

Advanced Java Programming Course

# Java Persistence API



Faculty of Information Technologies  
Industrial University of Ho Chi Minh City

# Session objectives

- Introduction to Java Persistence API
- ORM - Object/Relational Mapper
  - Entities
  - EntityManager & the Persistent Context
  - Persistence Units
  - Exceptions
  - Java Persistence Query Language
- OGM - Object/Grid Mapper
  - Introduction
  - OGM for MongoDB





# Introduction

# Introduction

- Previously we learnt about
    - JDBC
    - Data Access Objects (DAO) and Data Transfer Objects (DTO)
1. In JDBC, we "hard coded" SQL into our application
  2. Then used Data Source/Connection Pooling
  3. Then used DAO/DTO
  4. But this just "hides" implementation from our business logic, you still implement DAO with JDBC

## Issues not solved

- However,
  - We still have to understand a lot of implementation details (eg: connections, statements, resultsets etc)
  - What about relationships? Joins? Inheritance?
  - Object database impedance mismatch
- J2EE tried to solve this with "Entity Enterprise JavaBeans (EJB)"
  - Simpler alternatives included
    - Object Relational Mapping (ORM) tools:
    - e.g. Java Data Objects (JDO), **Hibernate**, iBatis, TopLink

# Notes: Object Relational Mismatch

- Object Relational Mismatch
  - SQL Types and Java Types are different
    - Databases also support SQL types differently
    - Tend to define their own internal data types e.g. Oracle's NUMBER type
    - Types must be mapped between Java and SQL/Database
    - JDBC (Generic SQL) Types are defined in `java.sql.Types`
    - java types are very rich; SQL types are more restrictive
  - How to map class to table? 1:1? 1:n?
  - How to map columns to class properties?
  - BLOB support? Streaming?
  - How to do Object Oriented design here? What about inheritance? Abstraction? Re-use?

# Java EE 5 to the rescue

- Java SE 5 added new constructs to Java language
  - Generics
  - Annotations
  - Enumerations
- Java EE 5 used these features to provide
  - Ease of development
  - "Dependency injection"
  - Meaningful defaults, "code by exception"
  - Simplified EJB
  - New Java Persistence API (JPA) replaced Entity EJB
- JPA can also be used in Java SE 5 without a container

# About JPA

- What is Java Persistence API (JPA)?
  - Database persistence technology for Java
    - Object-relational mapping (ORM) engine
    - Operates with POJO entities
    - Similar to Hibernate and JDO
  - JPA maps Java classes to database tables
    - Maps relationships between tables as associations between classes
  - Provides **CRUD** functionality
    - Create, read, update, delete





# History of JPA

- History of JPA
  - Created as part of EJB 3.0 within JSR 220
  - Released May 2006 as part of Java EE 5
  - Can be used as standalone library
- Standard API with many implementations
  - OpenJPA - <http://openjpa.apache.org/>
  - Hibernate - <http://www.hibernate.org>
  - TopLink JPA - <http://www.oracle.com/technology/jpa>
  - JPOX - <http://www.jpox.org/>



## JPA implementation

- Reference implementation: TopLink (GlassFish project)
- Most ORM vendors now have JPA interface
  - Hibernate-JPA,
  - EclipseLink (based on TopLink),
  - OpenJPA (based on BEA Kodo)
- All open source (under CDDL license)
  - Anyone can download/use source code or binary code in development or production



# The Java Persistence API Entities

# Anatomy of an Entity

- An **entity** is a plain old java object (POJO)
- The **Class** represents a **table** in a relational database.
- **Instances** correspond to **rows**
- Requirements:
  - annotated with the **javax.persistence.Entity** annotation
  - **public** or **protected**, no-argument (parameterless) constructor
  - the class must not be declared **final**
  - no methods or persistent instance variables must be declared **final**

# Requirements for Entities

- May be Serializable, but not required
  - Only needed if passed by value (in a remote call)
- Entities may extend both entity and non-entity classes
- Non-entity classes may extend entity classes
- Persistent instance variables must be declared private, protected, or package-private (*default visibility*) modifier
- No required business/callback interfaces
- Example:

```
@Entity  
class Person{  
    . . .  
}
```

# Persistent Fields and Properties


- The persistent state of an entity can be accessed:
  - through the entity's **instance variables**
  - through **JavaBeans-style properties** (getters/setters)
- Supported types:
  - primitive types, String, other serializable types, enumerated types
  - other entities and/or collections of entities
  - embeddable classes
- All fields not annotated with **@Transient** or not marked as Java **transient** will be persisted to the data store!

# Primary Keys in Entities

- Each entity must have a unique object identifier (persistent identifier)

@Entity

```
public class Employee {  
    @Id private int id;  
    private String name;  
    private Date age;  
  
    public int getId() { return id; }  
    public void setId(int id) { this.id = id; }  
    . . .  
}
```



Primary key

# Persistent Identity

- Identifier (id) in entity = primary key in database
- Uniquely identifies entity in memory and in DB
- Persistent identity types:
  - Simple id - single field/property  
`@Id int id;`
  - Compound id - multiple fields/properties  
`@Id int id;`  
`@Id String name;`
  - Embedded id - single field of PK class type  
`@EmbeddedId EmployeePK id;`



# Identifier Generation

- Identifiers can be generated in the database by specifying `@GeneratedValue` on the identifier
- Four pre-defined generation strategies:  
`AUTO, IDENTITY, SEQUENCE, TABLE`
- Generators may pre-exist or be generated
- Specifying strategy of `AUTO` indicates that the provider will choose a strategy
- Example

```
@Id  
@GeneratedValue(strategy=GenerationType.AUTO)  
private int id;
```

# Customizing the Entity Object

- In most of the cases, the defaults are sufficient
- By default the **table name** corresponds to the **unqualified name** of the class

- Customization:

```
@Entity
@Table(name = "FULLTIME_EMPLOYEE")
public class Employee{ ..... }
```

- The defaults of columns can be customized using the **@Column** annotation

```
@Id @Column(name = "EMPLOYEE_ID", nullable = false)
private String id;

@Column(name = "FULL_NAME" nullable = true, length = 100)
private String name;
```

# Entity Relationships

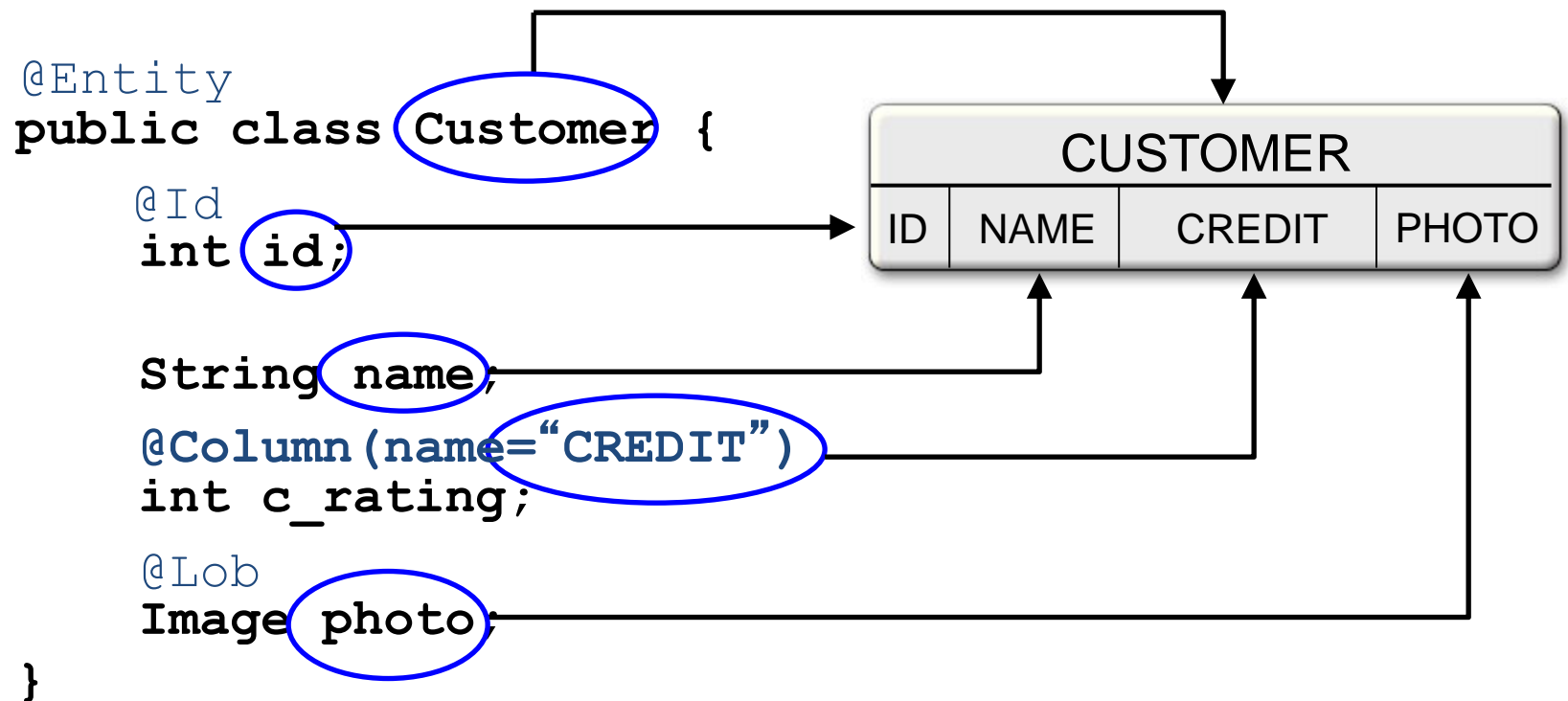
- There are four types of relationship multiplicities:
  - @OneToOne
  - @OneToMany
  - @ManyToOne
  - @ManyToMany
- The direction of a relationship can be:
  - **bidirectional** - owning side and inverse side
  - **unidirectional** - owning side only
- Owning side specifies the physical mapping

# Entity Relation Attributes

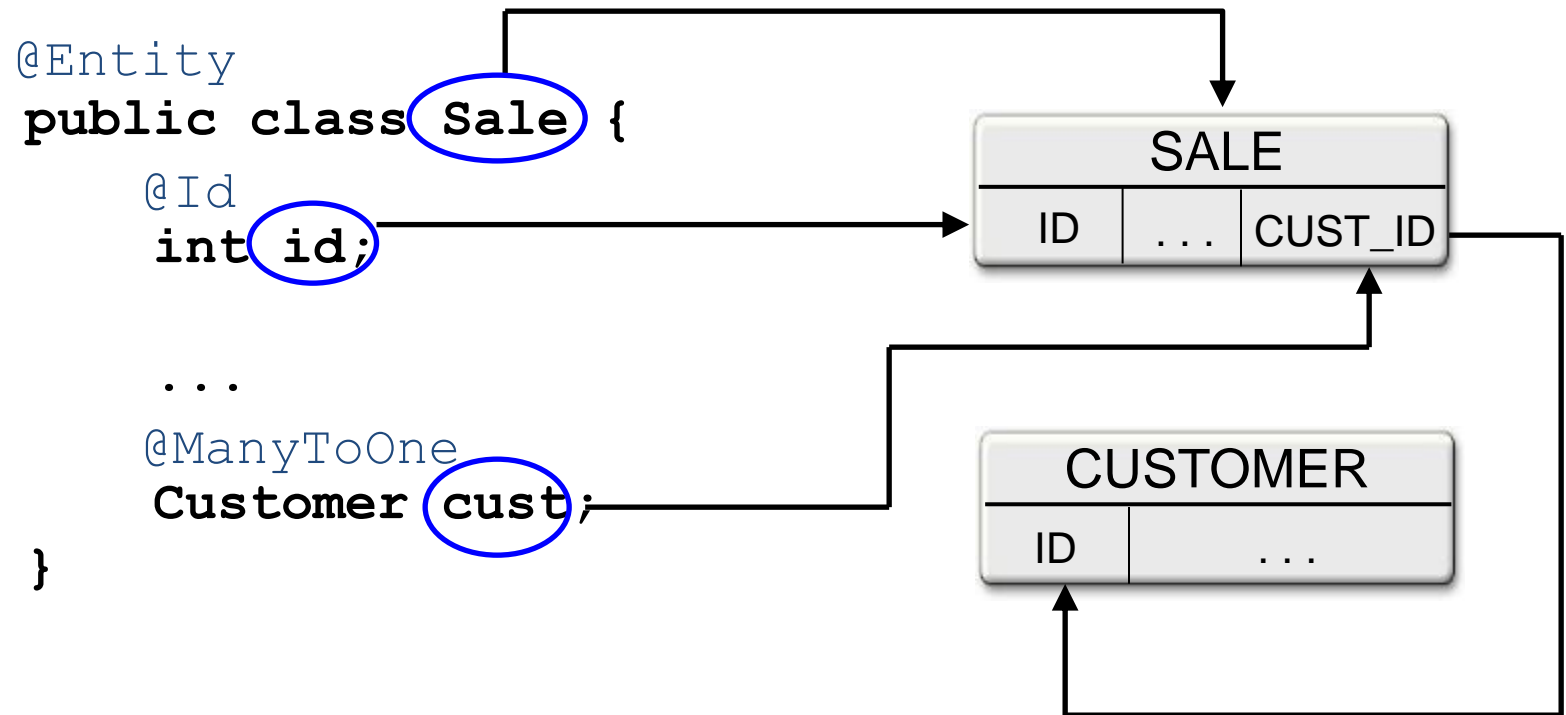
- JPA supports cascading updates/deletes
  - **CascadeType**
    - **ALL, PERSIST, MERGE, REMOVE, REFRESH**
- You can declare performance strategy to use with fetching related rows
  - **FetchType**
    - **LAZY, EAGER**
      - (Lazy means don't load row until the property is retrieved)

```
@ManyToMany(  
    cascade = {CascadeType.PERSIST, CascadeType.MERGE},  
    fetch = FetchType.EAGER)
```

# Simple Mappings



# ManyToOne Mapping



# OneToMany Mapping

```
@Entity  
public class Customer {
```

```
    @Id  
    int id;
```

```
    ...
```

```
    @OneToMany(mappedBy="cust")
```

```
    Set<Sale> sales;
```

```
}
```

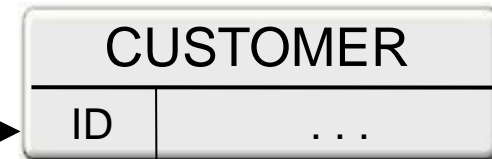
```
@Entity  
public class Sale {
```

```
    @Id  
    int id;
```

```
    ...
```

```
    @ManyToOne  
    Customer cust;
```

```
}
```



# ManyToMany Mapping

```
@Entity
public class Customer {
    ...
    @ManyToMany(cascade=CascadeType.ALL)
    @JoinTable (name="CUSTOMER_SALE",
        joinColumns=@JoinColumn (name="CUSTOMER_ID",
            referencedColumnName="customer_id"),
            inverseJoinColumns=@JoinColumn (
name="SALE_ID", referencedColumnName="sale_id")
        Collection<Sale> sales;
    }

@Entity
public class Sale {
    ...
    @ManyToMany (mappedBy="sales")
    Collection<Customer> customers;
}
```





# Persistence Units



## Persistence Unit

- A **persistence unit** defines a set of all entity classes that are managed by EntityManager instances in an application
- Each persistence unit can have different providers and database drivers
- Persistence units are defined by the **META-INF/persistence.xml** configuration file

# The persistence.xml

- A persistence.xml file defines one or more persistence units

```
<persistence-unit name="TemporalConstraint">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <class>my.package.MyEntity</class>
  <exclude-unlisted-classes>false</exclude-unlisted-classes>
  <properties>
    <property name="eclipselink.target-database" value="SQLServer" />
    <property name="eclipselink.ddl-generation" value="none" />
    <property name="javax.persistence.jdbc.url"
      value="jdbc:sqlserver://localhost:1433;databaseName=MyDatabase"/>
    <property name="javax.persistence.jdbc.user" value="username"/>
    <property name="javax.persistence.jdbc.password" value="password"/>
    <property name="javax.persistence.jdbc.driver"
      value="com.microsoft.sqlserver.jdbc.SQLServerDriver"/>
  </properties>
</persistence-unit>
```



# EntityManager & the Persistent Context

Using Persistence API

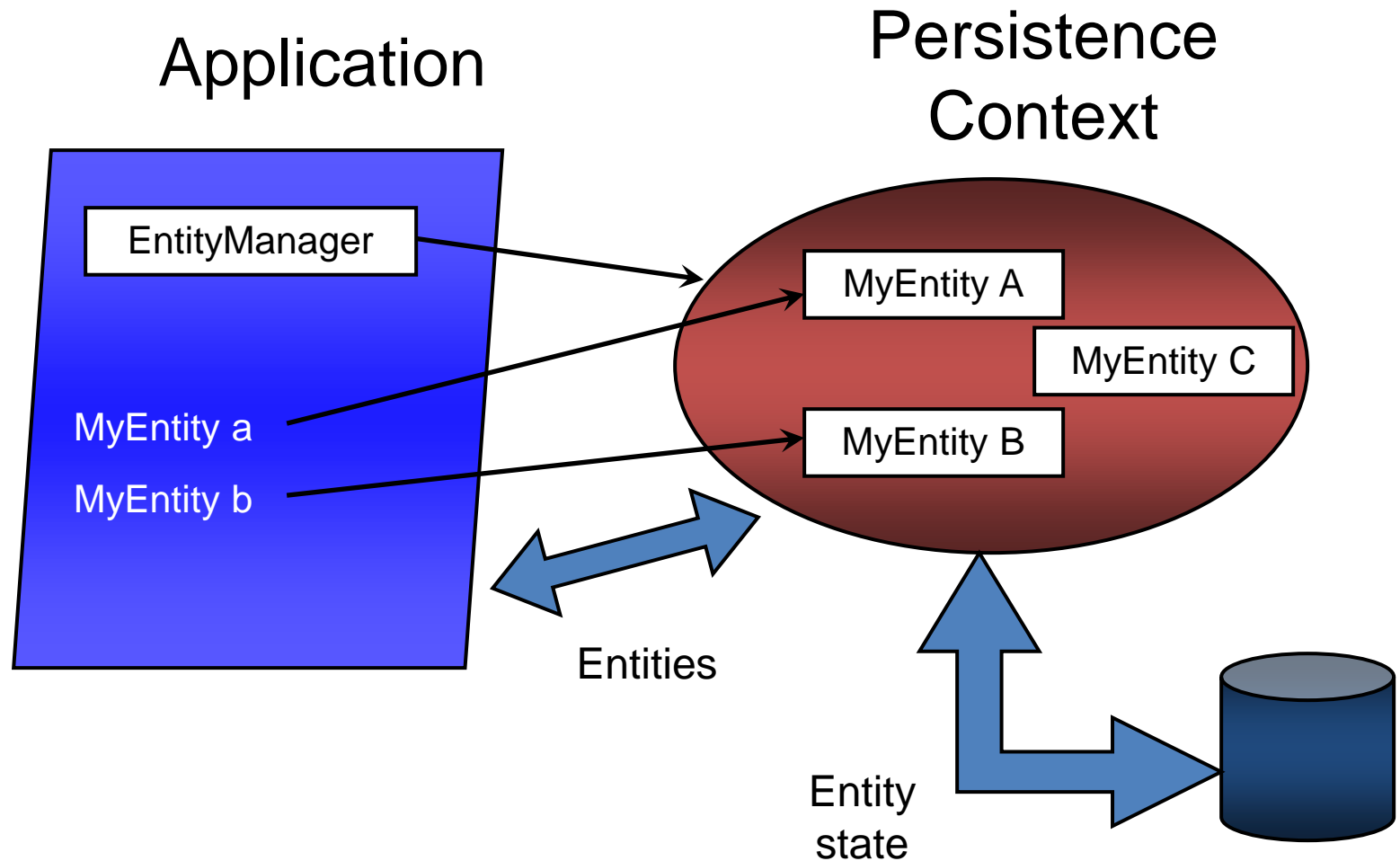
# Managing Entities

- Entities are managed by the **entity manager**
- The entity manager is represented by **`javax.persistence.EntityManager`** instances
- Each `EntityManager` instance is associated with a **persistence context**
- A persistence context defines the scope under which particular entity instances are created, persisted, and removed

# Persistence Context

- A **persistence context** is a set of managed entity instances that exist in a particular data store
  - Entities **keyed** by their persistent **identity**
  - Only one entity with a given persistent identity may exist in the persistence context
  - Entities are added to the persistence context, but are not individually removable ("detached")
- Controlled and managed by **EntityManager**
  - Contents of persistence context change as a result of operations on EntityManager API

# Persistence Context

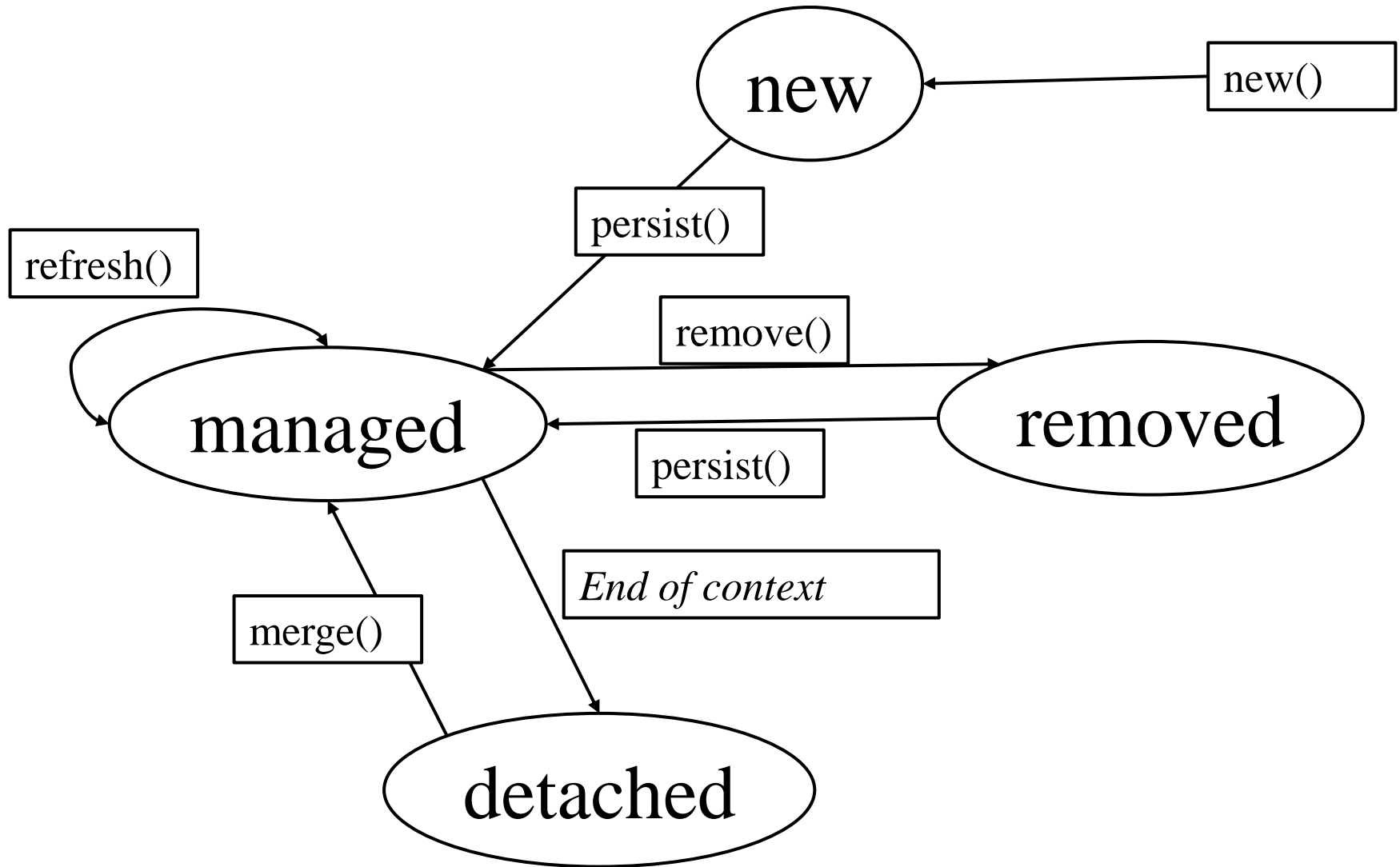


# Entity Manager

- An **EntityManager** instance is used to manage the state and life cycle of entities within a persistence context
- Entities can be in one of the following states:
  1. **New**
  2. **Managed**
  3. **Detached**
  4. **Removed**



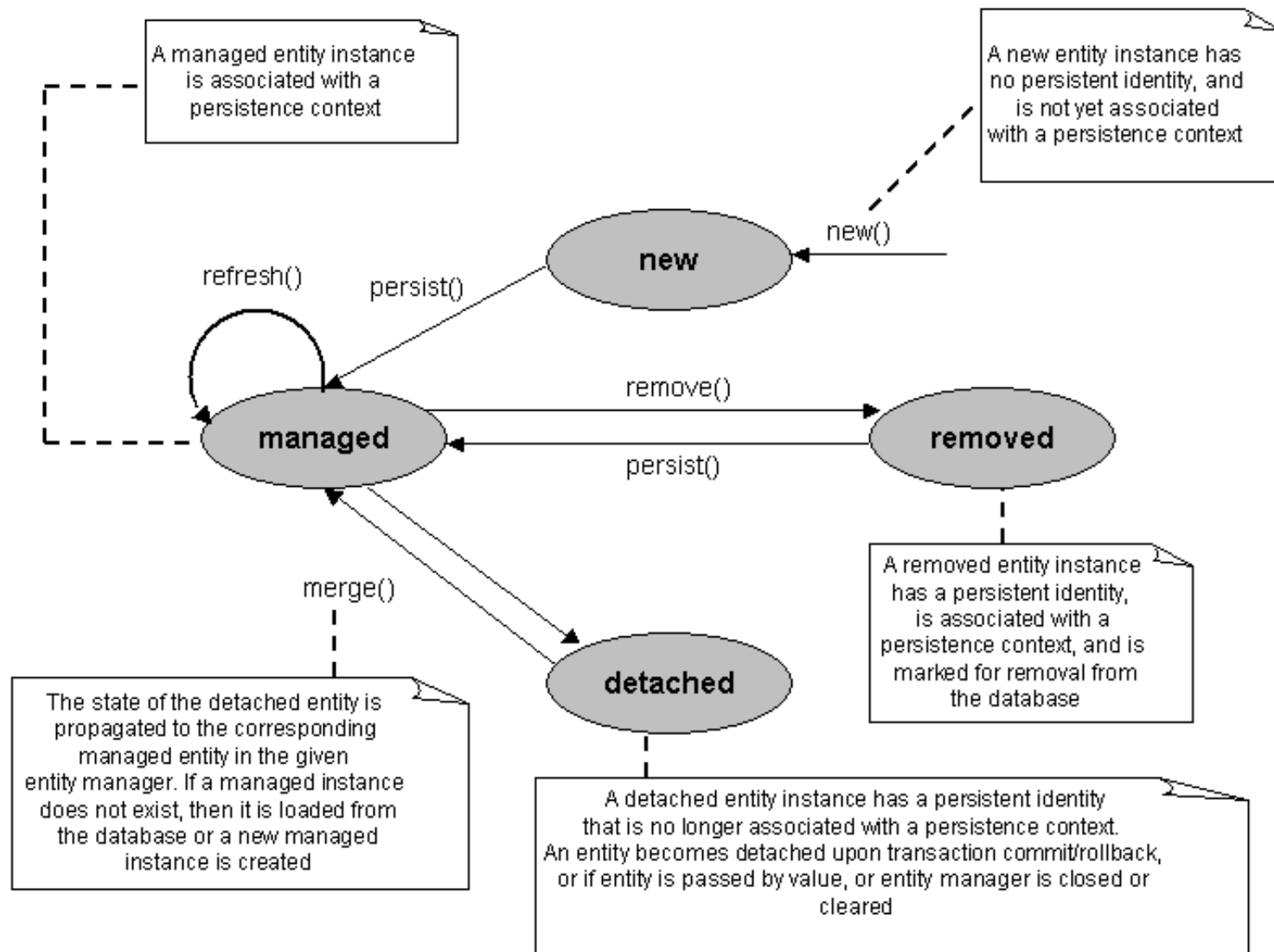
# Entity Lifecycle



# Entity Lifecycle

- **New** - entity is instantiated but not associated with persistence context. Not linked to database.
- **Managed** - associated with persistence context. Changes get synchronised with database
- **Detached** - has an id, but not connected to database
- **Removed** - associated with persistence context, but underlying row will be deleted.
- The state of persistent entities is synchronized to the database when the transaction commits

# Entity Lifecycle



# Entity Manager

- The EntityManager API:
  - creates and removes persistent entity instances
  - finds entities by the entity's primary key
  - allows queries to be run on entities
- There are two types of EntityManagers:
  - Application-Managed EntityManagers
    - ie: run via Java SE
  - Container-Managed EntityManagers
    - ie: run via Java EE Container eg: JBossAS, GlassFish,...

# Container-Managed Entity Managers (JavaEE)

- With a container-managed entity manager, an `EntityManager` instance's persistence context is automatically propagated by the container to all application components that use the `EntityManager` instance within a single Java Transaction API (JTA) transaction.
- The Java EE container manages the lifecycle of container-managed entity managers.
- To obtain an `EntityManager` instance, inject the entity manager into the application component:

`@PersistenceContext`

`private EntityManager em;`

# Application-Managed EntityManager (JavaSE)

- Java SE applications create EntityManager instances by using **directly Persistence** and **EntityManagerFactory**:
  - **javax.persistence.Persistence**
    - Root class for obtaining an EntityManager
    - Locates provider service for a named persistence unit
    - Invokes on the provider to obtain an EntityManagerFactory
  - **javax.persistence.EntityManagerFactory**
    - Creates EntityManagers for a named persistence unit or configuration

```
EntityManagerFactory fac =  
    Persistence.createEntityManagerFactory("JPADemo");  
EntityManager em = fac.createEntityManager();
```

# Entity Transactions (In JavaSE)

- Only used by resource-local EntityManagers
- Transaction demarcation under explicit application control using EntityTransaction API  
`begin()`, `commit()`, `rollback()`, `isActive()`
- Underlying (JDBC) resources allocated by EntityManager as required

```
EntityTransaction trs = entityManager.getTransaction();
try {
    trs.begin();
    //do your works...
    trs.commit();
} catch (Exception e) {
    trs.rollback();
}
```

# Operations on Entity Objects

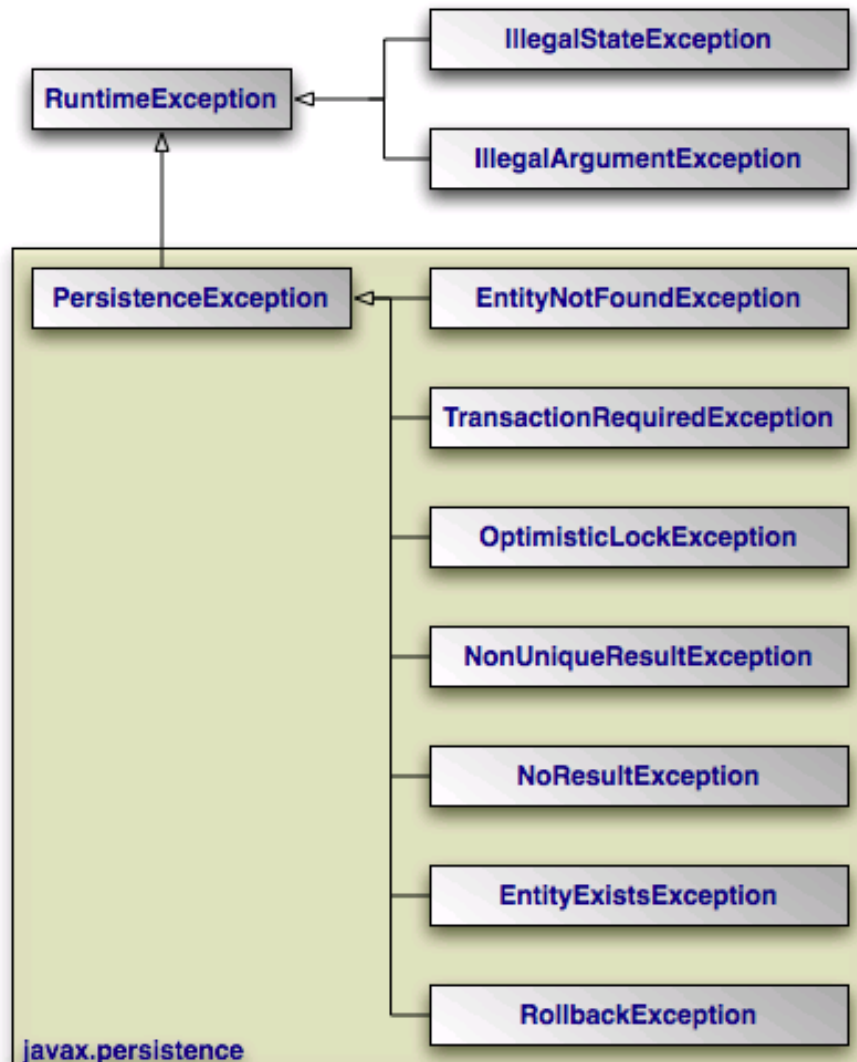
- EntityManager API operations:
  - `persist()` - Save the entity into the db
  - `remove()` - Delete the entity from the db
  - `refresh()` - Reload the entity state from the db
  - `merge()` - Synchronize a detached entity with the p/c
  - `find()` - Find by primary key
  - `createQuery()` - Create query using dynamic JP QL
  - `createNamedQuery()` - Create a predefined query
  - `createNativeQuery()` - Create a native "pure" SQL query. Can also call stored procedures.
  - `contains()` - Is entity is managed by p/c
  - `flush()` - Force synchronization of p/c to database
- Note: p/c == the current persistence context





# JPA exceptions

# JPA exceptions




- All exceptions are unchecked
- Exceptions in `javax.persistence` package are self-explanatory



# JPA Query Language (JPAQL)

JPA Query Language

- 
- JPA has a query language based on SQL
  - JPQL is an extension of EJB QL
  - More robust flexible and object-oriented than SQL
  - The persistence engine parses the query string, transform the JPQL to the native SQL before executing it

# Creating Queries

- Query instances are obtained using:
  - `EntityManager.createNamedQuery` (static query)
  - `EntityManager.createQuery` (dynamic query)
  - `EntityManager.createNativeQuery` (native query)
- Query API:
  - `getResultList()` - execute query returning multiple results
  - `getSingleResult()` - execute query returning single result
  - `executeUpdate()` - execute bulk update or delete
  - `setFirstResult()` - set the first result to retrieve
  - `setMaxResults()` - set the maximum number of results to retrieve
  - `setParameter()` - bind a value to a named or positional parameter
  - `setHint()` - apply a vendor-specific hint to the query
  - `setFlushMode()` - apply a flush mode to the query when it gets run

# Static (Named) Queries

- Defined statically with the help of `@NamedQuery` annotation together with the entity class
- `@NamedQuery` elements:
  - `name` - the name of the query that will be used with the `createNamedQuery` method
  - `query` - query string

```
@NamedQuery (name="Customer.findAll",  
              query="SELECT c FROM Customer c")
```

```
Query findAllQuery = entityManager.createNamedQuery(  
    "findAllCustomers");  
List customers = findAllQuery.getResultList();
```

# Multiple Named Queries

- Multiple named queries can be logically defined with the help of `@NamedQueries` annotation

```
@NamedQueries (  
    {  
        @NamedQuery(name = "Mobile.selectAllQuery"  
            query = "SELECT M FROM MOBILEENTITY M") ,  
        @NamedQuery(name = "Mobile.deleteAllQuery"  
            query = "DELETE M FROM MOBILEENTITY M")  
    }  
)
```

# Dynamic Queries

- Dynamic queries are queries that are defined directly within an application's business logic
- **Not efficient & slower.** Persistence engine has to parse, validate & map the JPQL to SQL at run-time

```
public List findAll(String entityName) {  
    return entityManager.createQuery(  
        "select e from " + entityName + " e")  
        .getResultList();  
}
```



# Named Parameters

- Named parameters are parameters in a query that are prefixed with a **colon** (:)
- To bound parameter to an argument use method:
  - **Query.setParameter**(String name, Object value)

```
public List findWithName(String name) {  
    return em.createQuery(  
        "SELECT c FROM Customer c WHERE c.name LIKE :custName")  
        .setParameter("custName", name)  
        .getResultList();  
}
```

# Positional Parameters

- Positional parameters are prefixed with a **question mark (?)** & **number of the parameter in the query**
  - To set parameter values use method:

`Query.setParameter(integer position, Object value)`

```
public List findWithName(String name) {  
    return em.createQuery(  
        "SELECT c FROM Customer c WHERE c.name LIKE ?1")  
        .setParameter(1, name)  
        .getResultList();  
}
```

# Native Queries

- Queries may be expressed in native SQL
- Use when you need to use native SQL of the target database
- Can call stored procedures using "call procname" syntax

```
Query q = em.createNativeQuery(  
    "SELECT o.id, o.quantity, o.item " +  
    "FROM Order o, Item i " +  
    "WHERE (o.item = i.id) AND (i.name = 'widget')",  
    com.acme.Order.class);
```

Use @SqlResultSetMapping annotation for more advanced cases

# Query Operations - Multiple Results

- `Query.getResultList()` will execute a query and may return a List object containing multiple entity instances

```
Query query = entityManager.createQuery("SELECT C FROM CUSTOMER");  
List<MobileEntity> mobiles = (List<MobileEntity>)query.getResultList();
```

- Will return a non-parameterized List object
- Can only execute on select statements as opposed to UPDATE or DELETE statements
- For a statement other than SELECT run-time `IllegalStateException` will be thrown

# Query Operations - Single Result

- A query that returns a single entity object

```
Query singleSelectQuery = entityManager.createQuery(  
    "SELECT C FROM CUSTOMER WHERE C.ID = 'ABC-123'");  
Customer custObj = singleSelectQuery.getSingleResult();
```

- If the match wasn't successful, then `EntityNotFoundException` is returned
- If more than one matches occur during query execution a runtime exception `NonUniqueResultException` will be thrown

# Paging Query Results

```
int maxRecords = 10; int startPosition = 0;
String queryString = "SELECT M FROM MOBILEENTITY";
while(true){
    Query selectQuery = entityManager.createQuery(queryString);
    selectQuery.setMaxResults(maxRecords);
    selectQuery.setFirstResult(startPosition);
    List<MobileEntity> mobiles =

        entityManager.getResultList(queryString);
    if (mobiles.isEmpty()){ break; }
    process(mobiles);           // process the mobile entities
    entityManager.clear();      // detach the mobile objects
    startPosition = startPosition + mobiles.size();
}
```

# Flushing Query Objects

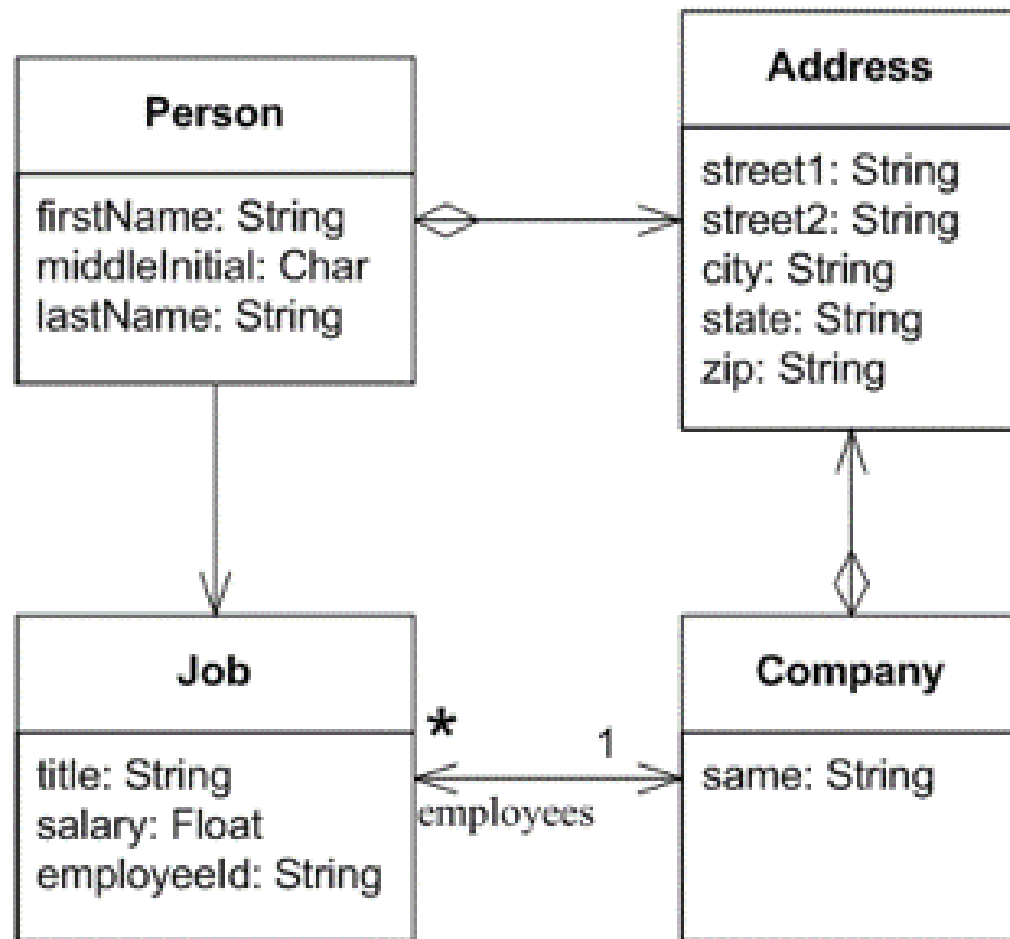
- Two modes of flushing query objects
  - **AUTO** (default) and **COMMIT**
- **AUTO** - any changes made to entity objects will be reflected the very next time when a `SELECT` query is made
- **COMMIT** - the persistence engine may only update all the state of the entities during the database `COMMIT`
- set via `Query.setFlushMode()`

# Exercises

1. Define an entity class `Student` which has `Id`, `FirstName` and `LastName`.
2. Define an entity class `Course` which has `Id`, `name` and list of students.
3. Create a database matching the entity classes. Use Apache Derby and its built-in identity columns support.
4. Create a program that lists all classes and the students in each class.
5. Create a program that adds a new class and few students inside it.



# Exercises



Define an entity class and mapping (relationship) among these objects



## Summary

# The Java Persistence API

- Entities
- EntityManager & the Persistent Context
- Persistence Units
- Exceptions
- JPA Query Language

# FAQ





*That's all for this session!*

Thank you all for your attention and patient !