Hibernate Framework

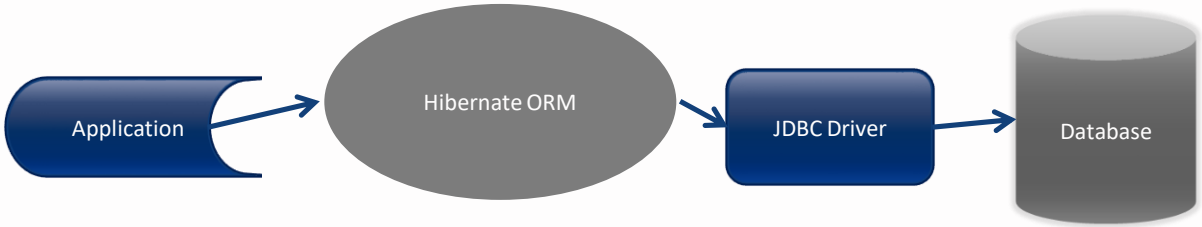
Object Relational Mapping (ORM)

- A major part of any enterprise application development project is the persistence layer
  - Accessing and manipulate persistent data typically with relational database
- ORM handles Object-relational impedance mismatch
  - Data lives in the relational database, which is table driven (with rows and columns)
- Relational database is designed for fast query operation of table-driven data
  - Work with objects, not rows and columns of table

Object Relational Mapping (ORM)

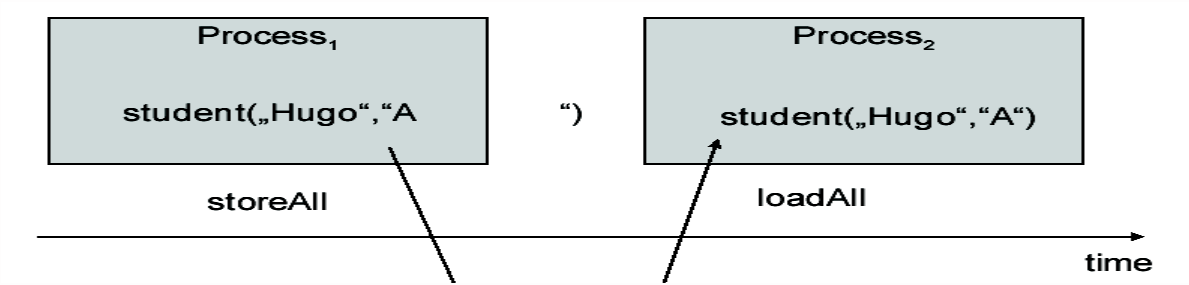
- Automated persistence of object to tables in RDBMS
- Usually with the help of metadata that describes the mapping
  - SQL is auto-generated by the metadata description
- An ORM Solution consists of the following pieces:
  - Persistence Manager with CRUD API
  - Query API
  - Mapping metadata

Hibernate Overview



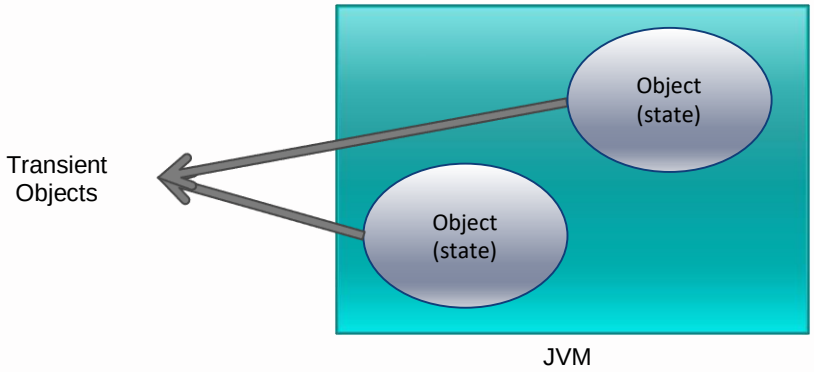
What is Persistence ?

- Ability of an object to survive even current session or program terminate
- The ability of an object to remain in existence past the lifetime of the program that creates it



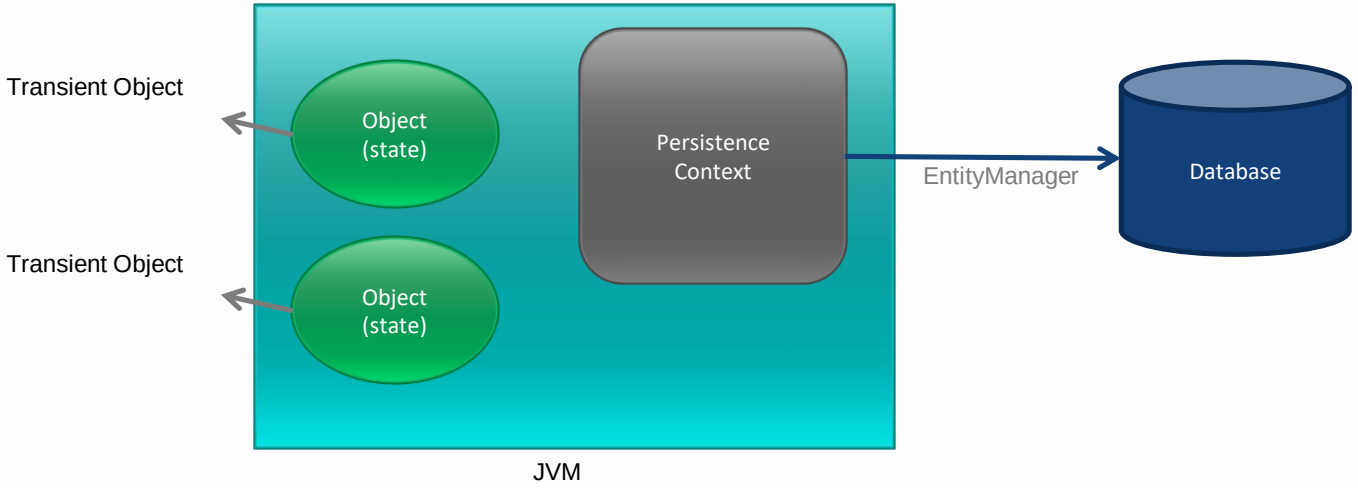
Transient Object

- Is an instance of a class created within the JVM process scope and valid as long as reference to the same exists
- Modifying the state of transient object does not affects the database



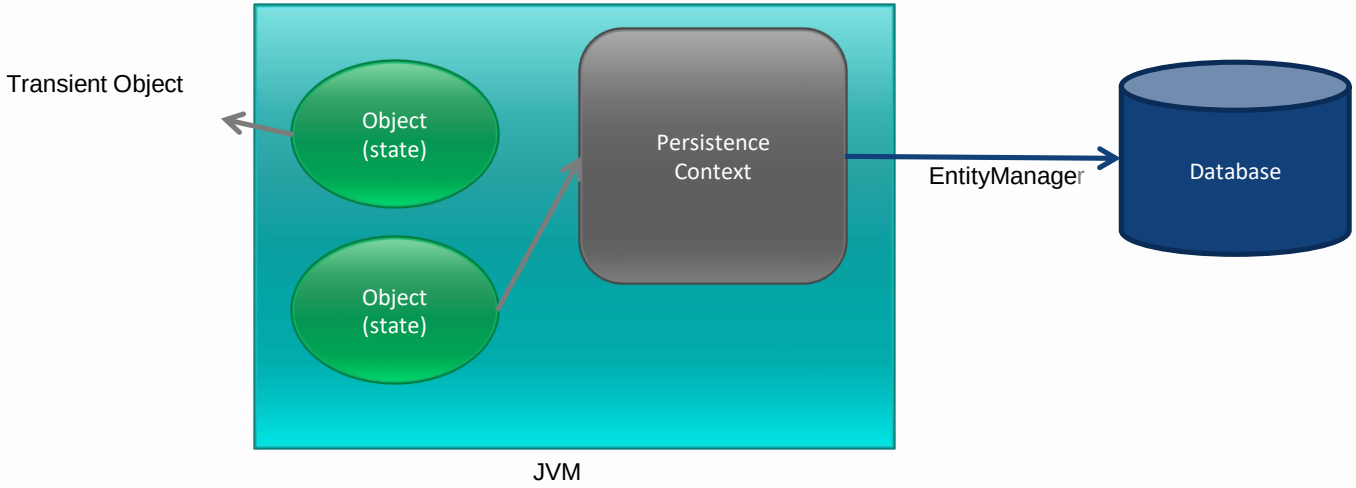
Managed/Persistent Object

- As soon as a transient object is associated with a persistence context, it's a persistent object
- Modifying the state of a persistent object will be synchronized with the underlying database



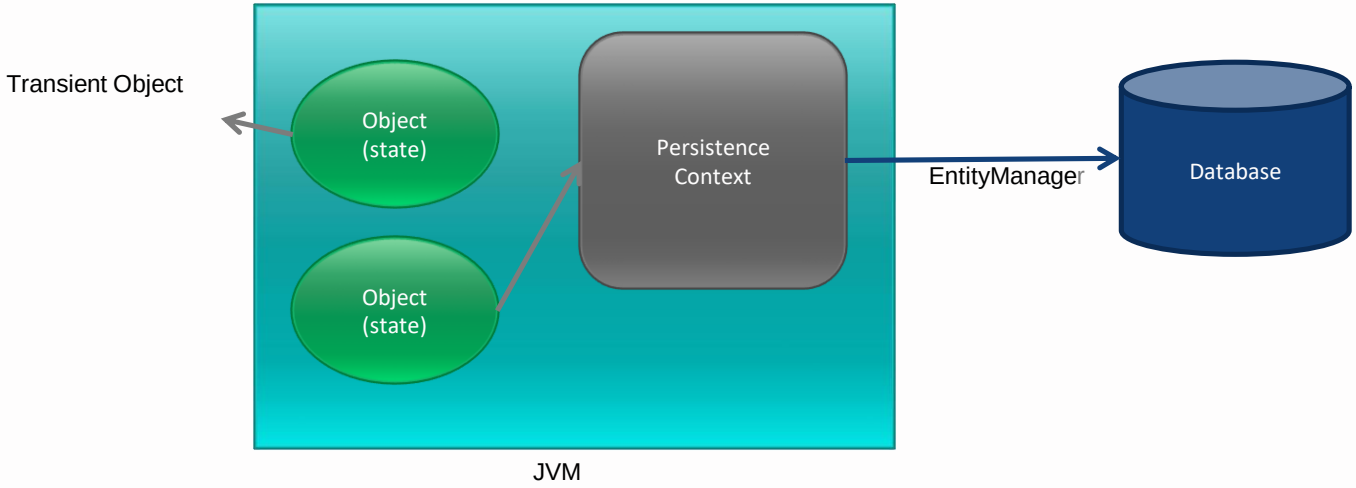
Managed/Persistent Object

- As soon as a transient object is associated with a persistence context, it's a managed object
- Modifying the state of a managed object will be synchronized with the underlying database



Managed/Persistent Object

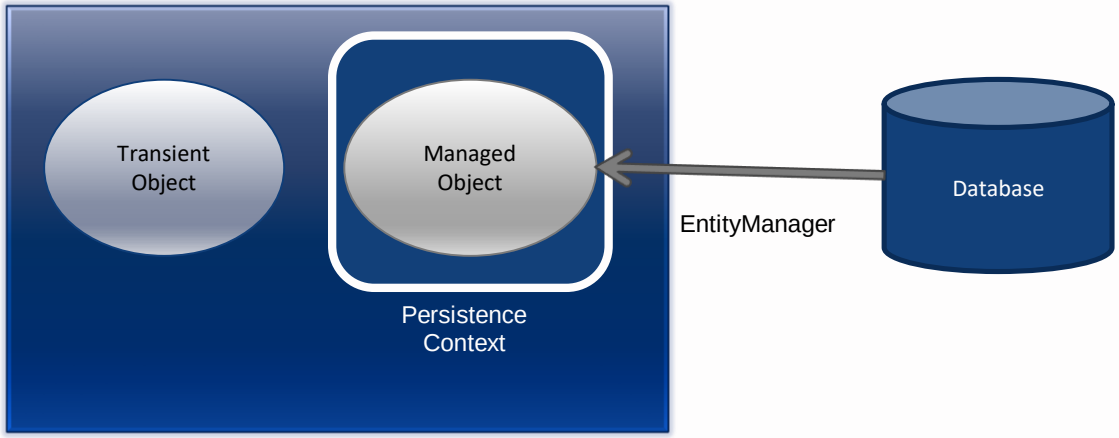
- As soon as a transient object is associated with a persistence context, it's a managed object
- Modifying the state of a managed object will be synchronized with the underlying database



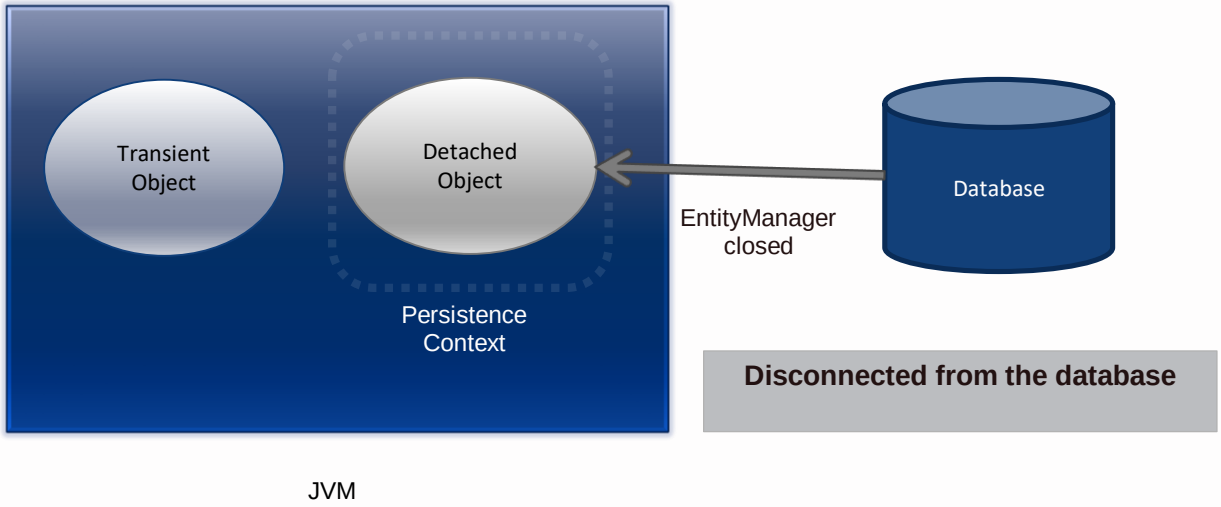
Detached Object

- An object which was loaded in some persistence context but the context has been closed on behalf of some transactional process being committed/rolled back
- Modifying state of detached instance will not be updated in the database till not reattached with some persistence context

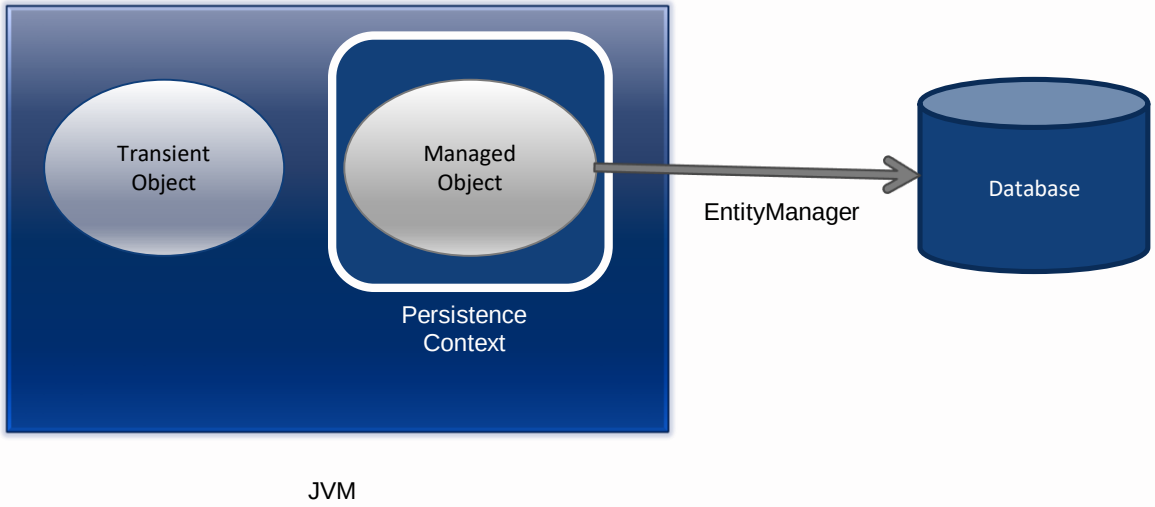
Detached Object



Detached Object

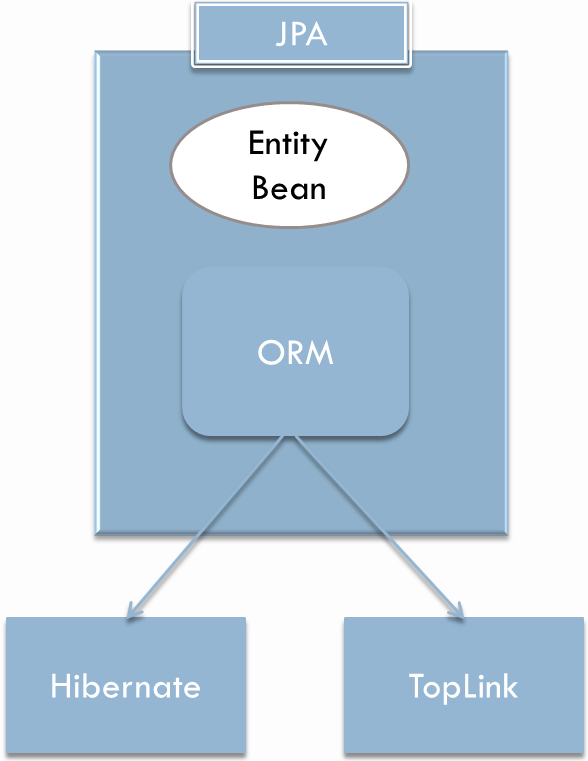


Detached Object



Hibernate vs. JPA

- Hibernate is an ORM which is compliant with a standard API called as JPA (Java Persistence API)
- Like Hibernate we have many other ORMs in Java like TopLink, EclipseLink and others
- Hibernate has it's own API made up of SessionFactory, Session, hbm(hibernate mapping) xml files, etc… but we use the standard API now a days in favour of Hibernate API
- Using JPA loosely couples our code with the underlying ORM and shifting from one ORM to another will not affect the code badly



What is an EntityManager?

- Manages the state and life-cycle of entities
  - Creates and removes entity instances within the persistence context
- Handles querying entities within a persistence context
  - Performs finding entities via their primary keys
- Lock entities
- Accessible through EntityManager Java interface
  - The life-cycle operations are defined in the EntityManager interface
- Similar in functionality to Hibernate Session

Types of EntityManager

- Application-Managed Entity Manager (Java SE environment)
  - Entity manager is created and managed by the application
- Container-Managed Entity Manager (Java EE environment)
  - Entity manager is created and managed by the Container
  - Entity manager will be provided to the application via dependency injection

Application Managed EntityManager

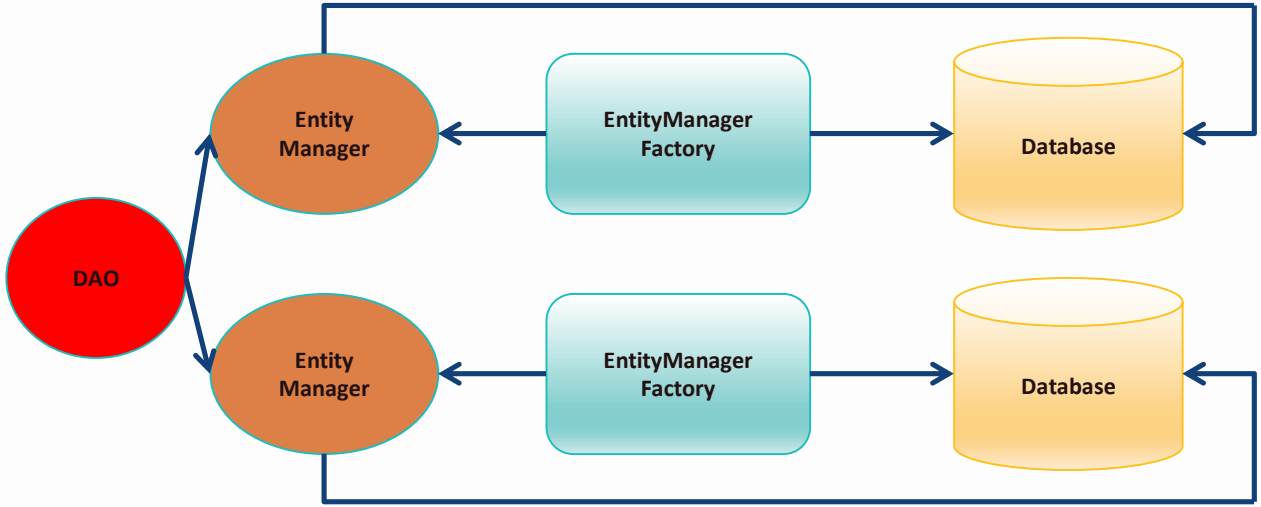
```
// Application is responsible for explicitly obtaining
Entity Manager // and life-cycle of it
EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("EmployeeS
ervice");
EntityManager em = emf.createEntityManager();

Collection emps = em.createQuery("SELECT e FROM Employee
e") .getResultList();
// Some code

em.close();
emf.close();
```

Container Managed EntityManager

```
@Stateless
public class OrderEntry {

// Entity Manager is created & injected by the container
@PersistenceContext EntityManager em;

    public void enterOrder(int custID, Order newOrder){
        // Use find method to locate customer entity
        Customer c = em.find(Customer.class, custID);
        // Add a new order to the Orders
        c.getOrders().add(newOrder);
        newOrder.setCustomer(c);
    }
    // No need to close EntityManager
}
```

Using JPA instead of Hibernate API

- In JPA, the names of the interfaces are different from Hibernate
- SessionFactory becomes EntityManagerFactory
- Session becomes EntityManager



Cont'd…

- In JPA, all ORM specific configuration is done by default in persistence.xml file. This file should be present in the META-INF folder of our project. This file will is read whenever we create the EntityManagerFactory object

- Even in JPA like Hibernate, we can map entities using xml, but that's not the general practice. Annotations are a preferred way of providing entity metadata in JPA

META-INF/persistence.xml file

```
<persistence-unit name="Hibernate-JPA">

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

<properties>
<!-- JPA 2.0 -->
<property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbc.JDBCDriver" />
<property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:hsql://localhost/db" />
<property name="javax.persistence.jdbc.user" value="sa" />
<property name="javax.persistence.jdbc.password" value="" />
<!-- JPA 2.1 -->
<!-- <property name="javax.persistence.schema-generation.database.action" value="drop-and-create" /> -->

<!-- ORM Specific -->
<property name="hibernate.show_sql" value="true" />
<property name="hibernate.hbm2ddl.auto" value="update" />

</properties>

</persistence-unit>
```

Some of the API methods

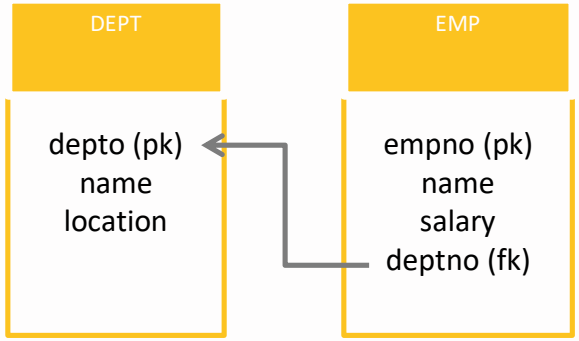
```
CD cd = (CD) session.get(CD.class, 1);

CD cd = (CD) entityManager.find(CD.class, 1);

session.update(cd);

session.saveOrUpdate(cd);

session.merge(cd);

entityManager.merge(cd);

session.delete(cd);

entityManager.remove(cd);
```

- get/find method is used for fetching a record based on the pk column
- update method is used for updating a detached object in the database
- saveOrUpdate can be used for insert/update depending on whether the object is transient/detached
- merge also can be used for the same purpose like saveOrUpdate. Can be used for insert as well as update both
- delete/remove deletes a record from the database

Associations

- The most common form of association is the *one to many* association. For ex: Customer->Order, Order->LineItem, Department->Employee, etc…
- All forms of association, *one-to-one*, *one-to-many*, *many-to-many* can be represented in an unidirectional as well as bidirectional fashion when writing the mapping classes
- Generally projects prefer bidirectional association

one to many association



In this example, Department and Employee class represent a bi-directional *one-to-many* association

```
public class Department {

 private int deptno;
 private String name;
 private String location;
 private Set<Employee> employees;
```

```
  public class Employee {

   private int empno;
   private String name;
   private double salary;
   private Department dept;
```

Important settings

- There are two important properties in Hibernate/JPA to control how the relationship between the entities is managed by the application. One is **fetch** and the other is **cascade**
- Fetching strategy helps us control what happens when one end of the relationship is fetched, will Hibernate automatically fetch the other end of the association or not
- In JPA, the default one-to-many & many-to-many fetching strategy is **lazy** while one-to-one and many-to-one fetching strategy is **eager**
- Cascade property allows us to control how to store/update/delete associated entity data in the database

Inheritance mapping

- Hibernate as well as JPA supports three basic inheritance mapping strategies
  - Single table per class hierarchy
  - Table per subclass
  - Table per concrete class

HQL/JPQL (Hibernate/Java Persistence Query Language)

- HQL/JPQL allows developers to write queries transparent to the differences which arises when using databases
- HQL leverages the same syntax of SQL so learning a new QL doesn't requires lot of time
- HQL queries directly return collection of objects, so there is no need to worry about resultset translation
- For ex:
  - select item from Item as item where item.initialPrice > 10000
  - Will return object of Item entity

Locking Support in Hibernate/JPA

- We are looking out for ways by which we can prevent concurrent updates at the same time
- Locking of the row is a way by which we can easily achieve the same
- Hibernate supports both the forms of locking:
  - Optimistic Locking (relies on version/timestamp column)
  - Pessimistic Locking (relies on database to manage row level locks)

LTI
Let's Solve

Hibernate

Quiz

Which annotation in JPA is used to customize the mapping of a field/property of an entity class to a column in the table?

- @Field
- @Property
- @Column
- @Mapping

What does hibernate.hbm2ddl.auto property helps us with?

- Create tables automatically
- Create EntityManager object automatically
- Create EntityManagerFactory object automatically
- Create database user automatically

Let's Solve