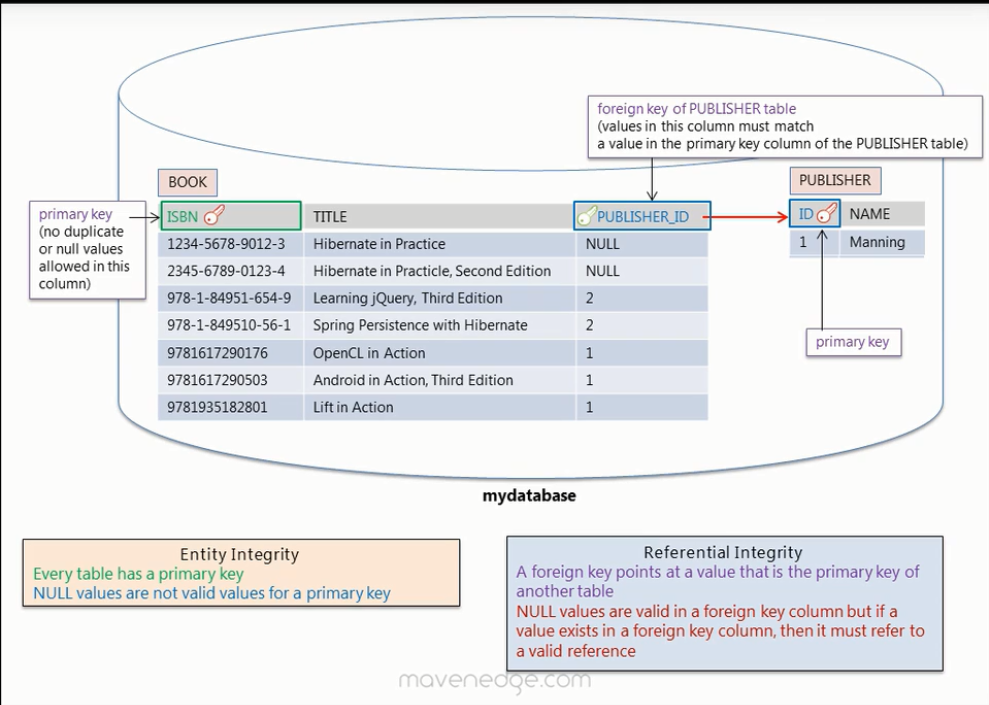
**Hibernate 101**

**Object/relational Persistence**

1. In a Database we have 2 main property that must be followed
   1. **Entity Integrity**: every table has a primary key and NULL values are not allowed for primary key.
   2. **Referential Integrity**: A foreign key points at a value that is the primary key of another table. NULL value is valid in foreign key column but if a value exits in a foreign column, then it must refer to a valid reference.



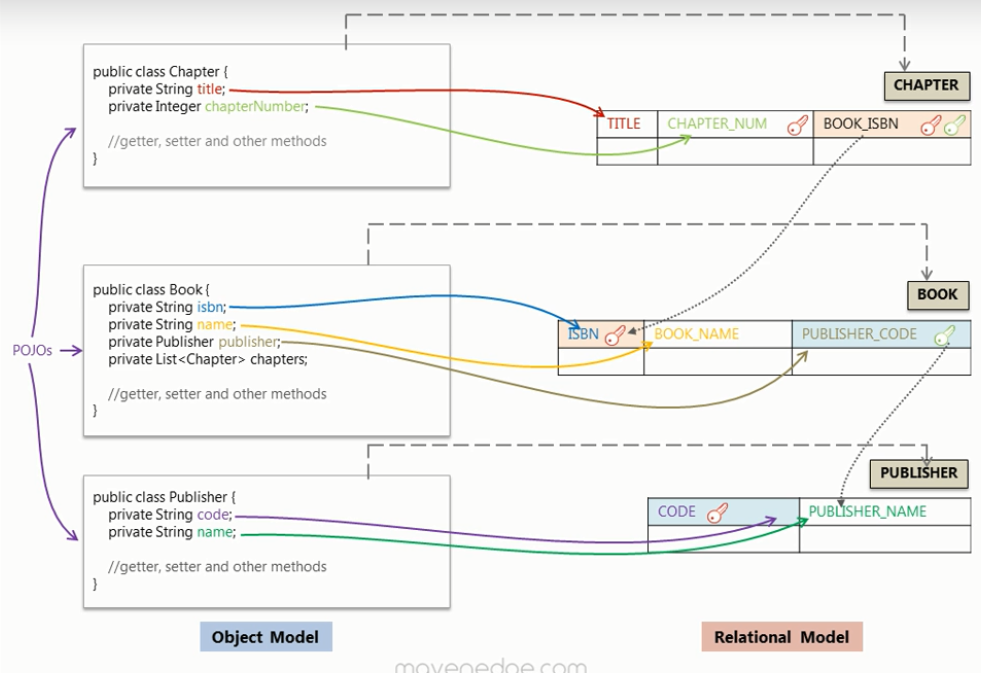
1. Object Model: Graphs of object

Relational Model: Table like format

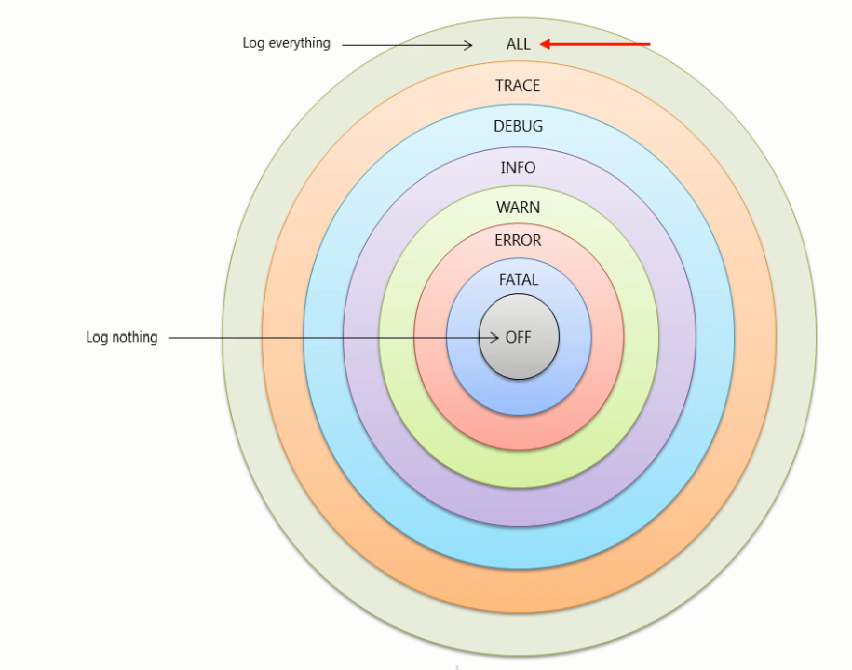
1. Object Model is more granular, supports inheritance, and Identity is based on object equality while in relational model the identity equality is via primary key.

**Hibernate and JPA Annotations**

1. Example of Object Relational mappings:



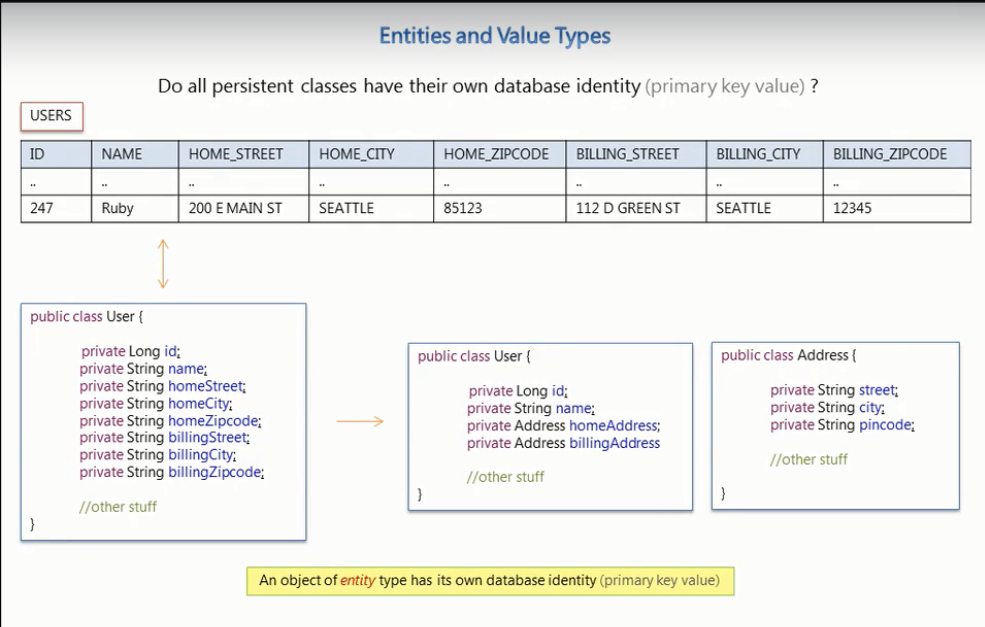
1. We make 1 session factory for 1 data source as it is resource intensive.
2. Make use for log4j for logging purpose:



1. Transactions in Hibernate? These are a group of operations that run as a single unit of work. When ever we modify the state of an object inside a transaction(Update operation) , then it performs **Dirty Checking** and updates that object in data base. Only when the transactions is committed then only we see the entry/update or deletion from database.
2. We must have a default constructor so that hibernate can instantiate objects using reflection.

**Mapping Concepts**

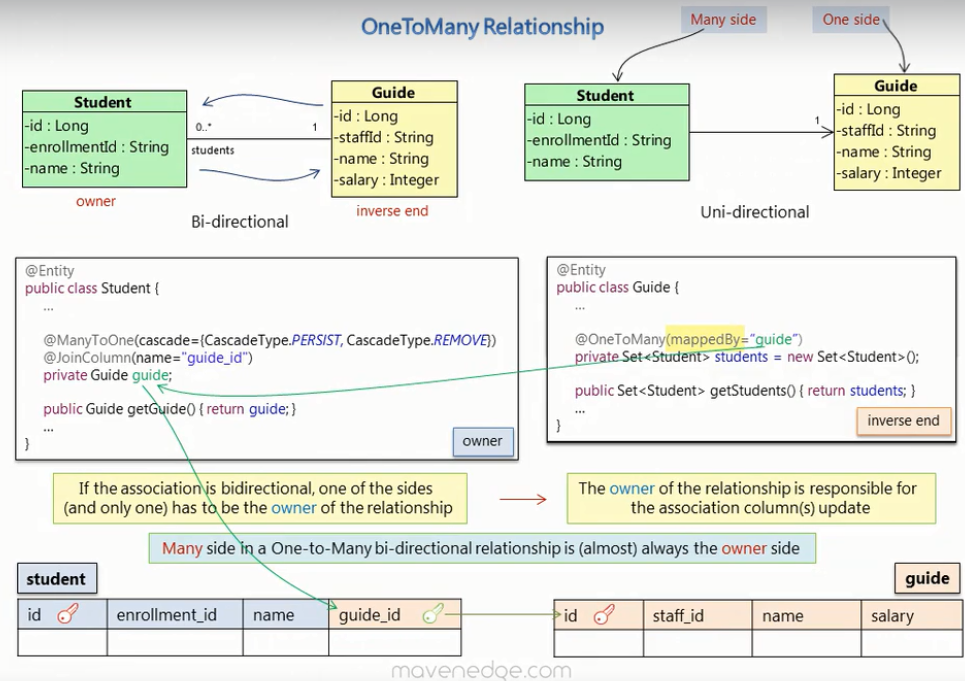
1. Aggregation indicates a relationship between a whole and its parts unlike Composition which is a strong form of aggregation and when whole is destroyed its parts also destroy with it.
2. Observe this: right side 2 class approach is better design. Why we gave ‘id’ to user class? Cause we cared about that only to uniquely identify.



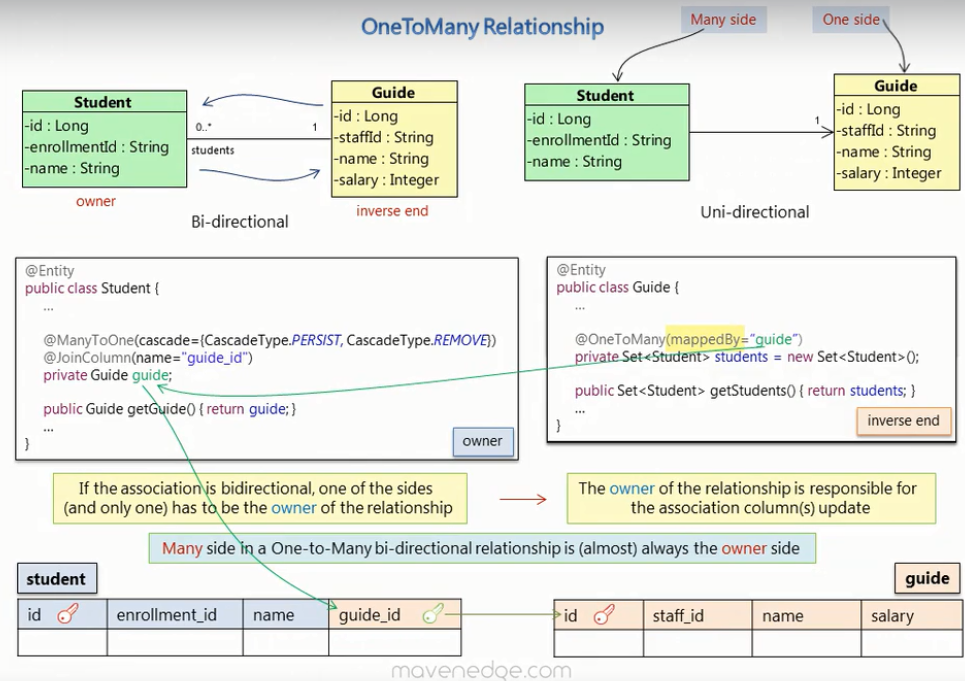
1. A object of **entity type** has its own datatype identity (primary key value).
2. A **value type** objects are identified through the owning entity.
3. **Note:** During the establishment of a bi-directional relationship we make use of the **mappedBy** attribute.

In the below example this attribute tells hibernate to load the collection of students using the foreign key column already mapped by the guide attribute.

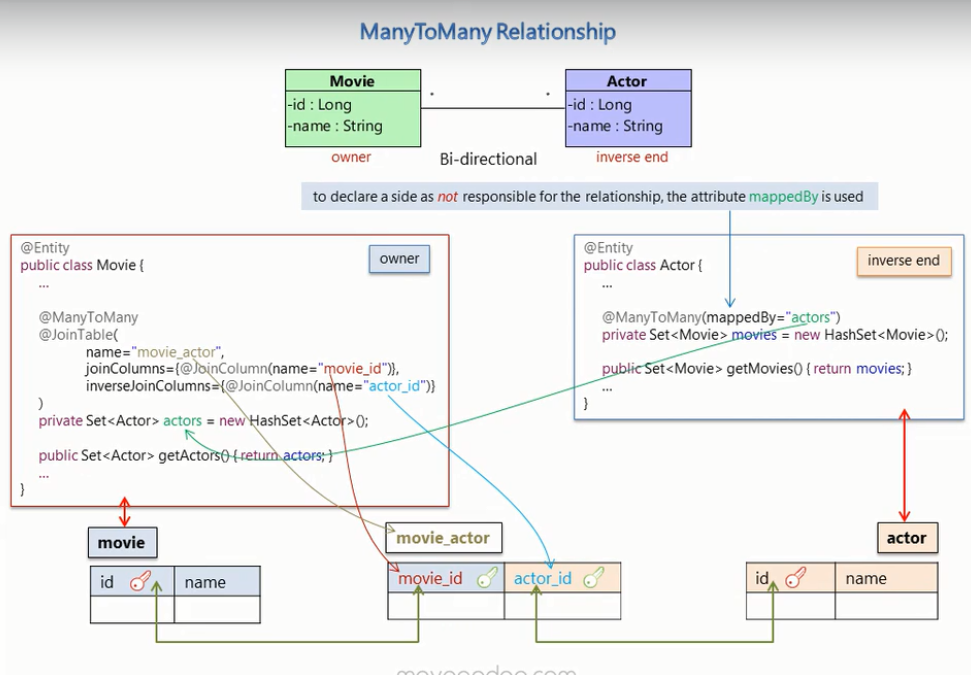
1. In a Bi-directional relationship one of the sides is the owner of the relation and is responsible for the association(guide\_id) column update. **Many** side in a such cases are usually the owners.



1. **NOTE:** If we update the inverse end then changes are not updated for the owner table (association column don’t get updated). While on updating the owner end everything gets updated.
2. **Owner** is the entity that is persisted to the table that has the foreign key column.
3. **MappedBy:** To declare a side as not responsible for the relationship, this attribute is used.
4. **One to One :** Example below:

****

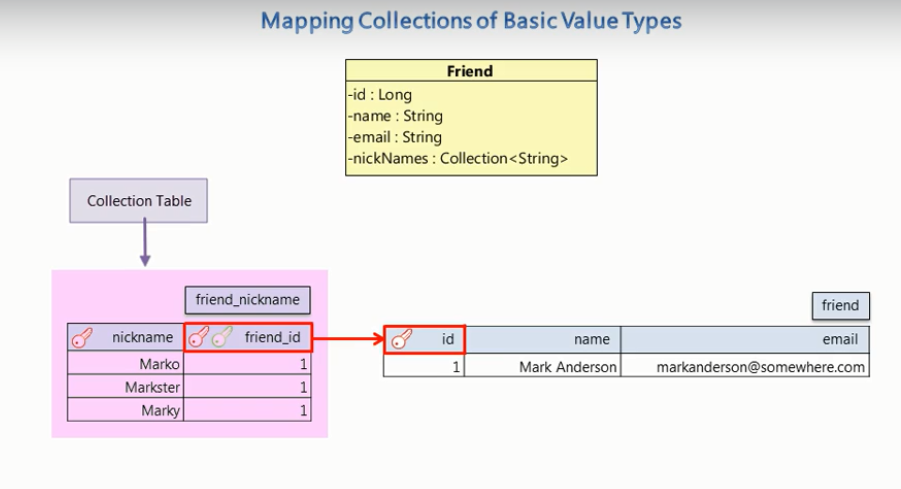
1. **Many To Many:**  Here we make use of a Join Table attribute in the following way:



1. **Storing Collection of basic value type:**  we store it via separate table for the collection.

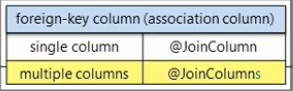
We use such syntax : @Collectionable(name=”TableName” @joinColumn(name=”xyz\_id”)).

To avoid duplicate rows in collection table we usually have a composite primary key.



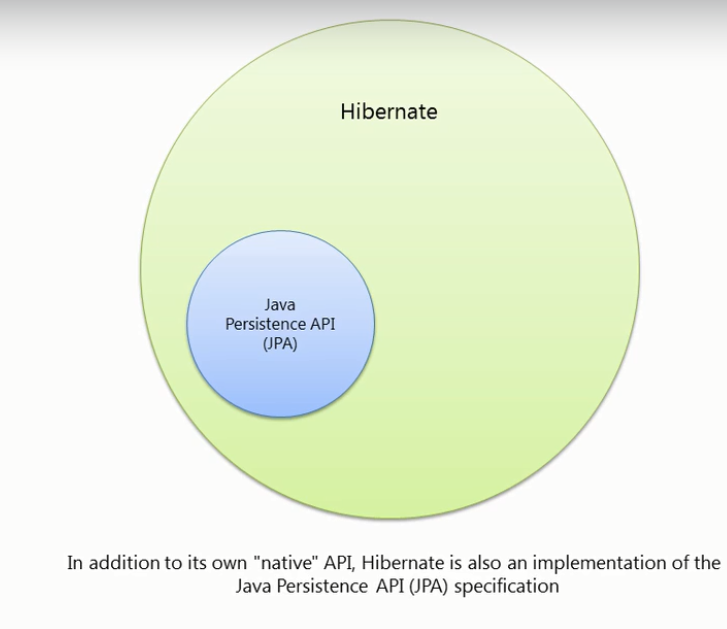
v

1. **Composite Primary Key:** A combination of more than 1 table columns that uniquely identifies a record. No matter how gud our selection is but it is not recommended to use random selected columns for composite primary key.
2. **Observe :**

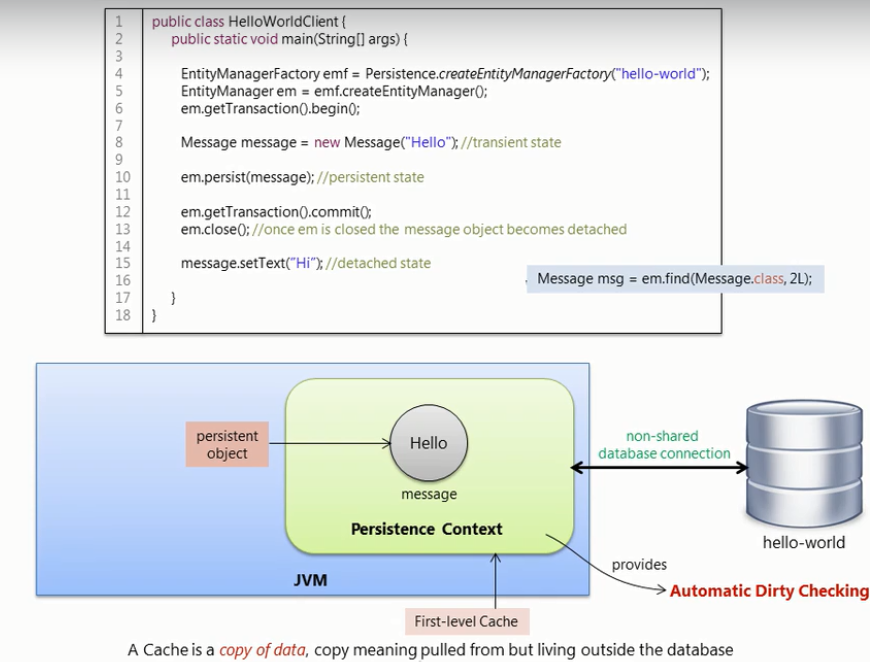
****

**Getting Started with JPA**

1. JPA is a java specification for accessing , persisting , and managing data between Java Objects and a relational database. It provides guild lines that a framework can implement to be considered JPA compatible.



1. In the “persistence.xml” file in the <persistence-unit> tag we give one property called “transaction-type=RESOURCE\_LOCAL” which tells that we will provide a Entity Manager by our self. If we use a Application Server like Glassfish , then it provides one and will use “transaction-type=JTA”.
2. **States Of Object:**

****

1. **Transient State :** Normal state when object is created.

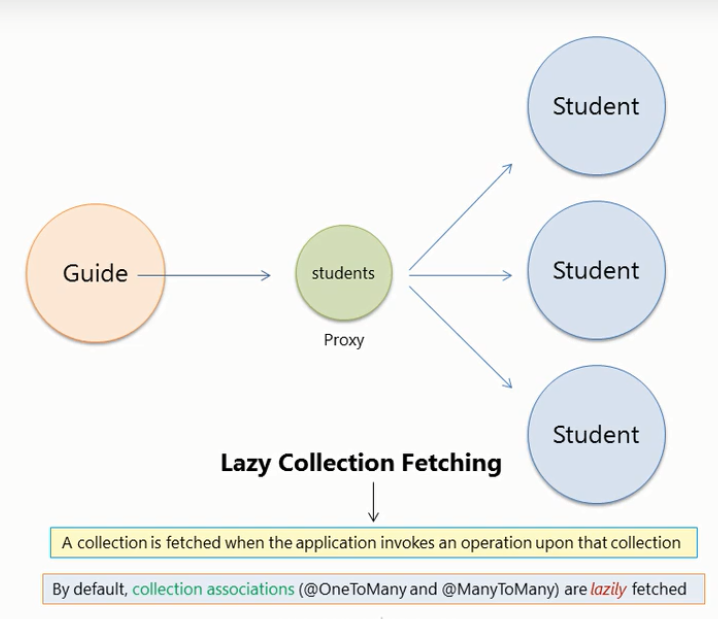
**Persistent State:** The message object has a database identity and will have its primary key value set as its database Identifier. Once the message object becomes a persistent object then it gets managed by the entity manager for the duration of the Transaction.

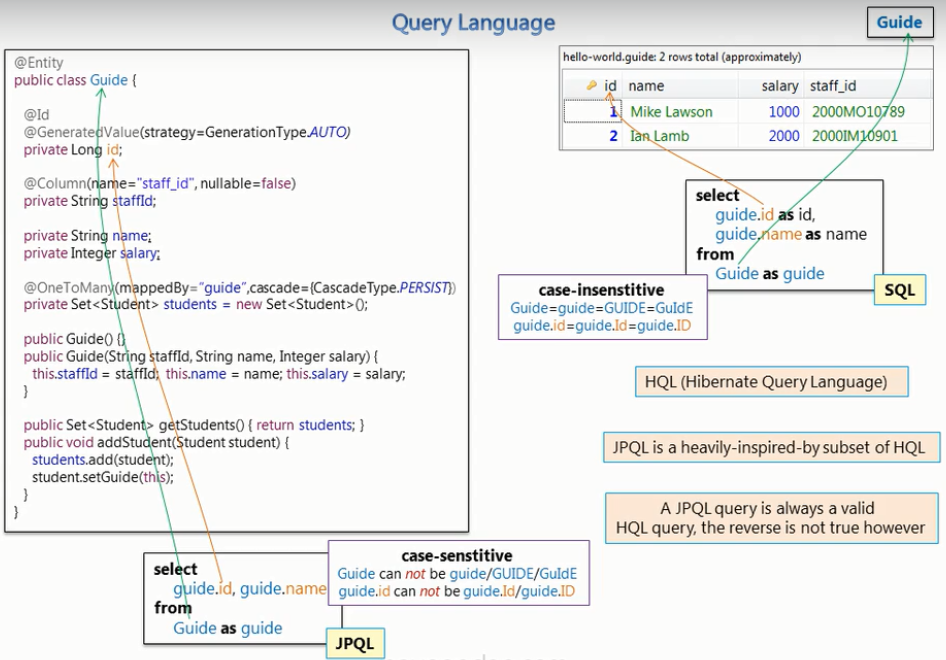
**Detached State:** Once Entity Manager is closed the message Becomes detached state

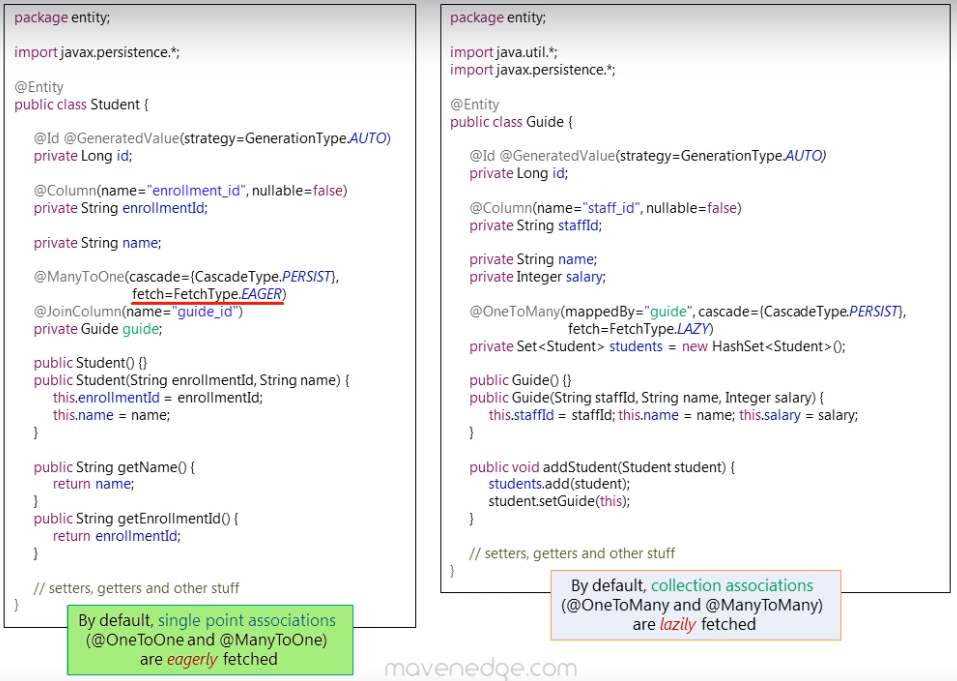
1. An Entity Manager has a persistent context.
2. **Caching Object:** A cache is a copy of data, copy meaning pulled from but living outside the database.

When a object is read for the first time from a external storage or relational Database then a copy of it is stored in the cache.

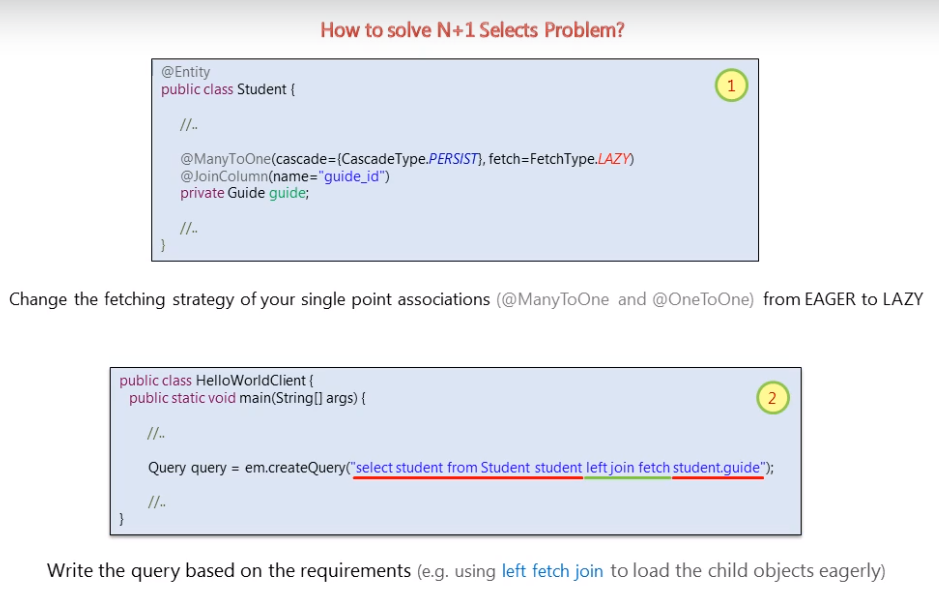
1. **First Level Cache:** Its Scope is Entity manager. Each EM comes with its own cache.
2. **Second Level Cache:** Its scope is Entity Manger Factory.
3. **Fetching:**



1. Also note that By default, single point associations (@oneToOne and @ManyToOne) are eagerly fetched.
2. J**ava Persistence Query Language**: Use to query Entity and their data attribute and is translated to SQL at Runtime. IN JPQL, name of Entity and its attribute are case sensitive and everything else is case in-sensitive.
3. HQL(Hibernate Query Language) : A JPQL Query is always a valid HQL query, the reverse is not true however.
4. How to Query Entites? We start by making a query object and passing the JPQL query.
5. **Inheritance Mapping and Polymorphic Query:** There are 3 strategy to implement such scenario-
   1. **Single table:** class hierarchy is represented in one table. A discriminator column identifies the type and subclass. (DType column is added in table). Performance of Polymorphic query and derived class query in this strategy is very good as there is no join. Bad thing is that all properties in subclass must not have not null constraint.
   2. **Joined table:** Each class has its table and querying a subclass entity requires joining the tables, so poor polymorphic query performance. The superclass has a table and each subclass has a table that contains only the un-inherited properties (*subclass tables have primary key that is a foreign key of the superclass).*
   3. **Table per class:** Each table contains all the properties of the concrete class and the properties that are inherited from its super classes. The database identifier and its mapping must be present in the superclass, to be shared in all subclasses and their tables. The child tables here have no relation between them, so while doing polymorphic query so use “Union” instead of “Join”. So not a good performance.
6. **N+1 Problem:**

****

**i.e. N+1 selects: 1** Select for all parent objects and **1** select for each child object. Basically, N selects here are extra which hurts the performance. **Solution** change the fetching strategy using the annotation or write our query as per requirement..



1. At one time it might me the case when the strategy is lazy but like in print statement we access the child table value, then again, the N+1 problem occur. In that case we can use left join as suggested above. This has also one problem that in case the table data is huge than solution by 2nd approach will cause a single big query which the DB might not handle. so better to use N+1 problem case as there we will large number of small queries…A better solution is the Batch Fetching Mode, Lets talk about is now