operation
  - on transaction commit() operation
- Consequence is that you may not see database insert immediately after persist()

Application Managed Entity Manager

- You must use close the entity manager through close() method
- Most typically you will use EntityTransaction and directly manage transaction boundaries
- In Container you will use @PersistenceUnit to get EMF and then you can create your EM

Example persist with Application Managed entity Manager

```
@PersistenceUnit
private EntityManagerFactory emf;
...
em = emf.createEntityManager();
em.getTransaction().begin();
em.persist(person);
person.setName(“Filip”);
em.getTransaction().commit();
em.close();
```

Example persist and detached instance

```
// somehow obtain instance of EntityManager
em.getTransaction().begin();
em.persist(person);
em.getTransaction().commit();
em.close();
person.setName(“Filip”);
```

Container Managed EntityManager

- You will not close() the EM
- Transactions are usually driven by container declaratively
- @PersistenceContext to get the manager via DI

Example persist and detached instance

```
@PersistenceContext
private EntityManager em;
….
em.persist(person);
person.setName(“Filip”);
```

JPA Configuration

- Standard persistence.xml configuration
- Persistence Unit
- Spring Java Based Configuration

PersistenceUnit

- List of classes to be used
- Database configuration
- Table creation strategy
- Configured in persistence.xml

persistence.xml

- Mandatory file for JPA
- Contains one or more Persistence Units

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/n
persistence/persistence_2_0.xsd"
             version="2.0">
  <persistence-unit name="manager1" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <mapping-file>ormap.xml</mapping-file>
    <jar-file>MyApp.jar</jar-file>
    <class>org.acme.Employee</class>
    <class>org.acme.Person</class>
    <class>org.acme.Address</class>
    <shared-cache-mode>ENABLE_SELECTOVE</shared-cache-mode>
    <validation-mode>CALLBACK</validation-mode>
    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>
```

Spring configuration

- In addition to persistence.xml we need to configure Spring to act as a container for Persistence services
- @Configuration
- Beans to configure:
  - JpaTransactionManager
  - LocalContainerEntityManagerFactoryBean
  - LoadTimeWeaver
  - DataSource
- Spring takes over much configuration that usually is in persistence.xml In the background the Spring Framework does *programatic* persistence context configuration, defined by JPA specification. Note this is not available in EE enviro

Spring Application Startup

- In Java SE application use AnnotationConfigApplicationContext
- In Web Application you can extend WebMvcConfigurationSupport which will be autodetected by Servlet 3.0 container

Basic operations

EntityManager em = emf.createEntityManager();

em.find – find an entity by ID

em.persist – persist NEW entity

em.merge – attach an entity to persistence context

em.refresh – attach an entity to persistence context and overwite database

em.remove

em.close

em.getTransaction()

  begin()

  commit()

Persistence II

- Implementing Persistence Layer – DAO objects
- JPA
  - Relationships
  - Schema Creation Strategies
  - Temporal Types
  - Cascading
  - JPQL
  - Criteria API
- Spring Data
- N+1 problem, Advantages/Disadvantages of ORM
- Beans Validation

Implementing Persistence Layer

- For each entity create a Dao object
  - Person entity will have PersonDao
- Each PersonDao is Spring @Component, is @Transactional and recieves EntityManager through @PersistenceContext annotation
- PersonDao has simple CRUD methods

JPA Relationships

- Unidirectional
- Bi-directional – maintaining runtime consistency
- Load State - PersistenceUtil
- Fetching Strategies
- Cascading

Unidirectional ManyToOne

- Field contains entity
- @ManyToOne annotation
- In Database this is represented by a foreign key
- The side with @ManyToOne is owning side
- In the example, referenced entity (Category) must be **persisted** before so that we can persist **Product**

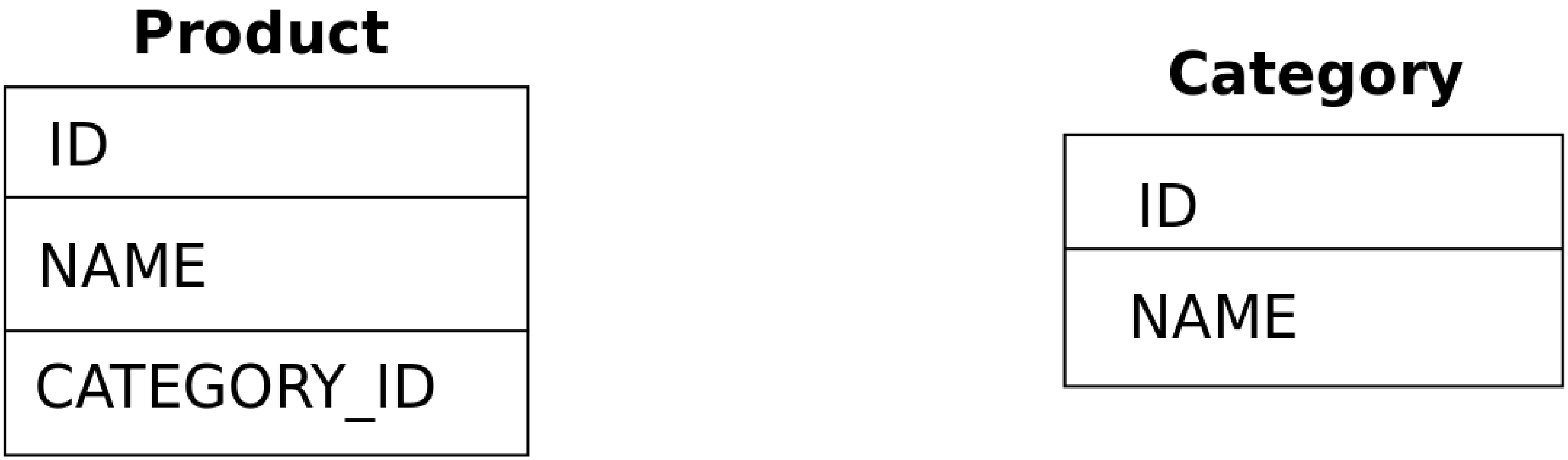
```
@Entity
public class Product {
  @Id
  private int id;
  private String name;

  @ManyToOne
  private Category category;

  public setCategory(Category cat)
  { this.category = cat; }

}
```

```
@Entity
public class Category {
  @Id
  private int id;
  private String name;
}
```



Product

ID

NAME

CATEGORY_ID

Category

ID

NAME

Owning Side, Inverse Side

- Unidirectional relationship has only *owning side*
- Bidirectional relationship has both *owning* and *inverse* side
- Owning side dictates propagation to the database

Bidirectional OneToMany

- Owning side is the one with FK on database
- Inverse side must use mappedBy
- Use defensive collections for returning the set
- Always initialize the collection

```
@Entity
public class Product {
  @Id
  private int id;
  private String name;

  @ManyToOne
  private Category category;

  public setCategory(Category cat)
  { this.category = cat; }

}
```

```
@Entity
public class Category {
  @Id
  private int id;
  private String name;

  @OneToMany(mappedBy="category")
  private Set<Product> products =
    new HashSet<Product>();

  public addProduct(Product p) {
    products.add(p);
  }

  public Set<Product> getProducts(){
    return Collections.unmodifiableSet(products);
  }

}
```

Database Schema

- In this setting we had the database schema for bi-directional will be the same as for unidirectional

Bidirectional runtime consistency

JPA spec section 2.9: *Note that it is the application that bears responsibility for maintaining the consistency of runtime relationships – for example, for insuring that the “one” and the “many” sides of a bidirectional relationship are consistent with one another when the application updates the relationship at runtime*

Database Schema

- In this setting we had the database schema for bi-directional will be the same as for unidirectional

Bidirectional runtime consistency

```
em.getTransaction.begin()
em.persist(product);
em.persist(category);
product.setCategory(category);
category.addProduct(product);
em.getTransaction.commit();
em.close();
```

LoadState

tx begin

```
List<Category> categories = em.createQuery(….);

// here a second SELECT is issued
categories.get(0).getProducts();
```

tx end

Load State

- Each attribute have default FetchType (important especially with collections)
- Collections have default FetchType.LAZY
  - They are loaded only after the collection is accessed (typically traversed by loop)
  - Setting FetchType.EAGER may result in serious performance problems since large number of objects might be loaded from the database
  - Leaving FetchType.LAZY may also result in serious performance problems since accessing the LAZY collections with loops might result in large number of queries sent to the database

```
@Entity
public class Category {
  @Id
  private int id;
  private String name;

  @OneToMany(mappedBy="category"
             fetch=FetchType.EAGER)
  private Set<Product> products =
    new HashSet<Product>();

  public addProduct(Product p) {
    products.add(p);
  }

  public Set<Product> getProducts(){
    return Collections.unmodifiableSet(products);
  }

}
```

PersistenceUtil

- You can use PersistenceUtilHelper to find out the load state of your collections on an Entity
  - LoadState.LOADED
  - LoadState.NOT_LOADED

Operation Cascading

Use of the cascade annotation element may be used to propagate the effect of an operation to associated entities. The cascade functionality is most typically used in parent-child relationships.

Operation Cascading

```
    @ManyToOne(cascade=CascadeType.PERSIST)
    private Category category;

….

em.getTransaction.begin();
em.persist(product)
product.setCategory(new Category());
em.getTransaction().commit();
```

JPA Schema Creation

- Bottom Up
- Top Down
- persistence.xml “hibernate.hbm2ddl.auto”:
  - create
  - create-drop
  - update
  - validate
  - None
- What to use in production?

Temporal Data Types

- java.util.Date vs java.sql.Date
- @Temporal()
  - TemporalType.DATE
  - TemporalType.TIME
  - TemporalType.TIMESTAMP
- How to handle Java 8 data types?

Spring Data

- http://projects.spring.io/spring-data-jpa/
- Simplifies Creation of Data Access Layer
- @EnableJpaRepositories
- public interface PriceRepository extends CrudRepository<Price, Long> {
  }
- http://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html

Entity Manager

- RESOURCE_LOCAL vs JPA transaction type
- BMT vs CMT transaction demarcation

Java Persistence Query Language

- Simple queries
- Aggregation
- Creation of new objects
- Parametrized JPQL queries
- Named Queries

Queries

```
List<Pet> pets =
em.createQuery("SELECT p FROM Pet p",Pet.class)
.getResultList();
```

- Developer is responsible to understand type of result a query generates
  - Usually the result is list of Entities or single Entity
  - More complicated results come as List<Object[]>

Creation the queries

- EntityManager.createQuery(String query,Class result)
- EntityManager.createNamedQuery(String queryName,Class result)

Using the TypedQuery Object

- getResultList() - runs the query and retrieves the list
- getSingleResult()
  - Single entity
  - Aggregation function COUNT, MAX, etc.
- setParameter(..) - used to supply parameters

Path Expression

- JPA Specification 4.4.4
  An identification variable followed by the navigation operator (.) and a state field or association field is a path expression

  ```
  SELECT s.name, COUNT(p)
  FROM Suppliers s LEFT JOIN s.products p
  GROUP BY s.name
  ```

Fetch Join

- A FETCH JOIN enables the fetching of an association or element collection as a side effect of executing the query
- The JPA provider will try to load the collection **in the same Query**! The collection will effectively be eager loaded!

```
SELECT d
FROM Department d LEFT JOIN FETCH
d.employees
WHERE d.deptno = 1
```

Empty Collection Predicate

- Test whether an associated collection is empty

```
SELECT o
FROM Order o
WHERE o.lineItems IS EMPTY
```

Named Query

- Named queries are static queries expressed in Entity metadata

```
@NamedQuery(name=”findAll”, query=”SELECT p FROM Pet p”)
@Entity
public class Pet {

List<Pet> pets = em.createNamedQuery(“findAll”,
Pet.class”).getResultList();
```

Constructor Expressions in SELECT

- So called SELECT NEW

```
SELECT NEW com.acme.exampleCustomerDetails(c.id, c.status, o.count)
FROM Customer o JOIN c.orders o
WHERE o.count > 100
```

Parametrized Queries

```
em.createQuery(“SELECT p
FROM Pet p where p.birthDate =
:date,Pet.class)
.setParameter(“date”, new Date());
```

Criteria API

- Typesafe way of creating queries
- Must pair with code generation to achieve 100% typesafeness (e.g. Hiberante JPA 2 Metamodel Generator which comes as a Maven Plugin)

Criteria API

select p from product p where p.name = 'Guitar'

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Product> cq = cb.createQuery(Product.class);
Root<Product> p = cq.from(Product.class);
cq.select(p).where(cb.equal(p.get("name"),"Guitar"));
TypedQuery<Product> tq= em.createQuery(cq);
tq.getResultList();
```

ORM Advantages

- Less code to write
- SQL dialect agnostic
- Caching and batch operations

ORM Disadvantages

- Potential performance problems
- Big abstraction (N+1 problem)
- Learning curve
- Less control over final SQL

N+1 Problem

How many SQL statements will be issued?

```
Product p  = find(1);
System.out.println(p.getName());
```

N+1 Problem

Typically one

```
select * from PRODUCT where ID = 1;
```

N+1 Problem

How many now?

```
List<Category> cats = findAll();

for (Category c : cats) {
  print(c.getProducts().size());
}
```

N+1 Problem

cats.size()+1

Beans Validation

- Validate your domain object/entities
- EntityManager will automatically detect validation annotations and enforce the constraints before persist
- You can create custom constraints

Testing JPA Implementation

- TestNG support
- Test setup

Application Server support

- EJB 2.0
- EJB 3.0 - JPA

EJB 2.1

- Application server with EJB container required
- Entity is a heavyweight component
- CMP or BMP
- JPA is preferred since EJB 3.0

EJB 3

- Java EE standard for ORM (inspired with Hibernate)
- Entity is lightweight POJO

Spring Framework support

- Spring Transactions
- Spring Emulation of Container for EntityManager
- Spring Data

What we didn't cover

- Inheritance
- Caching
- Transactions, Rollback vs detaching of entities
- Shared Primary Key, OneToOne mappings
- JoinColumn, JoinTable annotations
- Entity Lifecycle Callbacks
- Pessimistic and Optimistic Locking
- XML configuration
- And more...