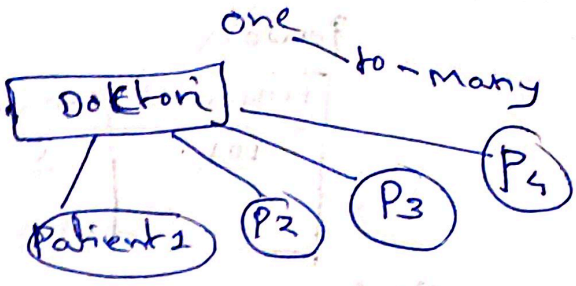
Associations (Relationship Annotation)

→ OnetoMany!



Doctor	Patient
D-id;	P-id;
D-name;	P-name;
	P-emailid;
	doctor;

List<Patient> list=
new ArrayList<>();

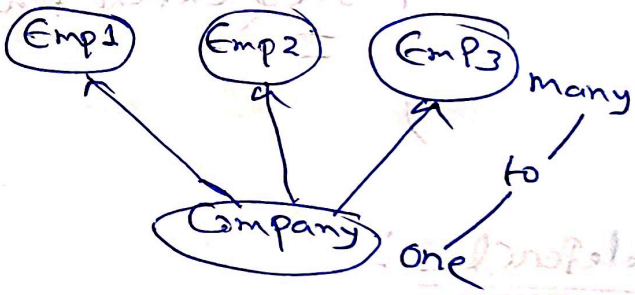
Doctor:

- d-id; @id
  @generatedValue (generationType.Auto)
- d-name; @Column
- @OneToMany (mappedBy = "doctor")
  List <Patient> list = new ArrayList<>(); (or) set property also can use.

Patient:

- P-id @id, @generatedValue,
- P-name
- @ManytoOne @JoinColumn(name="d-id");
- Doktor doctor;

→ ManytoOne!



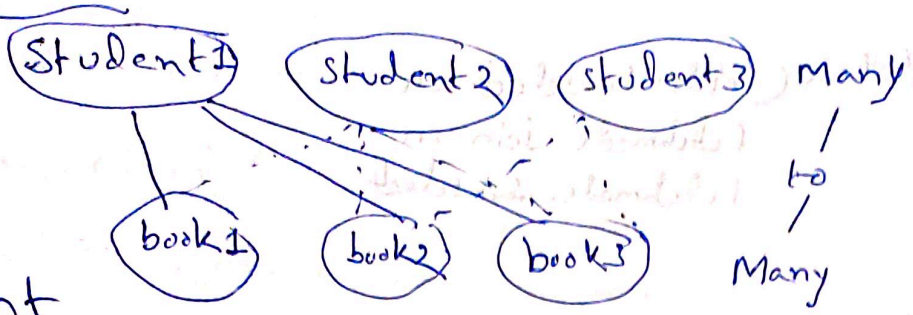
Employees:

- @id
- emp-id;
- @Column
- emp-name;
- @manytoone
- @joinColumn(name="c-id")
- Company company;

Company:

- @id
- c-id;
- c-code; @Column
- C-address; @Column

Many-to-many !



Student

- s_id; @id @Generatedvalue
- s_name; @column
- s_age; @column

```
@manytomany(cascadetype=cascade.All)
@jointable(name="s_b", joincolumns={
 @joincolumn(name="s_id")}inversejoincolm-
Set<Book> set = new HashSet<>();
```

Book

- b_id;
- b_name;
- b_author;

```
@ManytoMany (Cascadetype = Cascade.All, targetEntity = "Book.class")
@Jointable (name = "stu_book", joincolumns = {@Joincolumn (name = "s_id")}
, inversejoincolumns = {joincolum (name = "b_id")}).
Private Set<Book> set = new HashSet();
```

One-to-one !

```
@Entity
@Table (name = "CUSTOMER")
Public class Customer1 {
    @Id
    @Column (name = "txn_id")
    @GeneratedValue (generator = "gen")
    @GenericGenerator (name = "gen", strategy = "foreign",
                       parameters = {@parameter(
                           name = "property"
                           value = "txn")})
    Private long id;
    --
    @onetoone
    @PrimaryKeyJoinColumn
    Private Txn1 txn;
```

```
public class Txn1 {
    --
    @onetoone (mappedBy = "txn") @Cascade(value = Cascadetype.all)
    Private Customer1 customer;
```

✡ Inheritance @annotation type : (Mapping Strategy)

3 types of inheritance :

- → TablePerClass
- → TablePerSubclass
- → TablePerConcreteClass

→ TablePerSubclass :

Person

person_id	F_name	L_name
101	S	U
102	R	K
103	M	U

Employee

person_id	DoJ	Sal
101	21-06-90	2000

Owner

person_id	Age	Share price
102	58	$2000

```
@Entity @Table
@Inheritance(strategy = InheritanceType.JOINED)
class Person {
    =
}

@Entity @Table
@PrimaryKeyJoinColumn(name = "person_id")
class Employee extends Person {
    Date DoJ;
    int Sal;
}
```

⇒ TablePerConcreteClass:

Person

Person-id	F-name	L-name
101	S	U

Employee

Person-id	F-name	L-name	DoJ	Sal
101	R	K	21-06-90	2000

owner

```
@Entity, @Table
@Inheritance(strategy = InheritanceType.table-per-class)
class person{
    =
}

@Entity, @Table
@AttributeOverrides({
        @AttributeOverride(name = "firstname", column = @Column(name = "FIRSTNAME")
        @AttributeOverride(    "          "  ) })
class Employee2 extend person{
    =
}
```

⇒ TablePerClass:

```
@Inheritance( strategy = InheritanceType.single-table)
@discriminatorColumn
@discriminatorvalue
```

Fetch strategy:

```
@fetch(fetchmode.select or
       fetchmode.join or
       fetchmode.subselect
```

fetchType:

```
@OneToMany(fetch = fetchtype.LAZY)
           (fetch = fetchtype.EAGER)
```

How to Create Session!

```
Session session = entitymanager.unwrap(Session.class)
SessionFactory factory = entitymanager.unwrap(sessionfactory.class)
```

@Query:

It's a good approach to place the query definition above method in repository class rathenthan inside entity class as namedquery.

```
@Query("u from user u where u.status = 1")
Collection<user> findAllActiveUsers();
```

⇒

```
@Query("u from User u where u.status = ?1 and u.name = ?2")
User findUserByStatusAndName(Integer status, String name);
```

⇒

```
@Query("u from User where u.status = :status and u.name = :name)
User findByStatusAndName(@Param("status") Integer status,
                         @Param("name") String name);
```

NativeQuery:

```
@Query("u from user when u.status = :status and u.name = :name"
       nativeQuery = true)
User findByUserstatusAndUsername(@Param("status") Integer
       userstatus, @Param("name") String username);
```

⊛ NamedQuery:

NamedQuery is an approach to provide in Entity class.

```
@NamedQueries({
    @NamedQuery(name="FindByEmpNumber",
                query="from Dept where empno= :empno"),
    @NamedQuery(name="FindByDesignation",
                query="from Dept where desg= :desg").})

@NamedNativeQueries({
    @NamedNativeQuery(name="__"
                query="__") __})
```

⇒ In service class:

```
Query<Employee> query = session.createNamedQuery("FindByDesignation",
                                                  Employee.class)
query.setParameter("desg","Manager");
Employee e= query.getSingleResult();
```

⊛ Second Level Cache:

Second level cache is sessionfactory scoped i.e it is shared with all session created with same sessionfactory.

Application.properties:

```
hibernate.cache.use_second_level_cache=true
hibernate.cache.region.factory_class = ehcacheRegionfactory
```

Making Entity class Cacheable:

```
@Entity
@Cacheable
@Cache(usage=CacheConcurrencyStrategy.READ-WRITE)
public class Foo{
    @id
    private eid;
    @column
    private ename;
}
```

Cache Concurrency Strategy

- READ-WRITE
- READ-ONLY
- NONSTRICT_READ_WRITE
- TRANSACTIONAL

Hibernate Version:

Latest stable version: 5.4 (2019-07-30)

old stable version: 5.0 (2017-01-19)

↳ Can speak in interviews

Hibernate 5 new feature:

1) Support for java 8 Date and Time:

for example if we use LocalDate type then Hibernate 5 orm perform its Conversion to JDBC date type during persistence process.

2) Support for stream query results.

3) Load object by multipleIds:

```
MultiIdentifierLoadAccess<Student> multi = session.byMultipleIds(Student.class);
List<Student> list = multi.multiLoad(1L, 2L, 3L);
```

4) Joining unAssociated entities in a query

we can use ~~inner~~ innerJoin in a createQuery function.

5) @Repeatable Annotation support

```
@NamedQueries({@NamedQuery(...),
               @NamedQuery(...)})
```

④ Component Mapping:

One student can have permanent address and temporary address. So here we can have student and Address class. Address will be a ~~has a real~~ Has-A-relationship.

It doesn't have primary key

Student
- S-id;
- S-name;
- Address temp-address;
- Address perm-address;

Address
- village
- city
- post
- pin

Here only student table get created and Address is a component in student.

Student

S-id	S-name	village	city	post	pin	village	city	post	pin

~~@Entity~~
~~@Table~~
~~@Embeddable~~

```
@Embeddable
class Address {
    @Column
    village;
    @Column
    city;
    @Column
    post;
    @Column
    pin;
```

```
@Entity
@Table
class student {
    @Id
    S-id;
    @Column
    S-name;
    @Embedded
    Address temp-address;
    @Embedded
    Address perm-address;
}
```

CompositeId:

If more than one field need to have primarykey then compositeid comes to picture.

```
@Embeddable
Public class RolesMenu {

    @Column(name = "...")
    Private String roleID;
    @Column(name = "...");
    private String menuitemID;
}
```

```
@Entity
@table
public class RolesMenuItem
  {

@EmbeddedId
Private RolesMenu menu;
  @Column,
Private String table;
  - - -
  }
```

Connectionpool:

```
<dependency>
    <groupId> org.hibernate</groupId>
    <artifactId> hibernate-c3po</artifactId>
    <version> 5.3.6.final</version>
</dependency>
```

→ With hibernate5, just adding above dependency is enough to enable c3po.

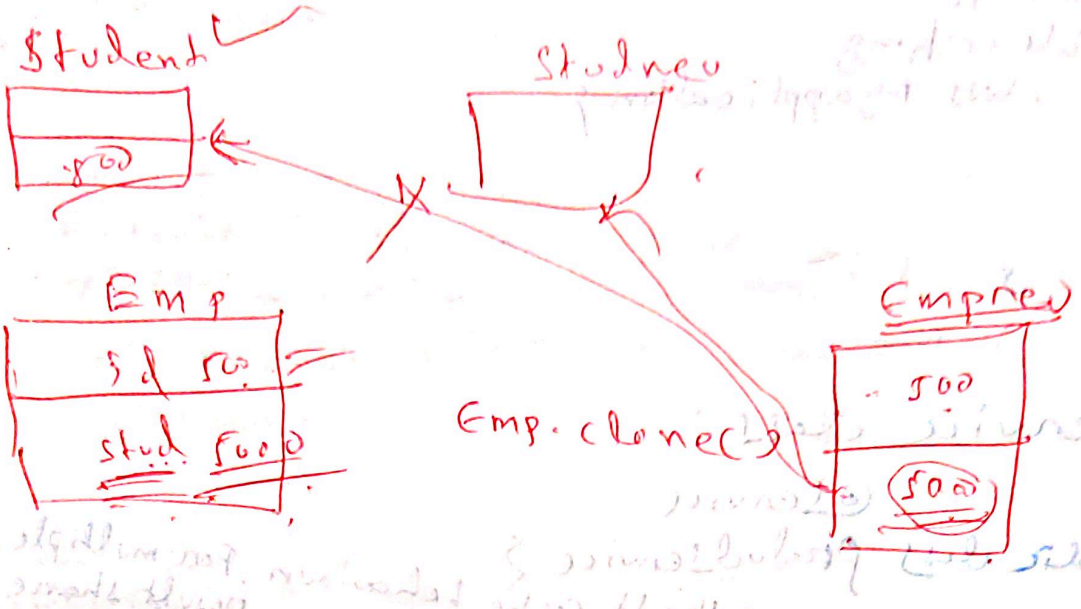
application.properties

```
hibernate.c3po.min-size = 5
          max-size = 20
          acquire-increment = 5
          timeout = 1800
```

Bydefault it will maintain atleast 3 Connection
" maximum 15 Connection.

Bydefault the connection will never expire from the pool.

Other approach for Connectionpooling are,
Hikaricp
DBCP
Proxool



Empnew.getStudents()

setAge(15);

⭐ Second Level Cache:

Ehcache:

↳ Add the Ehcache in pom.xml.

↓ springboot.starter.cache and org.ehcache

application.properties:

```
# Ehcache configuration
cache.cache-names = productscache
cache.type = ehcache
cache.ehcache.config = classpath:ehcache.xml
```

⇒ src/main/resource/ehcache.xml

ehcache.xml

```
<ehcache
    <cache name="productcache"
    =
</ehcache>
```

@SpringBootApplication
@EnableCaching
→ Public class MyApplication {

}

⇒ In service class :

@Service
public class ProductService {
→ It add cache behaviour. For multiple request the result share from cache storage

@Cacheable(value = "product cache")
Public Optional<Product> getProductById(int id) {

}

@CachePut(value = "product cache")
public Product updateProductById(...) {
saveOrUpdate(product);
}

@CacheEvict → Used in delete method to remove from cache.

}

⍟ 1+N problem:

⇒ In oneToMany relationship the 1+N problem arise.

For ex:

Parent table
→ PRec1
→ PRec2

Child Table
→ CRec1
→ CRec2
→ CRec3
→ CRec4

Here 3 select operation fired for two parent object.
This is called as 1+N problem.

→ 1 select query for load all parent object.
→ 2 select query for load child of respective parent object

⇒ To optimise the hibernate performance fetch strategy & Batchsize introduced.

e.g:
```
@OneToMany(mappedBy="custome")
@Fetch(FetchMode.SELECT)
Private Set<Invoice> invoices;
```

→ By Default the Fetchmode is select, so Here the N+1 problem occurs.

e.g:
```
@OneToMany
@Fetch(FetchMode.SELECT)
@BatchSize(size=10)
Private Set<Invoice> invoices;
```

no. of child record never matters

→ Suppose there are 20 parent record and 30 child record then total 3 select query get created.

1 select query to load all parent record.

2 select query to load the respective child record of parent object.

→ 1 select query to fetch 10 parent object with its child record

→ Another select query for next 10 parent object with its child record.

e.g:
```
@OneToMany
@Fetch(FetchMode.Subselect)
Private Set<Invoice> invoices;
```

↳ Here 2 select query created. One for to get all the parent object and another for fetch all child of respective parent object.

Here alway 2 select query created, doesn't matter how many parent and child object present.

⇒ e.g:

```
@OneToMany
@Fetch(FetchMode.JOIN)
Private Set<Invoice> invoices;
```

Here only one select query created to fetch all record. But duplicate record you may get. So by using Set it ll remove all duplicate record.

FetchType:

```
@OneToMany(Fetch=FetchType.LAZY)
```

Most commonly a decision is made to either load the collection when loading the entity (FetchType.EAGER) or to delay the loading until the collection on child object is used (FetchType.LAZY).

Cascading:

Suppose there is EmployeeEntity and AccountEntity.

```
@OneToMany(Cascade=CascadeType.ALL)
```

↓

This means any change happend on EmployeeEntity must cascade to AccountEntity.

If you save Employee, then all associated account will also be saved into database.

JPA CASCADETYPE:

- CascadeType.ALL
- CascadeType.PERSIST
- CascadeType.MERGE
- CascadeType.REFRESH
- CascadeType.REMOVE
- CascadeType.DETACH
- CascadeType.ALL

CascadeTyp.REPLICATE
- SAVE-UPDATE
- LOCK

} Hibernate Cascad Type

④ SpringBootDataJpa Using Join (InnerJoin, OuterJoin)

Entity Class: (one to many association) Dept ⟸ emp1, emp2, emp3

```
public class Employee
{
// @manytoone
}
```

```
public class Department
{
// @OneToMany
Set<Employee> employees = new
                          HashSet();
}
```

Repository:

```
public interface DepartmentRepository entends
                    JpaRepository<Department,
                                  Integer>{

@Query("SELECT new com.dto.DeptEmpDTO(d.name, e.name, e.email)" +
       "FROM Department d LEFT JOIN d.employees e")
List<DeptEmpDto> fetchEmpDeptDataLeftJoin();
}
```

```
public class DeptEmpDto {
    string empDept;
    string empname;
    String empEmail;
    Sting empAddres;
   //Construction
   // setter & getter
}
```

Service class:

```
public class ServiceClass{
@Autowired
Private DepartmentRepository deptRepository;
```

Object state in hibernate

Dt: 21/08/13

→ In a hibernate application, a pojo class object can be in any one of the following three state.

1) Transient state
2) Persistent state
3) detached state

Transient state:

→ When a pojo class object is newly created on a null value is assigned for the object then the object will be in a transient state

→ for ex:

```
Product p = new Product(); //P is in transient
Product p = null; //P is in transient
```

→ A transient state object is not associated with a session object of hibernate. So it is not associated with database also.

→ If we make any changes on a transient state object then the changes are not reflected on underlying database.

→ if a pojo class object is new then it can be converted from transient state to persistent state by doing save operation on that object.

→ To do Save operation, the methods are

1) Save()
2) persist()
3) Save or Update()

→ If a pojo class object is assigned with null then it can be converted to persistent state by doing load operation.

→ To do load operation, the methods are

1) load()
2) get()

→ If a transient state object is not converted to persistent state then finally the object goes to garbage at the end of application.

ex-1:

```
Product p = new Product(); // p is in transient state
p.setxxx(...);
Transaction tx = session.beginTransaction();
Session.save(p);
tx.commit();
```

ex-2:

```
Product p = null; // p is in transient state
Object o = session.get(Product.class, 111);
P = (Product)o; // p is in Persistent state
```

2) Persistent state:

When a pojo class object is added to session cache then that object will be in a state called persistent state.

→ If we make any changes on persistent state object then the changes are affected on database also.

→ A persistent state object can be moved to either detached state or transient state.

→ To move a persistent object to detached state then we need to call one of the following methods.

1) evict()
2) clear()
3) close()

To move an object from persistent state back to transient state then we need to call delete();

3) detached state:

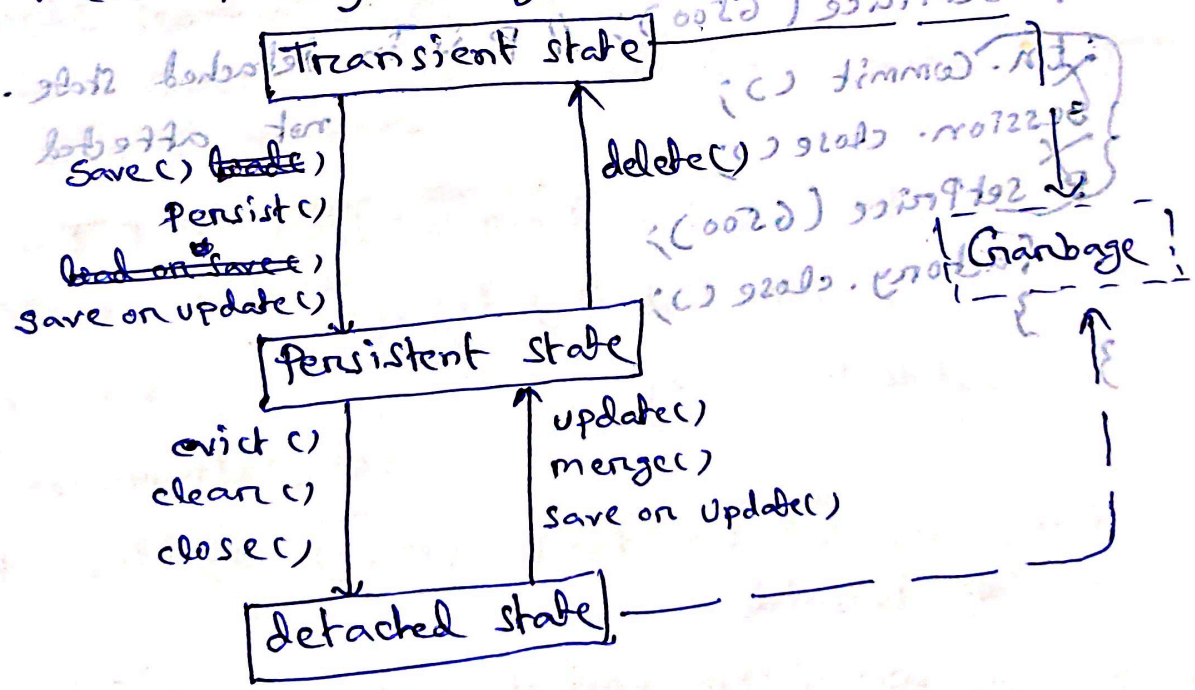
When an object comes out of a session cache then it will be entered into detached state.

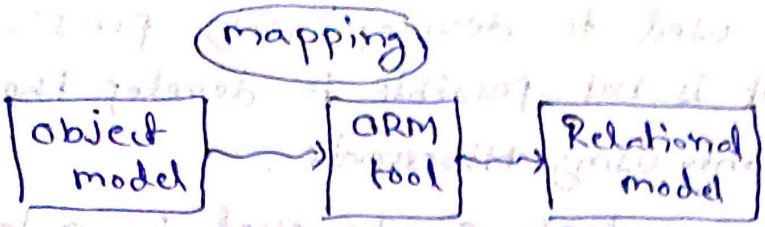
→ If any changes are made on detached state object then the changes are not affected on database.

→ A detached state object can be moved to persistent state, by calling one of the below methods

1) update()
2) merge()
3) saveOrUpdate()

→ if a detached object is not converted to persistent state then finally it goes to garbage.





More drawback of Jdbc :

1) In JDBC a Programmer has to take care about connection management. If not done properly then at some point of time in the execution there is a chance of getting errors.
2) If database table structure is modified after Jdbc application is created then again we need to modify the application at various places, whereever Jdbc code is written. It mean debugging process is complex.
3) Jdbc exceptions are checked, so we must handle exceptions, whenever Jdbc code is written in an application.
4) Jdbc is suitable for text format of data to tranfer but Complex for object format of data.
5) Jdbc application contain sql operation, if database is change then sometime sql operation also need to be modified. It means database operation of Jdbc are sometimes database dependent.

→ To overcome the above said problem of Jdbc, a top layer is added, on Jdbc This top layer is called an ORM tool.

What is Hibernate ?

→ Hibernate is an open source ORM tool, which acts as a middle layer between Java application and database, transfer to and from data in the form of objects.

→ Hibernate is an abstraction layer on top of Jdbc technology, which provides persistence service in the form of objects.