Module – JPA with Hibernate 3.0

Module Overview

Hibernate - This is an ORM (Object Relation Mapping) framework for data layer of software application. ORM framework helps in converting data into POJO (plain java object) and also provide other capabilities like default SQL operations like insert, delete, read and update, user can also create custom queries, caching of data, etc.

Module Objective

At the end of this module, students should be able to demonstrate appropriate knowledge, and show an understanding of the following:

- Understanding Object Relation Mapping
- Learning JPA API
- Understanding the requirement of entity classes
- Understand the importance of persistent fields, properties
- Know the steps to validate the persistent fields
- Understand the use of primary keys in entities
- Understand the steps to manage entities
- Understand the importance of JPQL and what is criteria API
- Understand the entity relationships.

Data Persistence

What is Data Persistence?

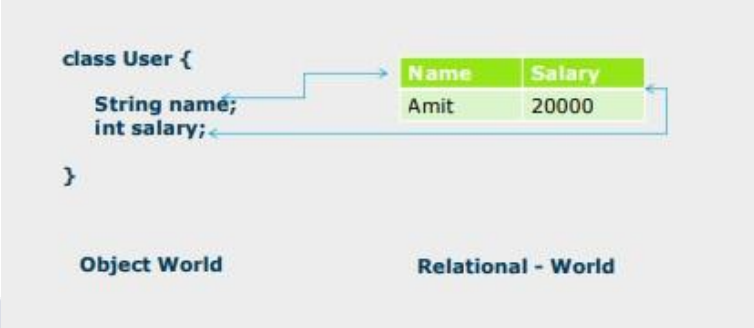
Data persistence involves saving data in a non-volatile storage system so that the data's value can be retrieved reliably later. Data can take many forms, including structured, unstructured, and semi-structured formats, so there are a variety of storage technologies designed to preserve the different types of data in their proper structure, including any metadata that describes the origin, format, or history of that data. Some examples include relational database management systems, key-value stores, NoSQL databases, Hadoop distributed file systems, and cloud data warehouses. Each technology has advantages and disadvantages in the way of cost, performance, reliability, latency, and access methods.

Why is Data Persistence Important?

Data persistence is required for data science and machine learning because the fuel for analysis comes from collecting comprehensive data sets that represent historical behavior as well as current operational input. Although data can be stored locally within a data science platform, it more commonly resides on internal or external data stores, or is consolidated into a data lake, or can be accessed from federated virtual data sources.

What is Object Persistence?

Persistence means to make application's data to outlive the applications process.
In Java terms, the objects to live beyond the scope of the JVM so that the same state is available later.



The above diagram depicts mapping of object state into database table columns. To do so, traditionally, we rely on JDBC API, which allows developers to save application data into database, however conversion is required from object format to database table format which un-necessarily increases line of code.

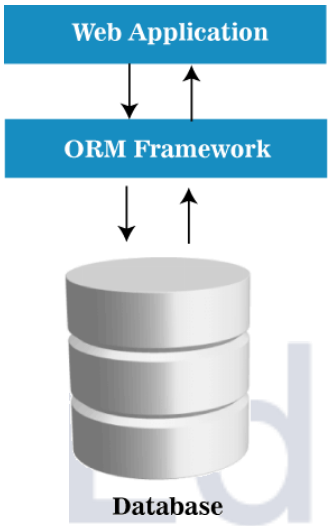
However, there are lot of challenges and mismatch in data processing in these two models. In addition, if database changes, then developer need to make modification in the configuration which is database specific.

So, to shorten the development time, and to save application object directly into database, there was a need to reinvent the approach of mapping object and relational model.

Overview of ORM tools

ORM stands for Object Relation Mapping. It is a middleware application or tool that sits between the web application and database. It wraps the implementation specific details of storage drivers in an API.

ORM is a technique for converting data between Java objects and relational databases (table). In simple words, we can say that the ORM implements responsibility of mapping the object to relational model and vice-versa. The ORM tool does mapping in such a way that model class becomes a table in the database and each instance becomes a row of the table.



Introduction ORM

Storing object-oriented entities in a relational database is often not a simple task and requires a great deal of repetitive code along with conversion between data types.

Object-relational mapper, or O/RM, were created to solve this problem. An O/RM persists entities in and retrieves entities from relational databases without the programmer having to write SQL statements and translate entity properties to statement parameters and result set columns to entity properties.

It consists of:

- An API, to perform CRUD operations on objects of persistent classes
- A language to specify queries that refer to classes and properties of classes
- A facility, to specify mapping metadata
- A technique, for the ORM implementation to interact with transactional objects to perform dirty checking. Lazy association, fetching, and other optimization functions.

Dirty Checking:

A dirty checking feature avoids unnecessary database write actions by performing SQL updates only on the modified fields of persistent objects. For example, if you modify salary of employee on object model, only salary field will be updated instead of updating entire employee object.

<u>Lazy association fetching:</u>

Lazy fetching decides whether to load child objects while loading the Parent Object. For example, consider department entity consist of many employees, and someone query to fetch department details, ORM fetches only department details and defers loading employees. This will be done, when one request details of employees working in that department.

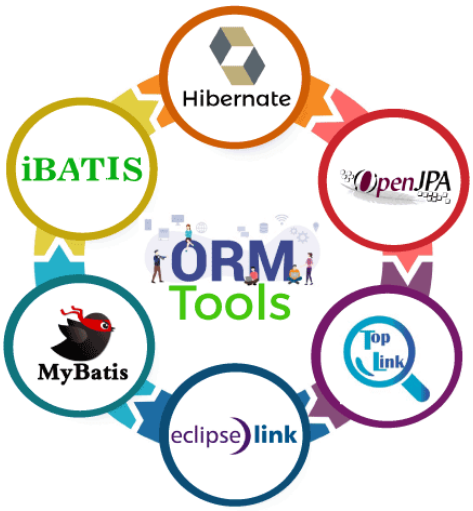
Why ORM?

- It “shields” developers from “messy” SQL.
- ORM tools allows developers to focus on the business logic of the application rather than repetitive CRUD (Create Read Update Delete) logic.
- Some of the benefits of ORM are:
  - Productivity
  - Maintainability
  - Performance
  - Vendor independence

ORM Tools

There are many ORM tools available but the following ORM tools are the most commonly used.

- **Hibernate** is a Java persistence framework that simplifies the development of Java application to interact with the database.
- **TopLink** is an ORM tool provides development tools and run-time functionalities that ease the development process and increases the functionality.
- **EclipseLink** is an extensible framework that allows Java developers to interact with various data services such as databases, web services, Object XML mapping, and enterprise information systems.
- Apache **OpenJPA** is a Java persistence project at The Apache Software Foundation that can be used as a stand-alone POJO persistence layer or integrated into any Java EE compliant container and many other lightweight frameworks, such as Tomcat and Spring.
- **MyBatis** is an open source persistence framework that simplifies the implementation of database.



Hibernate Framework

Hibernate is a Java framework that simplifies the development of Java application to interact with the database. It is an open source, lightweight, ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.

Hibernate maps Java classes to database tables and from Java data types to SQL data types and relieves the developer from 95% of common data persistence related programming tasks.

Hibernate sits between traditional Java objects and database server to handle all the works in persisting those objects based on the appropriate O/R mechanisms and patterns.

Downloading Hibernate

It is assumed that you already have the latest version of Java installed on your system. Following are the simple steps

to download and install Hibernate on your system –

Make a choice whether you want to install Hibernate on Windows, or Unix and then proceed to the next step to download .zip file for windows and .tz file for Unix.

Download the latest version of Hibernate from http://www.hibernate.org/downloads.

At the time of writing this tutorial, I downloaded hibernate-distribution3.6.4.Final and when you unzip the downloaded file, it will give you directory structure as shown in the following image

Name	Date modified	Type	Size
documentation	04/05/2011 15:16	File folder	
lib	04/05/2011 15:15	File folder	
project	04/05/2011 14:44	File folder	
changelog.txt	04/05/2011 14:44	Text Document	224 KB
hibernate_logo.gif	04/05/2011 14:44	GIF image	2 KB
hibernate3.jar	04/05/2011 15:15	Executable Jar File	4,037 KB
hibernate-testing.jar	04/05/2011 15:03	Executable Jar File	44 KB
lgpl.txt	04/05/2011 14:44	Text Document	26 KB

Installing Hibernate

Once you downloaded and unzipped the latest version of the Hibernate Installation file, you need to perform the following two simple steps. Make sure you are setting your CLASSPATH variable properly otherwise you will face problem while compiling your application.

- Now, copy all the library files from /lib into your CLASSPATH, and change your classpath variable to include allthe JARs.
- Finally, copy hibernate3.jar file into your CLASSPATH. This file lies in the root directory of the installation and is the primary JAR that Hibernate needs to do its work.

Hibernate Configuration

Hibernate requires to know in advance — where to find the mapping information that defines how your Java classes relate to the database tables. Hibernate also requires a set of configuration settings related to database and other related parameters. All such information is usually supplied as a standard Java properties file called hibernate.properties, or as an XML file named hibernate.cfg.xml.

I will consider XML formatted file hibernate.cfg.xml to specify required Hibernate properties in my examples. Most of the properties take their default values and it is not required to specify them in the property file unless it is really required. This file is kept in the root directory of your application's classpath.

Hibernate Properties

Following is the list of important properties; you will be required to configure for a databases in a standalone situation−

Sr.No.	Properties & Description
1	**hibernate.dialect** This property makes Hibernate generate the appropriate SQL for the chosen database.
2	**hibernate.connection.driver_class** The JDBC driver class.
3	**hibernate.connection.url** The JDBC URL to the database instance.
4	**hibernate.connection.username** The database username.
5	**hibernate.connection.password** The database password.
6	**hibernate.connection.pool_size** Limits the number of connections waiting in the Hibernate database connection pool.
7	**hibernate.connection.autocommit** Allows autocommit mode to be used for the JDBC connection.

If you are using a database along with an application server and JNDI, then you would have to configure the following properties –

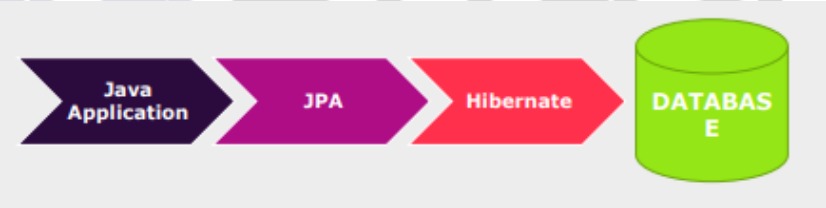
Sr.No.	Properties & Description
1	**hibernate.connection.datasource** The JNDI name defined in the application server context, which you are using for the application.
2	**hibernate.jndi.class** The InitialContext class for JNDI.
3	**hibernate.jndi.<JNDIpropertyname>** **Passes any JNDI property you like to the JNDI InitialContext.**
4	**hibernate.jndi.url** **Provides the URL for JNDI.**

5	**hibernate.connection.username** **The database username.**
6	**hibernate.connection.password** **The database password.**

Introduction to Java Persistence API

JPA is just a specification from Sun, which is released under JEE 5specification. JPA standardized the ORM persistence technology for Java developers. JPA is not a product and can't be used as it is for persistence. It needs an ORM implementation to work and persist the Java Objects. ORM frameworks that can be used with JPA are Hibernate, Top link, Open JPA etc.

The Java Persistence API (JPA) is one approach to ORM. Via JPA the developer can map, store, update and retrieve data from relational databases to Java Objects and vice versa, JPA permits the developer to work directly with objects rather than with SQL statements. JPA is a specification and several implementations are available.



JPA is not the first attempt to create an ORM solution in Java. Before JPA, there were Java Data Objects (JDO) andEnterprise JavaBeans (EJB). JDO used to be popular, but seems to have run out of steam.

EJB, up to version 2.1, was overly complex and hard to use, harder than losing weight. EJB 3.0 simplifies things a lot and even uses JPA as its persistence mechanism. In short, JPA has started as part of EJB 3.0. However, since people want to use JPA without an EJB container, JPA has become an independent specification.

JPA is merely a specification, i.e. a document. In order for it to be useful, it needs a reference implementation, which isa Java API that implements the specification. There are numerous software packages that are JPA reference implementations. Hibernate, EclipseLink, and Apache OpenJPA are some of them.

Below listed are few advantages of JPA:

1. You don't need to create tables. In some cases, you don't even need to create a database. If any of your entityclasses changes, the modern JPA provider can be configured to adapt the tables.
2. You don't need to write SQL statements, even though sometimes you may have to work with JPQL, the JavaPersistence Query Language.

3. Changing databases, say from Oracle to MySQL, is a breeze.

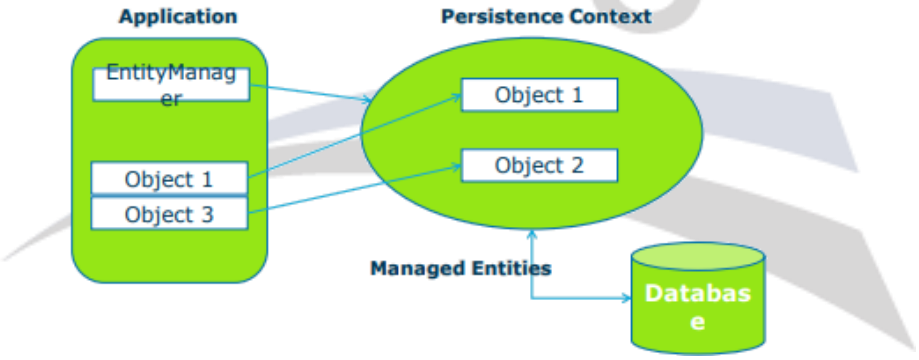
There are disadvantages too, but most of them are negligible:

1. JPA adds to the application's memory usage. Negligible in most cases.
2. JPA adds an extra layer to the application, making the system a bit slower than if it accesses the databasethrough JDBC directly. However, the performance penalty is small that it is considered negligible.

The Persistence Life Cycle

Before we start working with ORM, it is very important to understand how ORM works. The diagram shows an example of abstraction, when we dial or receive a call on mobile, lot of functionality goes in background. We as a user,least bothered about internal component working due to abstraction.
Similarly, objects created in your application, when passed to ORM, get stored in database table. How it happens? What work goes in background? How your object persisted in database?

Sample JPA Runtime:



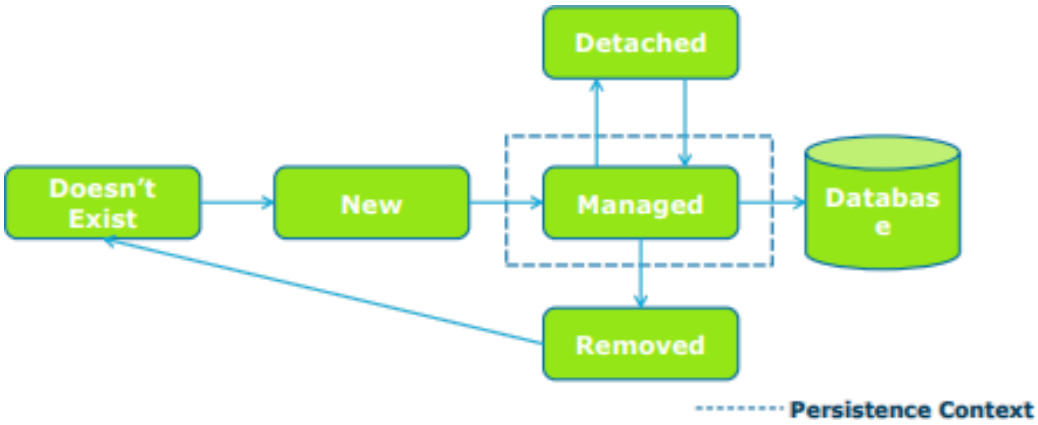
The diagram shows a sample JPA runtime

Entity Manager: The EntityManager is the primary interface used by application developers to interact with the JPAruntime.

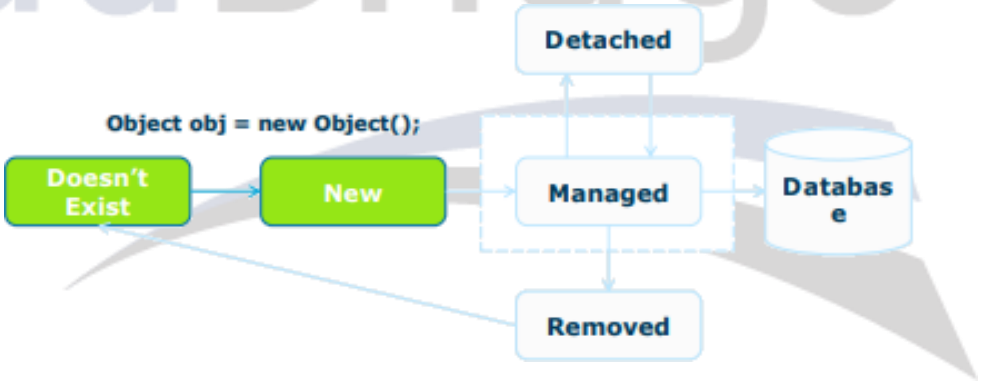
Persistence Context: Persistence context defines a scope under which particular entity instances are created,persisted, and removed.

Every EntityManager manages its own persistence context. In short, persistence context is a memory area for EntityManager to work on entity instance.

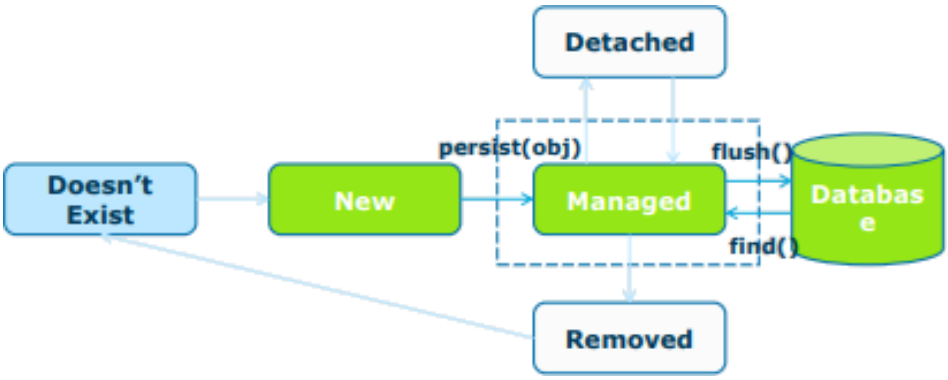
JPA uses EntityManager instance to manage objects which required to be persisted. Such objects are called Entities.Entities managed by EntityManager travels through different life cycle phases.



Object/Entity managed by ORM (using EntityManager) passes through different stages during its persistence.



New State: When an entity object is initially created its state is New. In this state the object is not yet associated with an Entity Manager and has no representation in the database.

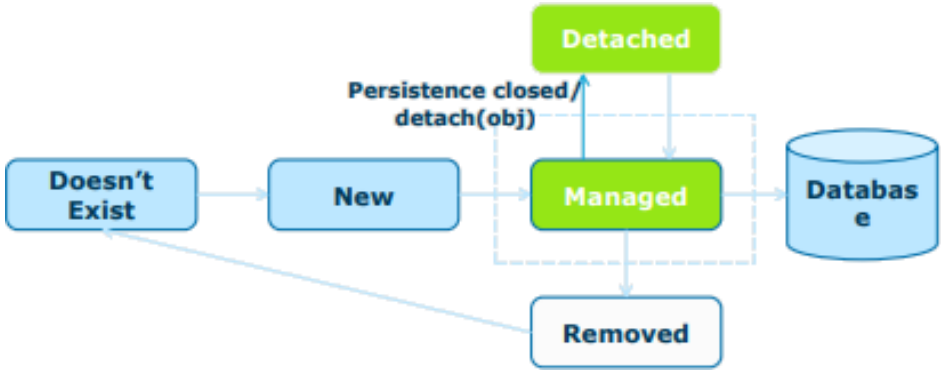


Managed State: An entity object becomes Managed when it is persisted to the database via an EntityManager'spersist method which must be invoked within an active transaction. On transaction commit, the owning Entity Manager stores the new entity object to the database.

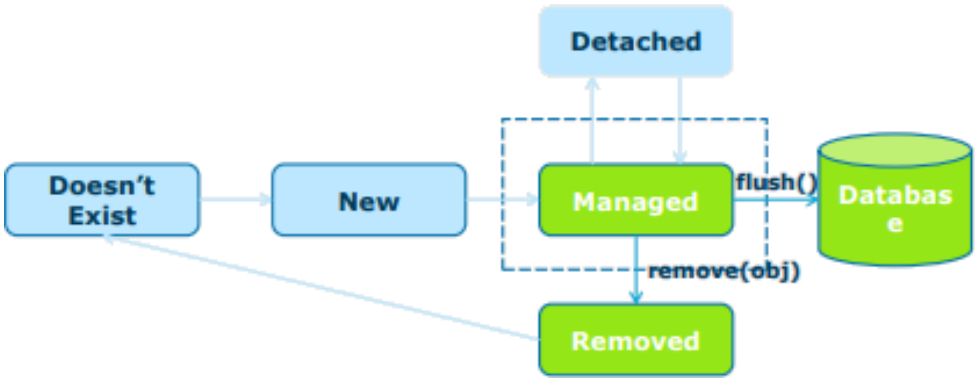
Entity objects retrieved from the database by an EntityManager are also in the Managed state.

If a managed entity object is modified within an active transaction the change is detected by the owning EntityManager and the update is propagated to the database on transaction commit.

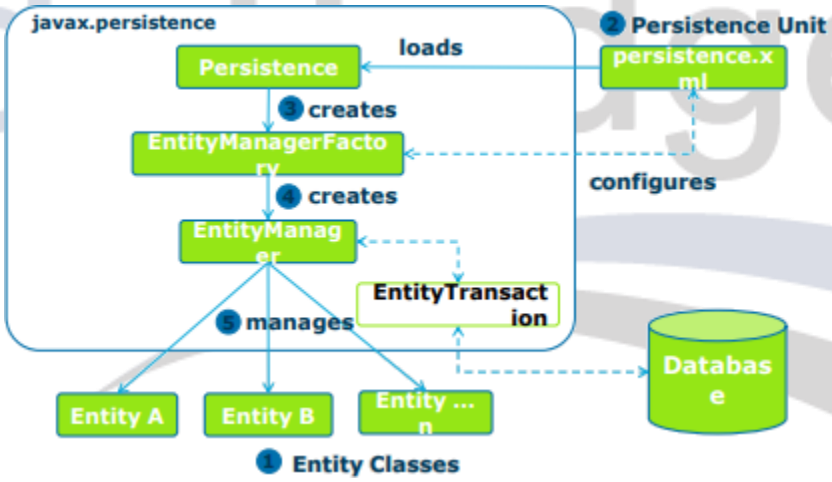
Detached State: represents entity objects that have been disconnected from the EntityManager. For instance, all themanaged objects of an EntityManager become detached when the EntityManager is closed.



Removed State: A managed entity object can also be retrieved from the database and marked for deletion, by using the EntityManager's remove method within an active transaction. The entity object changes its state from Managed toRemoved, and is physically deleted from the database during commit.



Working with JPA



Working with JPA

1. You normally start with a persistence strategy by identifying which classes need to be made entities.
2. Next step is to create configuration file (an XML document named persistence.xml) that contains the detailsabout the relational database.
3. EntityManagerFactory is a factory-based class responsible for creating EntityManager instance. It is obtainedusing Persistence class's createEntityManagerFactory static method.
4. EntityManagerFactory class designed to create EntityManager.
5. Once you have an EntityManager, you can start managing your entities. You can persist an entity, find one thatmatches a set of criteria, and so on. Each work of EntityManager with entities must be governed under EntityTransaction. Let us discuss each step in detail.

```
<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/ns/persistence/persistence_2_0.xsd"
version="2.0">
<persistence-unit name=“unit-name “>
<provider> <!-- JPA provider name like hibernate-->
</provider>
<properties> <!-- Database properties --></properties>
</persistence-unit>
</persistence>
```

Entities

An entity is a lightweight persistence domain object. Typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in that table. The primary programming artifact of an entity is the entityclass, although entities can use helper classes.

The persistent state of an entity is represented through either persistent fields or persistent properties. These fields orproperties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store.

Requirements for Entity Classes:

- The class must be annotated with the javax.persistence.Entity annotation.
- The class must have a public or protected, no-argument constructor.
- The class may have other constructors.
- The class must not be declared final. No methods or persistent instance variables must be declared final.
- Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
- Persistent instance variables must be declared private, protected, or packageprivate and can be accessed directly only by the entity class’s methods.

Entity Annotations:

The @Entity annotation marks this class as an entity bean, so it must have a no-argument constructor that is visiblewith at least protected scope.

Each entity bean has to have a primary key, which you annotate on the class with the @Id annotation.

In some situation, few properties of an entity, do not need to be stored in the database. In this case, ORM do not take this property for all the Database operation. This can be done using @Transient annotation.

By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to use—you can override this by also applying the @GeneratedValue annotation. This takes a pair of attributes: strategy and generator.

The strategy attribute must be a value from the GeneratorType enumeration, which defines four types of strategy constants.

1. **AUTO**: (Default) JPA decides which generator type to use, based on the database's support for primary key generation.
2. **IDENTITY**: The database is responsible for determining and assigning the next primary key.
3. **SEQUENCE**: Some databases support a SEQUENCE column type.
4. **TABLE**: This type keeps a separate table with the primary key values.

To connect with database, you need to set various properties regarding driver class, user name and password. Thisconfiguration is done with an XML file named persistence.xml.

Persistent Class

The entire concept of Hibernate is to take the values from Java class attributes and persist them to a database table. Amapping document helps Hibernate in determining how to pull the values from the classes and map them with table and associated fields.

Java classes whose objects or instances will be stored in database tables are called persistent classes in Hibernate. Hibernate works best if these classes follow some simple rules, also known as the Plain Old Java Object (POJO) programming model.

There are following main rules of persistent classes, however, none of these rules are hard requirements −

- All Java classes that will be persisted need a default constructor.
- All classes should contain an ID in order to allow easy identification of your objects within Hibernate and thedatabase. This property maps to the primary key column of a database table.
- All attributes that will be persisted should be declared private and have getXXX and setXXX methods defined inthe JavaBean style.
- A central feature of Hibernate, proxies, depends upon the persistent class being either non-final, or theimplementation of an interface that declares all public methods.
- All classes that do not extend or implement some specialized classes and interfaces required by the EJBframework.

The POJO name is used to emphasize that a given object is an ordinary Java Object, not a special object, and inparticular not an Enterprise JavaBean.

```
Public  class  Employee  {
  private int id;
  private  String  firstName;
  private  String  lastName;
  private int salary;

  public Employee() {}
  public Employee(String fname, String lname, int salary) {
    this.firstName = fname;
    this.lastName = lname;
    this.salary = salary;
  }

  public int getId() {
    return id;
  }
  public void setId( int id ) {
    this.id = id;
  }
  public String getFirstName() {
    return firstName;
  }
  public void setFirstName( String first_name ) {
    this.firstName = first_name;
  }
  public String getLastName() {
    return lastName;
  }
  public void setLastName( String last_name ) {
    this.lastName = last_name;
  }
  public int getSalary() {
    return salary;
  }
  public void setSalary( int salary ) {
    this.salary = salary;
  } }
```

A simple Persistent class should follow some rules:

- **A no-arg constructor**: It is recommended that you have a default constructor at least package visibility so that hibernate can create the instance of the Persistent class by newInstance() method.
- **Provide an identifier property**: It is better to assign an attribute as id. This attribute behaves as aprimary key in a database.
- **Declare getter and setter methods**: The Hibernate recognizes the method by getter and settermethod names by default.
- **Prefer non-final class**: Hibernate uses the concept of proxies, that depends on the persistent class.The application programmer will not be able to use proxies for lazy association fetching.

Persistent Fields and Properties in Entity Classes

The persistent state of an entity can be accessed through either the entity's instance variables or properties. The fieldsor properties must be of the following Java language types:

- Java primitive types
- java.lang.String
- Other serializable types, including:
  - Wrappers of Java primitive types
  - java.math.BigInteger
  - java.math.BigDecimal
  - java.util.Date
  - java.util.Calendar
  - java.sql.Date
  - java.sql.Time
  - java.sql.TimeStamp
  - User-defined serializable types
  - byte[]
  - Byte[]
  - char[]
  - Character[]
- Enumerated types
- Other entities and/or collections of entities
- Embeddable classes

Entities may use persistent fields, persistent properties, or a combination of both. If the mapping annotations are applied to the entity's instance variables, the entity uses persistent fields. If the mapping annotations are applied to the entity's getter methods for JavaBeans-style properties, the entity uses persistent properties.

Persistent Fields

If the entity class uses persistent fields, the Persistence runtime accesses entity-class instance variables directly. All fields not annotated javax.persistence.

Transient or not marked as Java transient will be persisted to the data store. The object/relational mappingannotations must be applied to the instance variables.

Persistent Properties

- If the entity uses persistent properties, the entity must follow the method conventions of JavaBeanscomponents.
- JavaBeans-style properties use getter and setter methods that are typically named after the entity class's instance variable names.
- For every persistent property, property of type, type of the entity, there is a getter method getProperty andsetter method setProperty.
- If the property is a Boolean, you may use isProperty instead of getProperty. For example, if a customer entity uses persistent properties and has a private instance variable called firstName, the class defines a getFirstName and setFirstName method for retrieving and setting the state of the firstName instance variable.
- The method signature for single-valued persistent properties are as follows:

```
Type getProperty()

void setProperty(Type type)
```

- The object/relational mapping annotations for persistent properties must be applied to the getter methods.Mapping annotations cannot be applied to fields or properties annotated @Transient or marked transient.

Using Collections in Entity Fields and Properties

Collection-valued persistent fields and properties must use the supported Java collection interfaces regardless of whether the entity uses persistent fields or properties. The following collection interfaces may be used:

- java.util.Collection
- java.util.Set
- java.util.List
- java.util.Map

If the entity class uses persistent fields, the type in the preceding method signatures must be one of these collection types. Generic variants of these collection types may also be used. For example, if it has a persistent property that contains a set of phone numbers, the Customer entity would have the following methods:

```
Set<PhoneNumber> getPhoneNumbers() { ... }
void setPhoneNumbers(Set<PhoneNumber>) { ... }
```

If a field or property of an entity consists of a collection of basic types or embeddable classes, usethe javax.persistence.ElementCollection annotation on the field or property.

The two attributes of @ElementCollection are targetClass and fetch. The targetClass attribute specifies the class nameof the basic or embeddable class and is optional if the field or property is defined using Java programming language generics. The optional fetch attribute is used to specify whether the collection should be retrieved lazily or eagerly, using the javax.persistence.FetchType constants of either LAZY or EAGER, respectively. By default, the collection will be fetched lazily.

The following entity, Person, has a persistent field, nicknames, which is a collection of String classes that will befetched eagerly. The targetClass element is not required, because it uses generics to define the field.

```
@Entity
public class Person {
  ...
  @ElementCollection(fetch=EAGER)
  protected Set<String> nickname = new HashSet();
  ...
}
```

Collections of entity elements and relationships may be represented by java.util.Map collections. A Map consists of akey and a value.

When using Map elements or relationships, the following rules apply.

- The Map key or value may be a basic Java programming language type, an embeddable class, or an entity.
- When the Map value is an embeddable class or basic type, use the @ElementCollection annotation.
- When the Map value is an entity, use the @OneToMany or @ManyToMany annotation.
- Use the Map type on only one side of a bidirectional relationship.

If the key type of a Map is a Java programming language basic type, use the annotation javax.persistence.MapKeyColumn to set the column mapping for the key. By default, the name attribute of @MapKeyColumn is of the form *RELATIONSHIP-FIELD/PROPERTY-NAME*_KEY. For example, if the referencingrelationship field name is image, the default name attribute is IMAGE_KEY.

If the key type of a Map is an entity, use the javax.persistence.MapKeyJoinColumn annotation. If the multiple columns are needed to set the mapping, use the annotation javax.persistence.MapKeyJoinColumns to include multiple @MapKeyJoinColumn annotations. If no @MapKeyJoinColumn is present, the mapping column name is by default set to *RELATIONSHIP-FIELD/PROPERTY-NAME*_KEY. For example, if the relationship field name is employee, thedefault name attribute is EMPLOYEE_KEY.

If Java programming language generic types are not used in the relationship field or property, the key class must be explicitly set using the javax.persistence.MapKeyClass annotation.

If the Map key is the primary key or a persistent field or property of the entity that is the Map value, use the javax.persistence.MapKey annotation. The @MapKeyClass and @MapKey annotations cannot be used on the same field or property.

If the Map value is a Java programming language basic type or an embeddable class, it will be mapped as a collectiontable in the underlying database. If generic types are not used, the @ElementCollection annotation's targetClass attribute must be set to the type of the Map value.

If the Map value is an entity and part of a many-to-many or one-to-many unidirectional relationship, it will be mapped as a join table in the underlying database. A unidirectional one-to-many relationship that uses a Map may also be mapped using the @JoinColumn annotation.

If the entity is part of a one-to-many/many-to-one bidirectional relationship, it will be mapped in the table of the entity that represents the value of the Map. If generic types are not used, the targetEntity attribute of the @OneToMany and @ManyToMany annotations must be set to the type of the Map value.

Validating Persistent Fields and Properties

The Java API for JavaBeans Validation (Bean Validation) provides a mechanism for validating application data. Bean Validation is integrated into the Java EE containers, allowing the same validation logic to be used in any of the tiers of an enterprise application.

Bean Validation constraints may be applied to persistent entity classes, embeddable classes, and mapped superclasses. By default, the Persistence provider will automatically perform validation on entities with persistent fields or properties annotated with Bean Validation constraints immediately after the PrePersist, PreUpdate, and PreRemove lifecycle events.

Bean Validation constraints are annotations applied to the fields or properties of Java programming language classes. Bean Validation provides a set of constraints as well as an API for defining custom constraints. Custom constraints can be specific combinations of the default constraints, or new constraints that don't use the default constraints. Each constraint is associated with at least one validator class that validates the value of the constrained field or property. Custom constraint developers must also provide a validator class for the constraint.

Bean Validation constraints are applied to the persistent fields or properties of persistent classes. When adding Bean Validation constraints, use the same access strategy as the persistent class. That is, if the persistent class uses field

access, apply the Bean Validation constraint annotations on the class's fields. If the class uses property access, apply the constraints on the getter methods.

Built in Bean Validation Constraints

Constraint	Description	Example
@AssertFalse	The value of the field or property must be false.	@AssertFalse boolean isUnsupported;
@AssertTrue	The value of the field or property must be true.	@AssertTrue boolean isActive;
@DecimalMax	The value of the field or property must be a decimal value lower than or equal to the number in the value element.	@DecimalMax("30.00") BigDecimal discount;
@DecimalMin	The value of the field or property must be a decimal value greater than or equal to the number in the value element.	@DecimalMin("5.00") BigDecimal discount;
@Digits	The value of the field or property must be a number within a specified range. The integer element specifies the maximum integral digits for the number, and the fraction element specifies the maximum fractional digits for the number.	@Digits(integer=6, fraction=2) BigDecimal price;
@Future	The value of the field or property must be a date in the future.	@Future Date eventDate;
@Max	The value of the field or property must be an integer value lower than or equal to the number in the value element.	@Max(10) int quantity;
@Min	The value of the field or property must be an integer value greater than or equal to the number in the value element.	@Min(5) int quantity;
@NotNull	The value of the field or property must not be null.	@NotNull String username;
@Null	The value of the field or property must be null.	@Null String unusedString;
@Past	The value of the field or property must be a date in the past.	@Past Date birthday;
@Pattern	The value of the field or property must match the regular expression defined in the regexp element.	@Pattern(regexp="\\(\\d{3}\\)\\d{3}-\\d{4}") String phoneNumber;
@Size	The size of the field or property is evaluated and must match the specified boundaries. If the field or property is a String, the size of the string is evaluated. If the field or property is a Collection, the size of the Collection is evaluated. If the field or property is a Map, the size of the Map is evaluated. If the field or property is an array, the size of the array is evaluated. Use one of the optional max or min elements to specify the boundaries.	@Size(min=2, max=240) String briefMessage;

The above table lists Bean Validation's built-in constraints, defined in the javax.validation.constraints package.

All the built-in constraints listed in above Table have a corresponding annotation, ConstraintName.List, for groupingmultiple constraints of the same type on the same field or property. For example, the following persistent field has two @Pattern constraints:

```
@Pattern.List({
   @Pattern(regexp="..."),
   @Pattern(regexp="...")
})
```

The following entity class, Contact, has Bean Validation constraints applied to its persistent fields.

```
@Entity
public class Contact implements Serializable {
private static final long serialVersionUID = 1L;
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
@NotNull
protected String firstName;
@NotNull
protected String lastName;
@Pattern(regexp="[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\."
      +"[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@"
      +"(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?",
          message="{invalid.email}")

protected String email;
@Pattern(regexp="^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
message="{invalid.phonenumber}")
protected String mobilePhone;
@Pattern(regexp="^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
     message="{invalid.phonenumber}")
protected String homePhone;
@Temporal(javax.persistence.TemporalType.DATE)
@Past
     protected Date birthday;
     ...
}
```

The @NotNull annotation on the firstName and lastName fields specifies that those fields are now required. If a new Contact instance is created where firstName or lastName have not been initialized, Bean Validation will throw a validation error. Similarly, if a previously created instance of Contact has been modified so that firstName or lastNameare null, a validation error will be thrown.

The email field has a @Pattern constraint applied to it, with a complicated regular expression that matches most valid email addresses. If the value of email doesn't match this regular expression, a validation error will be thrown.

The homePhone and mobilePhone fields have the same @Pattern constraints. The regular expression matches 10 digit telephone numbers in the United States and Canada of the form (xxx) xxx–xxxx.

The birthday field is annotated with the @Past constraint, which ensures that the value of birthday must be in the past.

Primary Keys in Entities

Each entity has a unique object identifier. A customer entity, for example, might be identified by a customer number.The unique identifier, or *primary key*, enables clients to locate a particular entity instance. Every entity must have a primary key. An entity may have either a simple or a composite primary key.

Simple primary keys use the javax.persistence.Id annotation to denote the primary key property or field.

Composite primary keys are used when a primary key consists of more than one attribute, which corresponds to a set of single persistent properties or fields. Composite primary keys must be defined in a primary key class. Composite primary keys are denoted using the javax.persistence.EmbeddedId and javax.persistence.IdClass annotations.

The primary key, or the property or field of a composite primary key, must be one of the following Java languagetypes:

- Java primitive types
- Java primitive wrapper types
- java.lang.String
- java.util.Date (the temporal type should be DATE)
- java.sql.Date
- java.math.BigDecimal
- java.math.BigInteger

Floating-point types should never be used in primary keys. If you use a generated primary key, only integral types willbe portable.

A primary key class must meet these requirements.

- The access control modifier of the class must be public.
- The properties of the primary key class must be public or protected if property-based access is used.
- The class must have a public default constructor.
- The class must implement the hashCode() and equals(Object other) methods.
- The class must be serializable.
- A composite primary key must be represented and mapped to multiple fields or properties of the entity classor must be represented and mapped as an embeddable class.
- If the class is mapped to multiple fields or properties of the entity class, the names and types of the primarykey fields or properties in the primary key class must match those of the entity class.

The following primary key class is a composite key, and the orderId and itemId fields together uniquely identify an entity:

```
public final class LineItemKey implements Serializable {
public Integer orderId;
public int itemId;

public LineItemKey() {}

public LineItemKey(Integer orderId, int itemId) {
this.orderId = orderId;
this.itemId = itemId;
    }
public boolean equals(Object otherOb) {
if (this == otherOb) {
        return true;
        }
if (!(otherOb instanceof LineItemKey)) {
return false;
        }
LineItemKey other = (LineItemKey) otherOb;
return (
          (orderId==null?other.orderId==null:orderId.equals
          (other.orderId)
                )
                &&
                (itemId == other.itemId)
             );
    }
    public int hashCode() {
       return (
                (orderId==null?0:orderId.hashCode())
                ^
                ((int) itemId)
             );
    }
    public String toString() {
       return "" + orderId + "-" + itemId;
    }
}
```

Managing Entities

Entities are managed by the entity manager, which is represented by javax.persistence.EntityManager instances. Each EntityManager instance is associated with a persistence context: a set of managed entity instances that exist in a particular data store. A persistence context defines the scope under which particular entity instances are created, persisted, and removed. The EntityManager interface defines the methods that are used to interact with the persistence context.

The EntityManager Interface

The EntityManager API creates and removes persistent entity instances, finds entities by the entity's primary key, and allows queries to be run on entities.

Container-Managed Entity Managers

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.

JTA transactions usually involve calls across application components. To complete a JTA transaction, these components usually need access to a single persistence context. This occurs when an EntityManager is injected into the application components by means of the javax.persistence.PersistenceContext annotation. The persistence context is automatically propagated with the current JTA transaction, and EntityManager references that are mappedto the same persistence unit provide access to the persistence context within that transaction. By automatically
propagating the persistence context, application components don't need to pass references to EntityManager instances to each other in order to make changes within a single transaction. The Java EE container manages thelifecycle of container-managed entity managers.

To obtain an EntityManager instance, inject the entity manager into the application component:

```
@PersistenceContext
EntityManager em;
```

Application-Managed Entity Managers

With an application-managed entity manager, on the other hand, the persistence context is not propagated toapplication components, and the lifecycle of EntityManager instances is managed by the application.

Application-managed entity managers are used when applications need to access a persistence context that is not propagated with the JTA transaction across EntityManager instances in a particular persistence unit. In this case,

eachEntityManager creates a new, isolated persistence context. The EntityManager and its associated persistence contextare created and destroyed explicitly by the application. They are also used when directly injecting EntityManager instances can't be done because EntityManager instances are not thread-safe. EntityManagerFactory instances are thread-safe.

Applications create EntityManager instances in this case by using the createEntityManager method of javax.persistence.EntityManagerFactory.

To obtain an EntityManager instance, you first must obtain an EntityManagerFactory instance by injecting it into the application component by means of the javax.persistence.PersistenceUnit annotation:

```
@PersistenceUnit
EntityManagerFactory emf;
```

Then obtain an EntityManager from the EntityManagerFactory instance:

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

Application-managed entity managers don't automatically propagate the JTA transaction context. Such applications need to manually gain access to the JTA transaction manager and add transaction demarcation information when performing entity operations. The javax.transaction.UserTransaction interface defines methods to begin, commit, androll back transactions. Inject an instance of UserTransaction by creating an instance variable annotated with @Resource:

```
@Resource
UserTransaction utx;
```

To begin a transaction, call the UserTransaction.begin method. When all the entity operations are complete, call the UserTransaction.commit method to commit the transaction. The UserTransaction.rollback method is used to roll back the current transaction.
The following example shows how to manage transactions in an application that uses an application-managed entity manager:

```
@PersistenceContext
EntityManagerFactory emf;
EntityManager em;
@Resource
UserTransaction utx;
...
em = emf.createEntityManager();
try {
  utx.begin();
  em.persist(SomeEntity);
  em.merge(AnotherEntity);
  em.remove(ThirdEntity);
  utx.commit();
} catch (Exception e) {
  utx.rollback();
}
```


Finding Entities Using the EntityManager

The EntityManager.find method is used to look up entities in the data store by the entity's primary key:

```
@PersistenceContext
EntityManager em;
public void enterOrder(int custID, Order newOrder) {
   Customer cust = em.find(Customer.class, custID);
   cust.getOrders().add(newOrder);
   newOrder.setCustomer(cust);
}
```

Managing an Entity Instance's Lifecycle

You manage entity instances by invoking operations on the entity by means of an EntityManager instance. Entityinstances are in one of four states: new, managed, detached, or removed.

- New entity instances have no persistent identity and are not yet associated with a persistence context.
- Managed entity instances have a persistent identity and are associated with a persistence context.
- Detached entity instances have a persistent identity and are not currently associated with a persistencecontext.
- Removed entity instances have a persistent identity, are associated with a persistent context, and arescheduled for removal from the data store.

Persisting Entity Instances

New entity instances become managed and persistent either by invoking the persist method or by a cascading persist operation invoked from related entities that have the cascade=PERSIST or cascade=ALL elements set in the relationship annotation. This means that the entity's data is stored to the database when the transaction associated with the persist operation is completed. If the entity is already managed, the persist operation is ignored, although the persist operation will cascade to related entities that have the cascade element set to PERSIST or ALL in the relationship annotation. If persist is called on a removed entity instance, the entity becomes managed. If the entityis detached, either persist will throw an IllegalArgumentException, or the transaction commit will fail.

```
@PersistenceContext
EntityManager em;
...
public LineItem createLineItem(Order order, Product product,
      int quantity) {
   LineItem li = new LineItem(order, product, quantity);
   order.getLineItems().add(li);
   em.persist(li);
   return li;
}
```

The persist operation is propagated to all entities related to the calling entity that have the cascade element setto ALL or PERSIST in the relationship annotation:

```
@OneToMany(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
   return lineItems;
}
```

Removing Entity Instances

Managed entity instances are removed by invoking the remove method or by a cascading remove operation invoked from related entities that have the cascade=REMOVE or cascade=ALL elements set in the relationship annotation. If the remove method is invoked on a new entity, the remove operation is ignored, although remove will cascade to related entities that have the cascade element set to REMOVE or ALL in the relationship annotation. If remove is invoked on a detached entity, either remove will throw an IllegalArgumentException, or the transaction commit will fail. If invoked on an already removed entity, remove will be ignored. The entity's data will be removed from the data store when the transaction is completed or as a result of the flush operation.

```
public void removeOrder(Integer orderId) {
   try {
      Order order = em.find(Order.class, orderId);
      em.remove(order);
   }...
```

In this example, all LineItem entities associated with the order are also removed, as Order.getLineItems

hascascade=ALL set in the relationship annotation.

Synchronizing Entity Data to the Database

The state of persistent entities is synchronized to the database when the transaction with which the entity is associated commits. If a managed entity is in a bidirectional relationship with another managed entity, the data will bepersisted, based on the owning side of the relationship.

To force synchronization of the managed entity to the data store, invoke the flush method of the EntityManager instance. If the entity is related to another entity and the relationship annotation has the cascade element set to PERSIST or ALL, the related entity's data will be synchronized with the data storewhen flush is called.

If the entity is removed, calling flush will remove the entity data from the data store.

Persistence Units

A persistence unit defines a set of all entity classes that are managed by EntityManager instances in an application.This set of entity classes represents the data contained within a single data store.

Persistence units are defined by the persistence.xml configuration file. The following is anexample persistence.xml file:

```
<persistence>
  <persistence-unit name="OrderManagement">
    <description>This unit manages orders and customers.
      It does not rely on any vendor-specific features and can
      therefore be deployed to any persistence provider.
    </description>
    <jta-data-source>jdbc/MyOrderDB</jta-data-source>
    <jar-file>MyOrderApp.jar</jar-file>
    <class>com.widgets.Order</class>
    <class>com.widgets.Customer</class>
  </persistence-unit>
</persistence>
```

This file defines a persistence unit named OrderManagement, which uses a JTA-aware data source: jdbc/MyOrderDB.The jar-file and class elements specify managed persistence classes: entity classes, embeddable classes, and mapped superclasses. The jar-file element specifies JAR files that are visible to the packaged persistence unit that contain managed persistence classes, whereas the class element explicitly names managed persistence classes.

The jta-data-source (for JTA-aware data sources) and non-jta-data-source (for non-JTA-aware data sources) elementsspecify the global JNDI name of the data source to be used by the container.

The JAR file or directory whose META-INF directory contains persistence.xml is called the root of the persistence unit. The scope of the persistence unit is determined by the persistence unit's root. Each persistence unit must be identifiedwith a name that is unique to the persistence unit's scope.

Persistent units can be packaged as part of a WAR or EJB JAR file or can be packaged as a JAR file that can then be included in a WAR or EAR file.

- If you package the persistent unit as a set of classes in an EJB JAR file, persistence.xml should be put in the EJB
  JAR's META-INF directory.
- If you package the persistence unit as a set of classes in a WAR file, persistence.xml should be located in the
  WAR file's WEB-INF/classes/META-INF directory.
- If you package the persistence unit in a JAR file that will be included in a WAR or EAR file, the JAR file shouldbe located in either
  - The WEB-INF/lib directory of a WAR
  - The EAR file's library directory

JPA Queries

Java Persistence Query Language (JPQL)

The Java Persistence Query Language (JPQL) is a platform-independent object-oriented query language defined as part of the Java Persistence API (JPA) specification.

JPQL is used to make queries against entities stored in a relational database. The JPQL defines queries for entities and their persistent state. The query language allows you to write portable queries that work regardless of the underlying data store.

The JPQL can be considered as an object-oriented version of SQL. Users familiar with SQL should find JPQL very easy to learn and use.

The main difference between SQL and JPQL is that SQL works with relational database tables, records and fields, whereas JPQL works with Java classes and objects.

Similarities between SQL and JPQL

As shown above, there is no difference between SQL and JPQL query syntax. Consider the following Entity class,

```
@Entity
public class Book implements Serializable {
@Id
private Long id;
private String bookTitle;
private String author;
private Double price; // getter and setter methods
}
```

If you want to find all books written by author 'Jim Kathy', then you need to write JPQL select statement on aboveentity class as given below:

```
SELECT b.id,b. bookTitle,b.price --property reference
FROM Book b --object reference
WHERE b.author = 'Jim Kathy';
```

Whereas the below query counts total books object available in data store.

```
SELECT COUNT(b.id)
FROM Book b;
```

Queries are represented in JPA 2 by two interfaces - the old Query interface, which was the only interface available forrepresenting queries in JPA 1, and the new TypedQuery<T> JPA interface that was introduced in JPA 2.

The TypedQuery interface extends the Query interface.

It is easier to run queries and process the query results in a type safe manner when using the TypedQuery

interface.The Query/TypedQuery<T> interface defines two methods for running SELECT queries:

1. getSingleResult() - for use when exactly one result object is expected.
2. getResultList() - for general use in any other case.

For UPDATE and DELETE use executeUpdate() method.

Query interface should be used mainly when the query result type is unknown or when a query returns polymorphic results and the lowest known common denominator of all the result objects is Object.

When a more specific result type is expected queries should usually use the TypedQuery.

Query parameters enable the definition of reusable queries. Such queries can be executed with different parametervalues to retrieve different results.

There are multiple ways to pass parameters to query.

1. Named Parameters (:name)
2. Ordinal Parameters (?index)
3. Criteria Query Parameter

Named Queries

The hibernate named query is way to use any query by some meaningful name. It is like using alias names. The Hibernate framework provides the concept of named queries so that application programmer need not to scatter queries to all the java code.

There are two ways to define the named query in hibernate:

- by annotation
- by mapping file.

Hibernate Named Query by annotation

If you want to use named query in hibernate, you need to have knowledge of @NamedQueries and @NamedQuery annotations.

@NameQueries annotation is used to define the multiple named queries.

@NameQuery annotation is used to define the single named query.

Let's see the example of using the named queries:

```
@NamedQueries(
  {
    @NamedQuery(
    name = "findEmployeeByName",
    query = "from Employee e where e.name = :name"
    )
  }
)
```

Example of Hibernate Named Query by annotation

In this example, we are using annotations to defined the named query in the persistent class. There are three files only:

- Employee.java
- hibernate.cfg.xml
- FetchDemo

In this example, we are assuming that there is em table in the database containing 4 columns id, name, job and salary

and there are some records in this table.

Employee.java

It is a persistent class that uses annotations to define named query and marks this class as entity.

```
package com.javatpoint;

import javax.persistence.*;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@NamedQueries(
  {
    @NamedQuery(
    name = "findEmployeeByName",
    query = "from Employee e where e.name = :name"
    )
  }
)

@Entity
@Table(name="em")
public class Employee {

  public String toString(){return id+" "+name+" "+salary+" "+job;}

  int id;
  String name;
  int  salary;
  String job;
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)

  //getters and setters
}
```

hibernate.cfg.xml

It is a configuration file that stores the informations about database such as driver class, url, username, password and mapping class etc.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">

<hibernate-configuration>

   <session-factory>
      <property name="hbm2ddl.auto">update</property>
      <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
      <property  name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
      <property name="connection.username">system</property>
      <property name="connection.password">jtp</property>
      <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
   <mapping class="com.javatpoint.Employee"/>
   </session-factory>

</hibernate-configuration>
```

FetchData.java

It is a java class that uses the named query and prints the informations based on the query. The getNamedQuerymethod uses the named query and returns the instance of Query.

```
package com.javatpoint;

import java.util.*;
import javax.persistence.*;
import org.hibernate.*;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import  org.hibernate.boot.registry.StandardServiceRegistry;
import  org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class Fetch {
public static void main(String[] args) {

    StandardServiceRegistry ssr=new StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
      Metadata meta=new MetadataSources(ssr).getMetadataBuilder().build();

      SessionFactory factory=meta.getSessionFactoryBuilder().build();
      Session session=factory.openSession();

   //Hibernate Named Query
         TypedQuery query = session.getNamedQuery("findEmployeeByName");
          query.setParameter("name","amit");

          List<Employee> employees=query.getResultList();

   Iterator<Employee> itr=employees.iterator();
    while(itr.hasNext()){
   Employee e=itr.next();
   System.out.println(e);
    }
   session.close();
 }
}
```

Hibernate Named Query by mapping file

If want to define named query by mapping file, you need to use query element of hibernate-mapping to define the named query.

In such case, you need to create hbm file that defines the named query. Other resources are same as given in the above example except Persistent class Employee.java where you don't need to use any annotation and hibernate.cfg.xml file where you need to specify mapping resource of the hbm file.

The hbm file should be like this:

Emp.hbm.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">

<hibernate-mapping>
<class name="com.javatpoint.Employee" table="em">
<id name="id">
<generator class="native"></generator>
</id>
<property name="name"></property>
<property name="job"></property>
<property name="salary"></property>
</class>

<query name="findEmployeeByName">
<![CDATA[from Employee e where e.name = :name]]>
</query>

</hibernate-mapping>
```

The persistent class should be like this:
Employee.java

```
package com.javatpoint;
public class Employee {
   int id;
   String name;
   int salary;
   String job;
   //getters and setters
}
```

Now include the mapping resource in the hbm file as:

hibernate.cfg.xml

```
<mapping resource="emp.hbm.xml"/>
```

JPQL allows us to create both static as well as dynamic queries. Now, we will perform some basic JPQL operations using both type of queries on the below table.

S_ID	S_NAME	S_AGE
101	Gaurav	24
102	Rahul	22
103	Chris	20
104	Ronit	26
105	Roy	21

JPQL Dynamic Query Example

In this example, we will fetch single column from database by using createQuery() method .
StudentEntity.java

```
package com.javatpoint.jpa;
import javax.persistence.*;

  @Entity
  @Table(name="student")
  public class StudentEntity {

    @Id
    private int s_id;
    private String s_name;
    private int s_age;

    public StudentEntity(int s_id, String s_name, int s_age) {
      super();
      this.s_id = s_id;
      this.s_name = s_name;
      this.s_age = s_age;
    }
    public StudentEntity() {
      super();
    }
    public int getS_id() {
      return s_id;
    }
    public void setS_id(int s_id) {
      this.s_id = s_id;
    }
    public String getS_name() {
      return s_name;
    }
    public void setS_name(String s_name) {
      this.s_name = s_name;
    }
    public int getS_age() {
      return s_age;
    }
    public void setS_age(int s_age) {
      this.s_age = s_age;
    }
  }
```

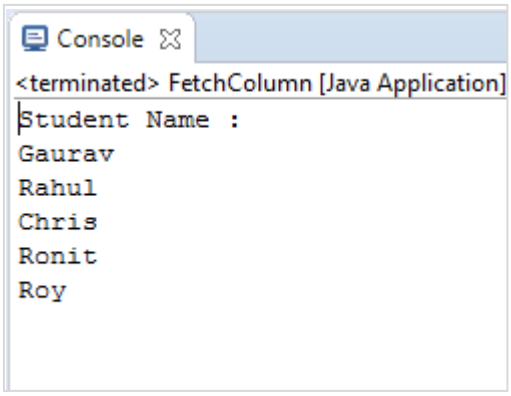
Persistence.xml

```
<persistence>
<persistence-unit name="Student_details">

    <class>com.javatpoint.jpa.StudentEntity</class>

    <properties>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/studentdata"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.password" value=""/>
      <property name="eclipselink.logging.level" value="SEVERE"/>
      <property name="eclipselink.ddl-generation" value="create-or-extend-tables"/>
    </properties>

  </persistence-unit>
</persistence>
```

FetchColumn.java

```
package com.javatpoint.jpa.jpql;
import javax.persistence.*;
import java.util.*;
public class FetchColumn {

    public static void main( String args[]) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory( "Student_details" );
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin( );

        Query query = em.createQuery("Select s.s_name from StudentEntity s");
        @SuppressWarnings("unchecked")
      List<String> list =query.getResultList();
        System.out.println("Student Name :");
        for(String s:list) {

          System.out.println(s);
        }
        em.close();
        emf.close();
    }
}
```

Output:



JPQL Static Query Example

In this example, we will fetch single column from database by using createNamedQuery() method .

StudentEntity.java

```
package com.javatpoint.jpa;
import javax.persistence.*;

@Entity
@Table(name="student")
@NamedQuery(name = "find name" , query = "Select s from StudentEntity s")
public class StudentEntity {

@Id
private int s_id;
private String s_name;
private int s_age;

public StudentEntity(int s_id, String s_name, int s_age) {
super();
this.s_id = s_id;
this.s_name = s_name;
this.s_age = s_age;
}
public StudentEntity() {
super();
}
public int getS_id() {
return s_id;
}
public void setS_id(int s_id)
{this.s_id = s_id;
}
public String getS_name() {
return s_name;
}
public void setS_name(String s_name) {
this.s_name = s_name;
}
public int getS_age() {
return s_age;
}
public void setS_age(int s_age) {
this.s_age = s_age;
        }
    }
```

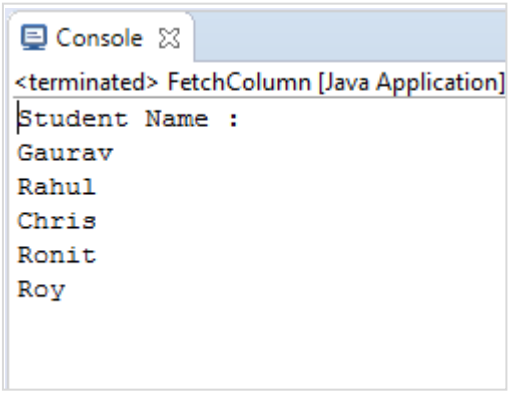
Persistence.xml

```
<persistence>
<persistence-unit name="Student_details">

    <class>com.javatpoint.jpa.StudentEntity</class>

    <properties>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/studentdata"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.password" value=""/>
      <property name="eclipselink.logging.level" value="SEVERE"/>
      <property name="eclipselink.ddl-generation" value="create-or-extend-tables"/>
    </properties>

  </persistence-unit>
</persistence>
```

FetchColumn.java

```
package com.javatpoint.jpa.jpql;
import javax.persistence.*;
import java.util.*;
public class FetchColumn {

    public static void main( String args[]) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory( "Student_details" );
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin( );

        Query query = em.createQuery("Select s.s_name from StudentEntity s");
        @SuppressWarnings("unchecked")
       List<String> list =query.getResultList();
        System.out.println("Student Name :");
        for(String s:list) {

          System.out.println(s);
        }
        em.close();
        emf.close();
    }
}
```

Output:



JPQL Static Query Example

In this example, we will fetch single column from database by using createNamedQuery() method .
StudentEntity.java

```
package com.javatpoint.jpa;
    import javax.persistence.*;

@Entity
@Table(name="student")
@NamedQuery(name = "find name" , query = "Select s from StudentEntity s")
public class StudentEntity {

        @Id
        private int s_id;
        private String s_name;
        private int s_age;

public StudentEntity(int s_id, String s_name, int s_age) {
super();
this.s_id = s_id;
this.s_name = s_name;
this.s_age = s_age;
        }
public StudentEntity() {
super();
        }
public int getS_id() {
return s_id;
        }
public void setS_id(int s_id) {
this.s_id = s_id;
        }
public String getS_name() {
return s_name;
        }
public void setS_name(String s_name) {
this.s_name = s_name;
        }
public int getS_age() {
return s_age;
        }
public void setS_age(int s_age) {
this.s_age = s_age;
        }
    }
```

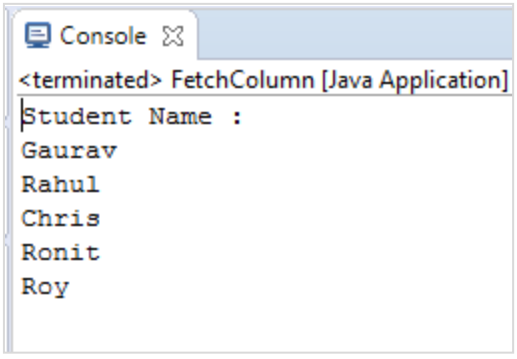
Persistence.xml

```
<persistence>
<persistence-unit name="Student_details">

    <class>com.javatpoint.jpa.StudentEntity</class>

    <properties>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/studentdata"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.password" value=""/>
      <property name="eclipselink.logging.level" value="SEVERE"/>
      <property name="eclipselink.ddl-generation" value="create-or-extend-tables"/>
    </properties>

  </persistence-unit>
</persistence>
```

FetchColumn.java

```
package com.javatpoint.jpa.jpql;
import javax.persistence.*;

import com.javatpoint.jpa.StudentEntity;

import java.util.*;
public class FetchColumn {

    public static void main( String args[]) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory( "Student_details" );
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin( );

        Query query = em.createNamedQuery("find name");
      @SuppressWarnings("unchecked")
      List<StudentEntity> list =query.getResultList();
      System.out.println("Student Name :");
      for(StudentEntity s:list) {

          System.out.println(s.getS_name());
        }
        em.close();
        emf.close();
    }
}
```

Output:



JPA Criteria API

The Criteria API is one of the most common ways of constructing queries for entities and their persistent state. It isjust an alternative method for defining JPA queries.

Criteria API defines a platform-independent criteria queries, written in Java programming language. It was introducedin JPA 2.0. The main purpose behind this is to provide a type-safe way to express a query.

Steps to create Criteria Query

To create a Criteria query, follow the below steps: -

- Create an object of CriteriaBuilder interface by invoking getCriteriaBuilder() method on the instance ofEntityManager interface.

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

    CriteriaBuilder cb=em.getCriteriaBuilder();
```

- Now, build an instance of CriteriaQuery interface to create a query object.

```
CriteriaQuery<StudentEntity> cq=cb.createQuery(StudentEntity.class);
```

- Call from method on CriteriaQuery object to set the query root.

```
Root<StudentEntity> stud=cq.from(StudentEntity.class);
```

- Now, call the select method of CriteriaQuery Object to specify type of query result.

```
CriteriaQuery<StudentEntity> select = cq.select(stud);
```

- Create an instance of Query interface and specify the type of method used to access the database records

```
Query q = em.createQuery(select);
```

- Now, control the execution of query by calling the methods of Query Interface.

```
List<StudentEntity> list = q.getResultList();
```

Methods of Criteria API Query Clauses

Following is the list of clauses with the corresponding interface and methods.

Clause	Criteria API Interface	Methods
SELECT	CriteriaQuery	select()
FROM	AbstractQuery	from()
WHERE	AbstractQuery	where()
ORDER BY	CriteriaQuery	orderBy()
GROUP BY	AbstractQuery	groupBy()
HAVING	AbstractQuery	having()

JPA Criteria SELECT Clause

The SELECT clause is used to fetch the data from database. The data can be retrieved in the form of single expressionor multiple expressions. In Criteria API, each form is expressed differently.

Criteria SELECT Example

Generally, select() method is used for the SELECT clause to fetch all type of forms. Here, we will perform several SELECT operations on student table. Let us assume the table contains the following records:

S_ID	S_NAME	S_AGE
101	Gaurav	24
102	Rahul	22
103	Chris	20
104	Ronit	26
105	Roy	21

Now, follow the below steps to perform operations: -

- Create an entity class. Here, we created StudentEntity.java under com.javatpoint.jpa package. This classcontains three attributes s_id, s_name, s_age with all the required annotations.

StudentEntity.java

```
package com.javatpoint.jpa;
    import javax.persistence.*;

    @Entity
    @Table(name="student")
    public class StudentEntity {

      @Id
      private int s_id;
      private String s_name;
      private int s_age;

      public StudentEntity(int s_id, String s_name, int s_age) {
        super();
        this.s_id =s_id;
        this.s_name = s_name;
        this.s_age = s_age;
      }
       public StudentEntity() {
        super();
      }
      public int getS_id() {
        return s_id;
      }
      public void setS_id(int s_id) {
        this.s_id = s_id;
      }
      public String getS_name() {
        return s_name;
      }
      public void setS_name(String s_name) {
        this.s_name = s_name;
      }
      public int getS_age() {
        return s_age;
      }
      public void setS_age(int s_age) {
        this.s_age = s_age;
      }
    }
```

Now, map the entity class and other databases confiuguration in Persistence.xml file.

Persistence.xml

```
<persistence>
<persistence-unit name="Student_details">

    <class>com.javatpoint.jpa.StudentEntity</class>

    <properties>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/studentdata"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.password" value=""/>
      <property name="eclipselink.logging.level" value="SEVERE"/>
      <property name="eclipselink.ddl-generation" value="create-or-extend-tables"/>
    </properties>

  </persistence-unit>
</persistence>
```

Once, we have created the basic entity class and mapped the configuration into persistence.xml file, we can perform the different types of select operations in the following ways: -

- **Selecting Single Expression**

Here, we will fetch single column from database with the help of a simple example.

SingleFetch.java

```
package com.javatpoint.jpa.jpql;
import com.javatpoint.jpa.StudentEntity;
import javax.persistence.*;
import javax.persistence.criteria.*;

import java.util.*;
public class SingleFetch {


   public static void main( String args[]) {
      EntityManagerFactory emf = Persistence.createEntityManagerFactory( "Student_details" );
       EntityManager em = emf.createEntityManager();
       em.getTransaction().begin( );
       CriteriaBuilder cb=em.getCriteriaBuilder();
       CriteriaQuery<StudentEntity> cq=cb.createQuery(StudentEntity.class);

      Root<StudentEntity> stud=cq.from(StudentEntity.class);

      cq.select(stud.get("s_name"));

       CriteriaQuery<StudentEntity> select = cq.select(stud);
       TypedQuery<StudentEntity> q = em.createQuery(select);
       List<StudentEntity> list = q.getResultList();

       System.out.println("s_id");

       for(StudentEntity s:list)
       {
       System.out.println(s.getS_id());

     }
em.getTransaction().commit();
       em.close();
       emf.close();
   }
}
```

Output:

```
s_id
101
102
103
104
105
```

- **Selecting Multiple Expression**

Here, we will fetch multiple columns from database with the help of a simple example.

MultiFetch.java

```
package com.javatpoint.jpa.jpql;
importcom.javatpoint.jpa.StudentEntity;
import javax.persistence.*;
import javax.persistence.criteria.*;

  import  java.util.*;
  public  classMultiFetch {

public static void main( String args[]) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory( "Student_details" );
         EntityManager em = emf.createEntityManager();
         em.getTransaction().begin( );
         CriteriaBuilder cb=em.getCriteriaBuilder();
         CriteriaQuery<StudentEntity> cq=cb.createQuery(StudentEntity.class);

        Root<StudentEntity> stud=cq.from(StudentEntity.class);

            cq.multiselect(stud.get("s_id"),stud.get("s_name"),stud.get("s_age") );
         CriteriaQuery<StudentEntity> select = cq.select(stud);
         TypedQuery<StudentEntity> q = em.createQuery(select);
         List<StudentEntity> list = q.getResultList();

         System.out.print("s_id");
           System.out.print("\t s_name");
           System.out.println("\t s_age");

         for(StudentEntity s:list)
         {
         System.out.print(s.getS_id());
         System.out.print("\t"+s.getS_name());
         System.out.println("\t"+s.getS_age());
         }
  em.getTransaction().commit();
         em.close();
         emf.close();
     }
}
```

Output:

```
s_id      s_name   s_age
101       Gaurav   24
102       Rahul    22
103       Chris    20
104       Ronit    26
105       Roy      21
```

Activity 1

Let us assume the table contains the following records: -

S_ID	S_NAME	S_AGE
101	Gaurav	24
102	Rahul	22
103	Chris	20
104	Ronit	26
105	Roy	21

1. Create an entity class. Here, we created StudentEntity.java under com.javatpoint.jpa package. This class contains three attributes s_id, s_name, s_age with all the required annotations.
2. Create an entity class. Here, we created StudentEntity.java under com.javatpoint.jpa package. This class contains three attributes s_id, s_name, s_age with all the required annotations.
3. Sort the table in Ascending order to get the following result:

```
s_id      s_name   s_age
103       Chris    20
105       Roy      21
102       Rahul    22
101       Gaurav   24
104       Ronit    26
```

(Note: Use JPA Orderby clause)

Activity 2

Let us assume the table contains the following records: -

S_ID	S_NAME	S_AGE
101	Gaurav	24
102	Rahul	22
103	Chris	20
104	Ronit	26
105	Roy	21

1. Create an entity class. Here, we created StudentEntity.java under com.javatpoint.jpa package. This class contains three attributes s_id, s_name, s_age with all the required annotations.
2. Create an entity class. Here, we created StudentEntity.java under com.javatpoint.jpa package. This class contains three attributes s_id, s_name, s_age with all the required annotations.
3. Display the records of students having age greater than 22

```
Students having age greater than 22
s_id     s_name  s_age
101      Gaurav  24
104      Ronit   26
```

(Note: Use JPQL Greater Than and Less Than)

Entity Relationships

What is Entity Association?

Association represents relationship between entities. A Java class can contain an object of another class or a set of objects of another class.

There is no directionality involved in relational world, its just a matter of writing a query. But there is notion ofdirectionality which is possible in java.

Hence associations are classified as

- Unidirectional

▪ Bidirectional

Unidirectional Relationships

- **One To One relationship**

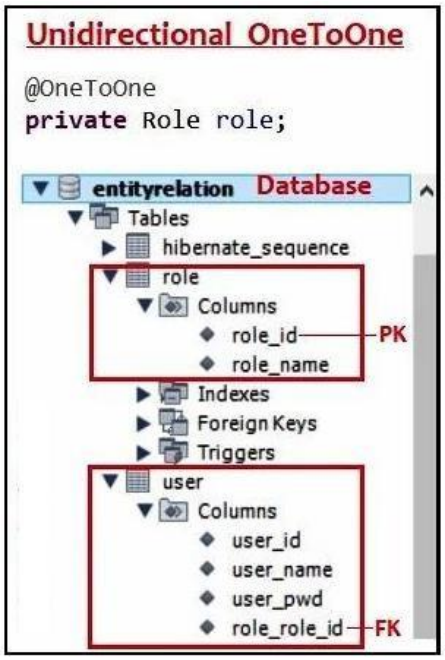
Defines a single-valued association to another entity that has one-to-one multiplicity. These annotations can havefollowing optional attributes:

1. **cascade (Optional):** The operations that must be cascaded to the target of the association. i.e. It indicates JPAoperations on associated entity along with owner of association.

2. **fetch (Optional)**: Whether the association should be lazily loaded or must be eagerly fetched. i.e. When you fetchStudent entity, if you want to load the associated entity (Address) immediately, then you have to mention this
attribute with 'EAGER'. Default is LAZY, means the associated entity (Address) will be loaded when required.

Example:

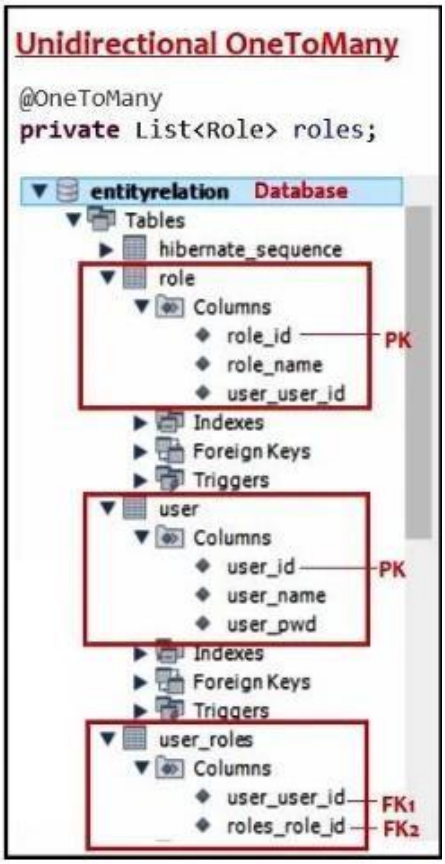
We can obtain a unidirectional relationship between User & Role entity by applying @OneToOne on the relational field at any side. For example, if we want to have a one to one relationship from User to Role, we need to add a fieldwith type Role in the User entity. It means one user will have one role. Hence, we need to apply @OneToOne on thefield with a type Role in the User entity.



```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity
public class User {
   @Id
   @GeneratedValue
   private Integer userId;
   private String userName;
   private String userPwd;

   @OneToOne
   private Role role;
}
```

- **One to Many Relationship**

We can obtain a unidirectional relationship between User & Role entity by applying @OneToMany on the relational field at any side. For example, if we want to have a one to many relationship from User to Role, we need to add a fieldwith type List<Role> in the User entity. It means one user will have many roles. Hence, we need to apply @OneToMany on the field with a type List<Role> in the User entity. For example, below code demonstrates the concept.

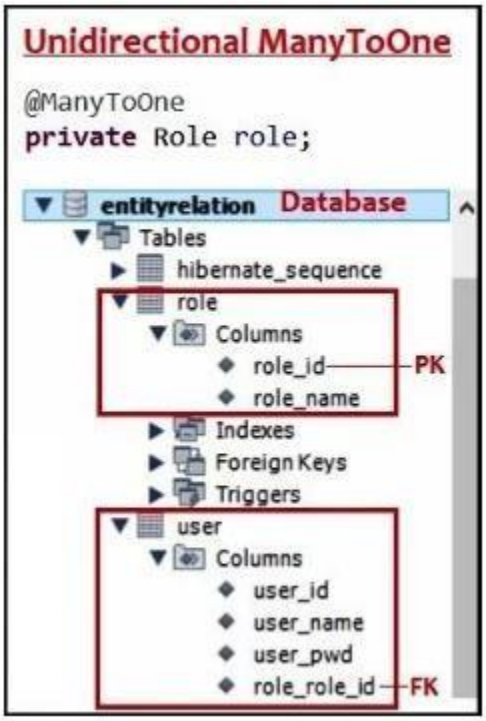



```
import java.util.List;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;

@Entity
public class User {

   @Id
   @GeneratedValue
   private Integer userId;
   private String userName;
   private String userPwd;

   @OneToMany
   private List<Role> roles;
}
```

- **Many to one Relationship**

Use-case: Let's assume that we have to maintain a relation between User & Role table. In order to satisfy Many to One relationship between the User and the Role table, multiple Users will have only one Role.

We can obtain a unidirectional relationship between User & Role entity by applying @ManyToOne on the relational field at any side. For example, if we want to have a many to one relationship from User to Role, we need to add a fieldwith type Role in the User entity. It means many users will have one role. Hence, we need to apply @ManyToOne on the field with a type Role in the User entity. For example, below code demonstrates the concept.

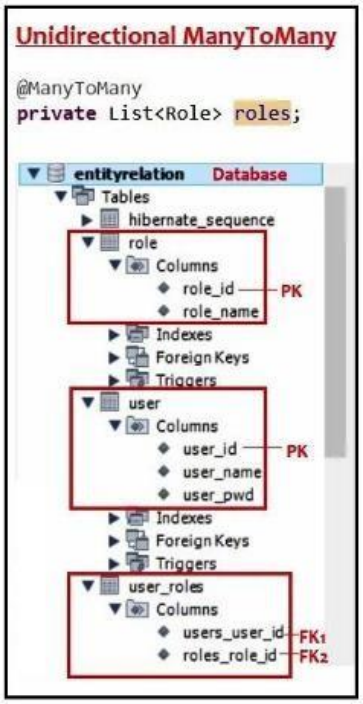


```
import javax.persistence.Entity;
Import Java.persistence.GeneratedValue;
import javax.persistence.Id;
Import Java.persistence.ManyToOne;

@Entity
public class User {

   @Id
   @GeneratedValue
   private Integer userId;

   private String userName;
   private String userPwd;

   @ManyToOne
   private Role role;
}
```

- **Many to many relationship**

Use-case: Let's assume that we have to maintain a relation of many to many between User & Role table. In order to satisfy many to many relationship between the User and the Role table, multiple Users will have multiple roles.

We can obtain a unidirectional relationship between User & Role entity by applying @ManyToMany on the relational field at any side. For example, if we want to have a many to many relationship from User to Role, we need to add a field with type List<Role> in the User entity. It means many users will have many roles. Hence, we need to apply @ManyToMany on the field with a type List<Role> in the User entity. For example, below code demonstrates the concept.




```
import java.util.List;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class User {

   @Id
   @GeneratedValue
   private Integer userId;

   private String userName;
   private String userPwd;

    @ManyToMany
   private List<Role> roles;
}
```

Bidirectional Relationships

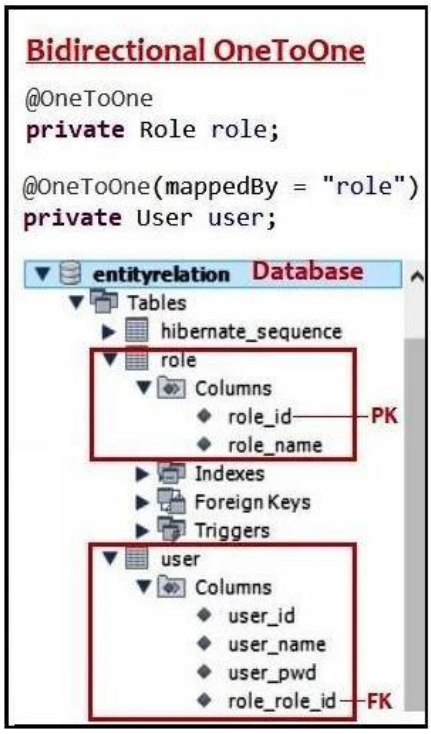
A bidirectional relationship has both an owning side and an inverse side. A unidirectional relationship has only an owning side.

The owning side of a relationship determines how the Persistence runtime makes updates to the relationship in the database.

The inverse side of a bidirectional relationship must refer to its owning side by using the mappedBy element of the @OneToOne, @OneToMany, or @ManyToMany annotation.

- **One to One relationship**

In order to satisfy the bidirectional relationship, we need to apply @OneToOne on both the sides ie. on the field with type Role in User entity and also on the field with type User in the Role entity. Additionally, we need to have mappedBy attribute into any side of @OneToOne to tell JPA/Hibernate that the mapping is already done by other sideand don't create additional column. For Example, below code demonstrates how we will create this relation between two tables using annotation @OneToOne.



```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity
public class User {

   @Id
   @GeneratedValue
   private Integer userId;
   private String userName;
   private String userPwd;

   @OneToOne
   private Role role;
}
```

And,

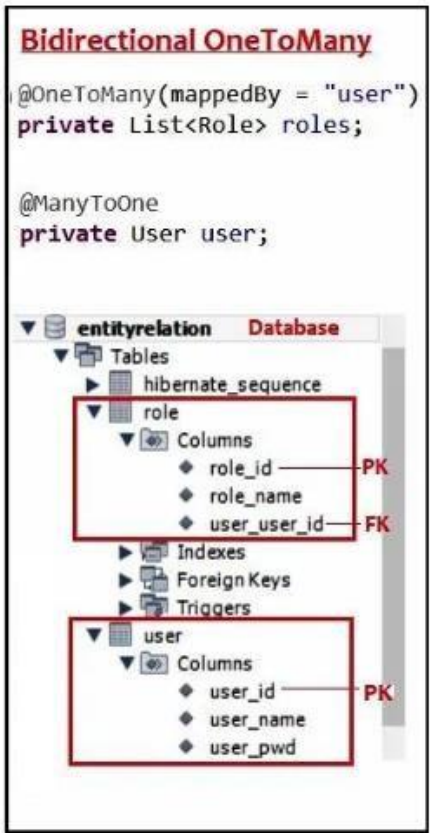
```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity
public class Role {

   @Id
   @GeneratedValue
   private Integer roleId;
   private String roleName;

   @OneToOne(mappedBy = "role")
   private User user;
}
```

- **One to Many Relationship**

In order to satisfy the bidirectional relationship, we need to apply @OneToMany at one sides ie. on the field with type

List<Role> in User entity and @ManyToOne at other side also ie. on the field with type User in the Role entity.

Additionally, we need to have mappedBy attribute in @OneToMany to tell JPA/Hibernate that the mapping is already done by other side and don't create additional column. For Example, below code demonstrates how we will create this relation between two tables using annotation @ManyToOne and @OneToMany.



```
import java.util.List;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;

@Entity
public class User {

   @Id
   @GeneratedValue
   private Integer userId;

   private String userName;
   private String userPwd;

   @OneToMany(mappedBy = "user")
   private List<Role> roles;
}
```

And,

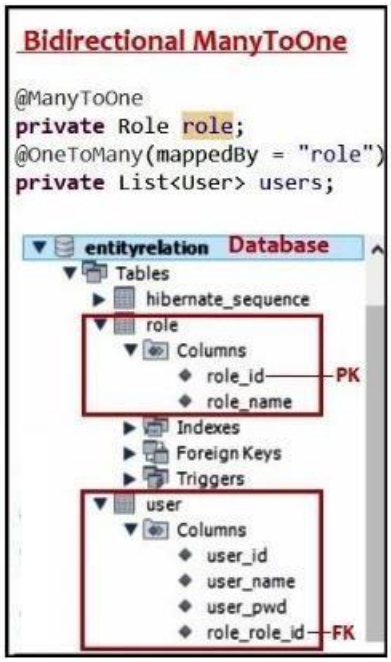
```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

@Entity
public class Role {

@Id
@GeneratedValue
private Integer roleId;

private String roleName;

@ManyToOne
private User user;
}
```

- **Many to one relationship**

Use-case: Let's assume that we have to maintain a relation between User & Role table. In order to satisfy Many to One relationship between the User and the Role table, multiple Users will have only one Role.

In order to satisfy the bidirectional relationship, we need to apply @ManyToOne at one sides ie. on the field with type Role in User entity and @OneToMany at other side also ie. on the field with type List<User> in the Role entity.

Additionally, we need to have mappedBy attribute in @OneToMany to tell JPA/Hibernate that the mapping is already done by other side and don't create additional column. For Example, below code demonstrates how we will create this relation between two tables using annotation @ManyToOne and @OneToMany.



```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

@Entity
public class User {

   @Id
   @GeneratedValue
   private Integer userId;

   private String userName;
   private String userPwd;

   @ManyToOne
   private Role role;
}
```

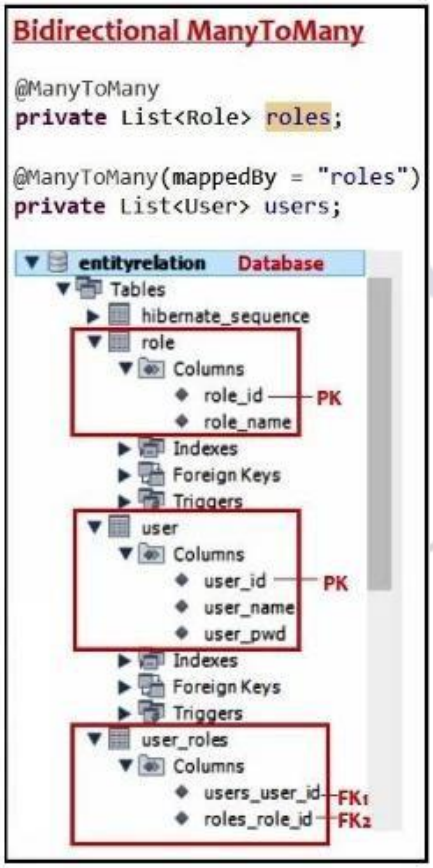
And,

```
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;

@Entity
public class Role {

   @Id
   @GeneratedValue
   private Integer roleId;

   private String roleName;

   @OneToMany(mappedBy = "role", cascade = CascadeType.ALL)
   private List<User> users;
}
```

- **Many to many relationship**

Use-case: Let's assume that we have to maintain a relation of many to many between User & Role table. In order to satisfy many to many relationship between the User and the Role table, multiple Users will have multiple roles.

In order to satisfy the bidirectional relationship, we need to apply @ManyToMany at both the sides ie. on the field with type List<Role> in User entity and @ManyToMany at other side also ie. on the field with type List<User> in theRole entity. Additionally, we need to have mappedBy attribute in any side to tell JPA/Hibernate that the mapping is already done by other side and don't create additional column. For Example, below code demonstrates how we willcreate this relation between two tables using annotation @ManyToMany.



```
import java.util.List;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class User {

   @Id
   @GeneratedValue
   private Integer userId;

   private String userName;
   private String userPwd;

   @ManyToMany
   private List<Role> roles;
}
```

And,

```
import java.util.List;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class Role {

   @Id
   @GeneratedValue
   private Integer roleId;

   private String roleName;

   @ManyToMany(mappedBy = "roles")
   private List<User> users;
}
```

Queries and Relationship Direction

Java Persistence query language and Criteria API queries often navigate across relationships. The direction of a relationship determines whether a query can navigate from one entity to another. For example, a query can navigate from LineItem to Product but cannot navigate in the opposite direction. For Order and LineItem, a query could navigate in both directions because these two entities have a bidirectional relationship.

Cascade Operations and

RelationshipsWhat Is Cascading?

Entity relationships often depend on the existence of another entity, for example the Person–Address relationship. Without the Person, the Address entity doesn't have any meaning of its own. When we delete the Person entity, ourAddress entity should also get deleted.

Cascading is the way to achieve this. When we perform some action on the target entity, the same action will be applied to the associated entity.

JPA Cascade Type

Entities that use relationships often have dependencies on the existence of the other entity in the relationship. Forexample, a line item is part of an order; if the order is deleted, the line item also should be deleted. This is called a cascade delete relationship.

The javax.persistence.CascadeType enumerated type defines the cascade operations that are applied in the cascadeelement of the relationship annotations. The below table, lists the cascade operations for entities.

Cascade Operations for Entities

Cascade Operation	Description
ALL	All cascade operations will be applied to the parent entity's related entity. All is equivalent to specifying cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}
DETACH	If the parent entity is detached from the persistence context, the related entity will also be detached.
MERGE	If the parent entity is merged into the persistence context, the related entity will also be merged.
PERSIST	If the parent entity is persisted into the persistence context, the related entity will also be persisted.

REFRESH	If the parent entity is refreshed in the current persistence context, the related entity will also be refreshed.
REMOVE	If the parent entity is removed from the current persistence context, the related entity will also be removed.

JPA Cascade Remove

Example:

Cascade delete relationships are specified using the cascade=REMOVE element specification for @OneToOne and @OneToMany relationships. For example:

```
@OneToMany(cascade=REMOVE, mappedBy="customer")
public Set<Order> getOrders() { return orders; }
```

JPA Cascade Persist

The cascade persist is used to specify that if an entity is persisted then all its associated child entities will also be persisted. The following syntax is used to perform cascade persist operation: -

```
@OneToOne(cascade=CascadeType.PERSIST)
```

JPA Cascade Persist Example

In this example, we will create two entity classes that are related to each other but to establish the dependency between them we will perform cascading operation.

This example contains the following steps: -

- Create an entity class named as StudentEntity.java under com.javatpoint.jpa.student package that containsattributes s_id, s_name, s_age and an object of Subject type marked with cascade specification.

StudentEntity.java

```
import javax.persistence.*;
importcom.javatpoint.jpa.subject.Subject;
@Entity
@Table(name="student")
public class StudentEntity {

    @Id
    private int s_id;
    private String s_name;
    private int s_age;

    @OneToOne(cascade=CascadeType.PERSIST)
    private Subject sub;

    public Subject getSub() {
      return sub;
    }
    public void setSub(Subject sub) {
      this.sub = sub;
    }
     public StudentEntity(int s_id, String s_name, int s_age , Subject sub) {
      super();
      this.s_id = s_id;
      this.s_name = s_name;
      this.s_age = s_age;
      this.sub=sub;
    }
    public
      StudentEntity() {
      super();
    }
    public int getS_id() {
      return s_id;
    }
    public void setS_id(int s_id) {
      this.s_id = s_id;
    }

    public String getS_name() {
      return s_name;
    }
```

```
        public void setS_name(String s_name) {
        this.s_name = s_name;
    }
    public int getS_age() {
       return s_age;
    }

    public void setS_age(int s_age) {
       this.s_age = s_age;
    }
}
```

- Create another entity class named as Subject.java under com.javatpoint.jpa.subject package.

Subject.java

```
package com.javatpoint.jpa.subject;
import javax.persistence.*;

  @Entity
  @Table(name="subject")
  public class Subject {


private String name;
private int marks;
@Id
private int s_id;
public Subject(String name, int marks, int s_id) {
super();
        this.name = name;
        this.marks = marks;
            this.s_id=s_id;
    }
        public Subject()
    {
       super();
    }
    public String getName() {
       return name;
    }
    public void setName(String name) {
       this.name = name;
    }
    public int getMarks() {
       return marks;
    }
    public void setMarks(int marks) {
       this.marks = marks;
    }
  public int getS_id() {
       return s_id;
    }
    public void setS_id(int s_id) {
       this.s_id = s_id;
    } }
```

- Now, map the entity class and other databases configuration in Persistence.xml file.

Persistence.xml

```
 <persistence>
<persistence-unit name="Student_details">

  <class>com.javatpoint.jpa.student.StudentEntity</class>
  <class>com.javatpoint.jpa.subject.Subject</class>
   <properties>
     <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/studentdata"/>
     <property name="javax.persistence.jdbc.user" value="root"/>
     <property name="javax.persistence.jdbc.password" value=""/>
     <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
     <property name="eclipselink.logging.level" value="SEVERE"/>
     <property name="eclipselink.ddl-generation" value="create-or-extend-tables"/>
   </properties>

  </persistence-unit>
</persistence>
```

- Create a persistence class named as StudentCascade.java under com.javatpoint.jpa.cascade package to persistthe entity object with data.

StudentCascade.java

```
package com.javatpoint.jpa.cascade;
import javax.persistence.*;
  import com.javatpoint.jpa.student.*;
  import com.javatpoint.jpa.subject.Subject;
  public class StudentCascade {
      public static void main( String[ ] args ) {

          EntityManagerFactory emf = Persistence.createEntityManagerFactory( "Student_details" );

          EntityManager em = emf.createEntityManager( );
         em.getTransaction().begin();

          StudentEntity s1=new StudentEntity();
          s1.setS_id(101);
          s1.setS_name("Vipul");
          s1.setS_age(20);

          StudentEntity s2=new StudentEntity();
          s2.setS_id(102);
          s2.setS_name("Aman");
          s2.setS_age(22);

          Subject sb1=new Subject();
          sb1.setName("ENGLISH");
          sb1.setMarks(80);
          sb1.setS_id(s1.getS_id());

           Subject sb2=new Subject();
          sb2.setName("Maths");
          sb2.setMarks(75);
             sb2.setS_id(s2.getS_id());

          s1.setSub(sb1);
          s2.setSub(sb2);

          em.persist( s1 );//No need to perform persist operation separately for different entities.
          em.persist(s2);

          em.getTransaction().commit();
          em.close( );
          emf.close( );

  } }
```

Output:

After the execution of the program, the following tables are generated under MySQL workbench.

- Student Table - To fetch data, run select * from student in MySQL.

S_ID	S_NAME	S_AGE
101	Vipul	20
102	Aman	22

- Subject Table - To fetch data, run select * from subject in MySQL.

NAME	MARKS	S_ID
ENGLISH	80	101
Maths	75	102