HIBERNATE

⇨ What is Hibernate (From Softtree (RedHat))?

- Hibernate is Opens source light weight ORM tool.
- Hibernate is a non-invasive framework, means it won't force the programmer to extends/implements any class/interface, and in hibernate we have all POJO classes so its lightweight.

⇨ What is ORM?

ORM stands for Object Relational mapping. It is a programming technique for converting data between relational database and object-oriented programming languages such as Java, C# etc.

⇨ Features of Hibernate?

- Hibernate persists java object into database (Instead of primitives).
- It generates efficient queries for java application to communicate with database.
- It provides fine grained exception handling mechanism. In hibernate we only have un-checked Exception.
- It supports synchronization between in memory java objects and relational records.
- Hibernate provides implicit connection pooling mechanism.
- Hibernate supports Inheritance, Associations, Collection.
- Hibernate supports a special query language (HQL) which database vendor independent.
- Hibernate has capability to generate primary keys automatically while we are storing the records into database.
- Hibernates address the mismatches between java and database.
- Supports over 30 dialects.
- Hibernate provides caching mechanism for efficient data retrieval.
- Lazy loading concept is also included in hibernate so you can easily load objects on startup time.
- Getting Pagination in Hibernate is quite simple.
- It supports automatic versioning of rows.

⇨ Required files in Hibernate?

- Any Hibernate application must contain:
    1. POJO class/ Persistence class/ Entity Class/ Domain Class
    2. Mapping Files
    3. Configuration Files
    4. Client App
- In order to work with hibernate we don't requrie any server as mandatory but we need hibenate liberary files (.jar).

⇨ Persistence class/Entity Class/POJO class

It should use have bean mapped with database table Client app uses this class Object to develop mapping persistence logic.

⇨ Mapping Files:

- In this file hibernate application developer specify the mapping from entity class name to table name and entity name and entity properties names to table column names. Mapping of an object-oriented data to relational data is done in this file.
- Standard name for this file is <domain-object-name.hbm.xml>
- Mapping file is metadata not data.
- Generally, an object contains 3 properties like Identity (Object name) State (Object values) Behavior (Object Model).

- **In ORM Storing of an Object is nothing but storing the state of an object, but not the Identity and behavior.**
- While constructing a mapping file in Hibernate, it is possible to write multiple java classes mapping in a single mapping file.
- Syntax of Mapping XML:
  <DOCKTYPE….>
  <hibernate-mapping>
  <class name=" Fully qualified name of the class" table= "database table name">
  <id name=" variable name" column= "column name" type="java/hibernate type"/>
  <property name=" variable name" column= "column name" type="java/hibernate type"/>
  .
  </class>
  </hibernate-mapping>
- Each hibernate mapping file must contain one <id> tag. Java object identified by the <id> property.
- <id> tag property corresponding column can be primary key (of) non-primary key in the database.
- In Mapping file class name and property name are Case-sensitive.
- But table name and column names are not case sensitive. When property name and column are same then we no need to give table attribute.
- In Mapping file, it is not required to map all properties of the entity (Pojo class) and all the column of table.
- As per our requirement we can configure required properties of the entity with required columns of the table.

⇨ Configuration file:

- Configuration is the file loaded into application when working with hibernate this configuration file contains 3 types of information:
  1. Connection Properties,
  2. Hibernate Properties,
  3. Mapping file name(s)
- Syntax of Configuration file:
  <hibernate-configuration>
  <session-factory>
  <! — Connection properties START>
  <property name=" connection.driver_class">Driver Class Name</property>
  <property name="connection-url">URL</property>
  <property name="connection-username">URL</property>
  <property name="connection-password">URL</property>
  <! — Connection Properties END>
  <! — Hibernate Properties START>
  <property name="show_sql">true</property>
  <property name="dialect">true</property>
  <property name="hbm2ddl.auto">true</property>
  <! — Hibernate Properties END>
  <! — Hibernate MApping START>
  <mapping resource="hbmfile1name.xml"/>
  <mapping resource="hbmfile2name.xml"/>
  <! — Hibernate Mapping END>
  </session-factory>
  </ hibernate-configuration>

⇨ Hibernate Dialect:

- Hibernate Dialect is used to convert the hibernate query language (HQL) into database specific query language.
- We can override our own Dialect by extending Dialect class.
- All the Dialect classes are present in org.hibernate.dialect package

⇨ Configuration class:

- The org.hubernate.cfg.configuration is a class and is the basic element of the Hibernate API.
- An Object-Oriented Representation of Hibernate configuration file along with mapping file is known as Configuration object.
- By default, hibernate reads configuration file with the name “hibernate.cfg.xml” which is located in class folder.
- If we and to change the file name or location then:
  Configuration cfg = new Configuration ();
  Cfg.configure(“/com/nareshit/xml/hibernate.cfg.xml”);
- We can add all the configuration properties like connection details Hibernate properties details and mapping location and details programmatically also in hibernate using “cfg.setProperty(“connection.driver_class”,”oracl.jdbc.driver.OracleDriver”);”
- Configuration Object we can create only once for an application at startup-time that us while initializing the application.

⇨ SessionFactory:

- SessionFactory is an Interface present in “org.hibernate” package.
- SessionFactoryImpl is an implemented class of SessionFactory interface.
- SessionFactory is a heavy weight object that has to be created only once per application.
- It is not a Singleton.
- It is Thread safe (Simply, thread-safe means that a method or class instance can be used by multiple threads at the same time without any problems occurring. We can achieve thread safety using synchronized feature.).
- It is Immutable Object.
- SessionFactory object provides lightweight session objects.
- SessionFactory object is the factory for session object.
- Generally, one SessionFactory should be created for one database.
- When we have multiple databases in your application you should create multiple SessionFactory objects.
- Cfg.buildSessionFactory().

⇨ Why SessionFactory is heavy weight?

SessionFactory encapsulates session object, connections, Hibernate properties caching and mappings.

⇨ Session:

- Session is an interface present in org.hibernate package and SessionImpl is an implemented class.
- It is single-thread (not-thread safe) short-lived object.
- It encapsulates the functionality of JDBC connection, statement obj.
- When session is open then internally a connection with the database will be established.
- It is a factory for Transaction objects.
- Session holds mandatory first-level first level cache of persistent object.
- After we complete the use of the session, it has to be closed, to release all the resources such as associated object and JDBC connection.
- sessionFactory.openSession().

⇨ Hibernate Properties:

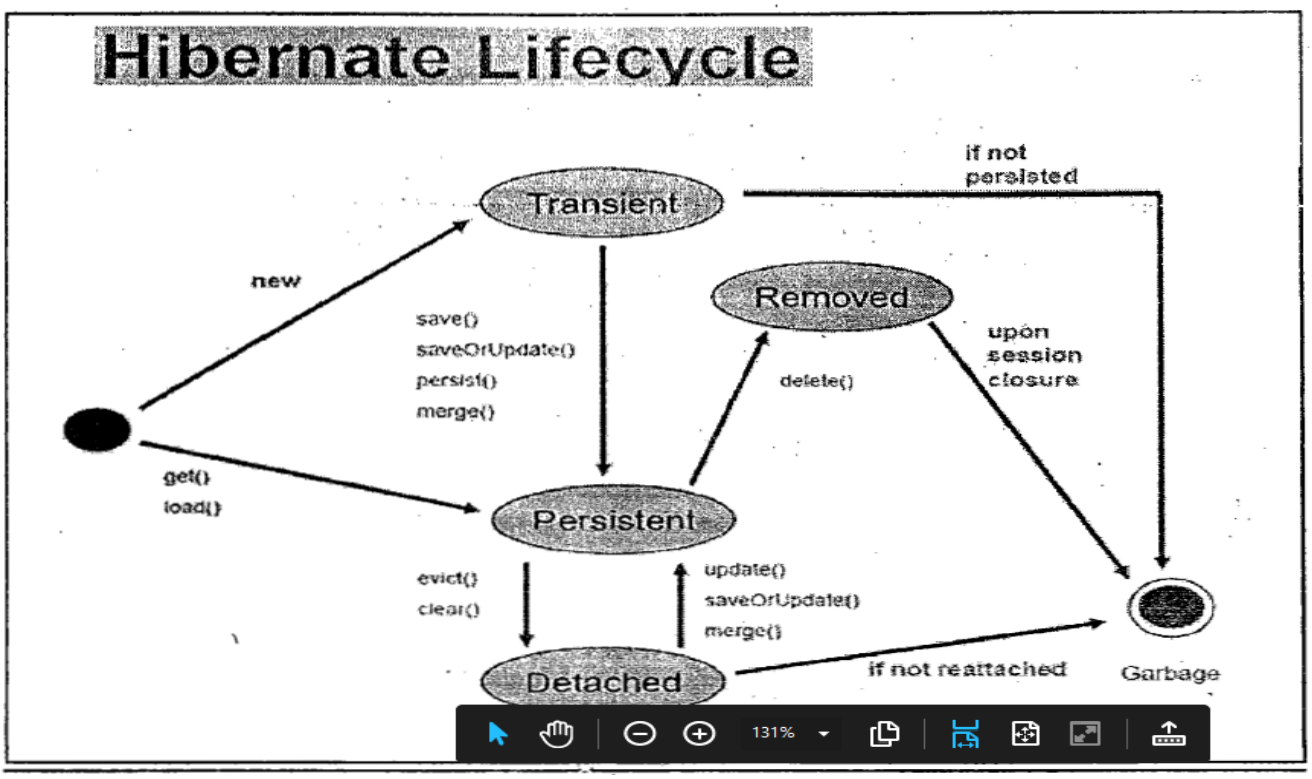
1. “show-sql” – If this value is true then we can view all the hibernate generated sql queries in the console.
2. “hbm2ddl.auto”: It has two values
   - Create – If table exists then drop and create new else created new.
   - Update – If table present then update else create.
   - Validate – This is default If table not present then throw exception else perform operation.
   - Create-drop – Drop table if present and then create and before closing connection drop the table.

⇨ What is different between load() and get() methods?

- If the given id Object does not exist in the database then **load()** method throws an ObjectNotFoundExeption but **get()** return null.
- Method load() loads the data lazly i.e on demand from database. And it return Proxy object (Pseudo object which contains only Id). However **get()** immediately loads the data from database so it is early loading.
- Eg: Account account = (Account)session.load(Account.class, 1001);

⇨ Persistence Object Life Cyle:

- **Transient state**
  An object is in T**ransient** state if it has just been instantiated using the new operator, and it is not associated with a **Hibernate** Session. It has no persistent representation in the database and no identifier value has been assigned.
  Person person = new Person();
  person.setName("Foobar");
  // person is in a transient state
- **Persistence state**
  And object is said to be in persistence state when it is associated with session as well object present in database.
  Long id = (Long) session.save(person);
  // person is now in a persistent state
- **Detached state**
  And object is said to be detached state when the object is not associated with session but present in database.



⇨ What is Transaction?

- Transaction used by the application to specify atomic unit of work (Transaction management).
- Using Session object, we can create Transaction object in two ways.
    1. Transaction transaction = session.getTransaction();
    2. Transaction trans = session.beginTransaction();
- Transaction object is unique per session object.
- Transaction interface defines following methods to deal with Transaction:
    1. Begin()
    2. Commit()
    3. Rollback()
- Default auto commit value is false in Hibernate.
- Default auto commit value is true in JDBC.
- In Hibernate even to execute one DML operation, we need to implement Transaction.
- Hibernate supports:
    1. JDBC Transaction
    2. JTA Transaction
    3. Spring Transaction.

⇨ Session Interface methods:

❖ **save()**

Public serializable save(Object obj) throws HibernateException

Save method stores and object into database. i.e it inserts an object into the database.

❖ **saveOrUpdate()**

public void saveOrUpdate(Object obj) throws HibernateException

This method can be used to either insert or update based upon existence of record. Now clearly saveOrUpdate() is more flexible in terms of use but it involves an extra processing to find out whether record already exists in table or not.

❖ **update(Object obj)**

Public void update(Object obj) throws HibernateException

Update() method is used for updating the object identifier. If identifier is missing or doesn't exits it will return exception.

- **merge()**
  Suppose we create a session and load an object. Now object is in session cache. If we close the session at this point and we edit state of object and tried to save using update() method it will throw exception. To make object persistence we need to open another session. Now we load same object again in current session. So if we want to update present object with previous object changes we have to use merge() method. Merge method will merge changes of both states of object and will save in database.
- **persist()**
  This method is similar to save but have following differences:
  - First difference between save and persist is its return type. Similar to save method persist also INSERT records into database but return type of persist() is void while return type of save is serializable object.
  - Another difference between persist and save is that both methods make a transient instance persistence. However, persist() method doesn't guarantee that identifier value will be assigned to the persistent instance immediately, the assignment might happen at flush time.
  - Persist() method guarantees that it will not execute an Insert statement if it is called outside of transaction boundaries. Save() method doesn't guarantees the same, it performs insert immediately, no matter if control is inside or outside of a transaction.

⇨ Hibernate Id Generator algorithms:

1. Sequence
2. Assigned
3. Increment
4. Hilo
5. Seqhilo
6. Identity
7. Native
8. Uuid
9. Guid
10. Select
11. Foregn

Apart from this we can define user define id generators as well.

⇨ JPA supported Identifiers:

- The @Id annotation lets us define which property is identifier of our entity bean.
- This property can be set by the application itself or generated by Hibernate. We can define Identifier Generation strategy using @GeneratedValue annotation.
- The @GeneratedValue annotation is used to specify the primary key generation strategy to use. If the strategy is not specified by default AUTO will be used.
- Different Generation Type strategies:
  1. AUTO (Used identity column of sequence depends on db.)
  2. TABLE (hilo algorithm)
  3. IDENTITY (Identity column)
  4. SEQUENCE
- @GeneratedValue annotation will allow the two attributes -**strategy & generator.**
- Example:
  @Entity
  @Table(name="EMPLOYEE")
  public class Employee{

```
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name=”eno”)
private int empNo;
@Column(name=”eno”, length=12)
private String name;
…
}
```

Note- Identity not supported by ORACLE but Supported by MySql.

⇨ Composite Id:

Process of defining primary key on more than one column is called composite id. Usage is combination of values should not be repeated.

⇨ Component Mapping/ Composition Mapping in Hibernate:

In Component mapping we will map the dependent object as a component. An component is an object that is stored as an value rather than entity reference. It mainly used if dependent object doesn't have primary key. It is used in case of composition (HAS-A) relation.

- **Using XML**

  Address.java

```
public class Address{
private String city;
private String pinCode;
….}
```

  Employee.java

```
public class Employee{
private int id;
private String name;
private Address address;
…}
```

  employee.hbm.xml

```
<class name=”com.test.domian.Employee” table=”EMPLOYEE”>
<id name=”id”>
<generator class=”increment”></generator>
</id>
<property name=”name”></property>
<component name=”address” class =”com.test.domain.Address”>
<property name=”city”/>
<property name=”pinCode”/>
</component >
</class>
```

  Note: **In this component all the data means Employee class and Address class both will store in Employee table. Because both are strongly related.**

- **Uning Annotation**

  Student.java

```
@Entity
@Table(name = " STUDENT")
public class Student {
@Id
```

```
@GeneratedValue
@Column(name = "STUDENT_ID")
private long studentId;

@Column(name = "STUDENT_NAME", nullable = false, length = 100)
private String studentName;

@Embedded
private Address studentAddress;
…}
```

Address.java

```
@Embeddable
public class Address {

    @Column(name = "ADDRESS_STREET", nullable = false, length=250)
    private String street;
    @Column(name = "ADDRESS_CITY", nullable = false, length=50)
    private String city;

    @Column(name = "ADDRESS_STATE", nullable = false, length=50)
    private String state;
    @Column(name = "ADDRESS_ZIPCODE", nullable = false, length=10)
    private String zipcode;
…}
```

⇨ Componet/ Composition example of project:

Contact.java

```
@Embedded
private UserDetails userDetails;
```

UserDetails.java

```
@Embeddable
public class UserDetails{
@Column (name=”CONTACT_SOEID”)
priavate String contactSoeId;
@Column (name=”CONTACT_NAME”)
priavate String contactName;
@Column (name=”BUSINESS_UNIT_CD”)
priavate String businessUnit;
…}
```

Note- All the UserDetails columns is present in Contact table only.

⇨ Inheritance Mapping

Hibernate supports three different approaches of Inheritance mapping:

1. **Table per class Hierarchy**
2. **Table per subclass Hierarchy**
3. **Table per concrete class Hierarchy**

⇨ Table per Class Hierarchy

- In this approach all the hibernate persistence classes of Inheritance will use single database table to store and manage their data.
- By this inheritance strategy we can map the whole hierarchy by single table only. Here, an extra column (also known as discriminator column) is created in the table to identified the class.

- <discriminator> tag is used to define the discriminator column.
- In this approach sub class will be configured by sub-class tag.
- **@Inheritance** – Defines the inheritance strategy to be used for an entity class hierarchy. It is specified on the entity/persistence class that is the root of the entity class hierarchy.
- **@DiscriminatorColumn** – Is used to define the discriminator column for the SINGLE_TABLE inheritance mapping strategies.
- @DiscriminatorValue- Is used to specify the value of the discriminator column for entities of the given type. The Discriminator value annotation can only be specified on a concrete entity class. If DiscriminatorValue annotation is not specified and a discriminator column is used then a provider specific function will be used to generate a value representing the entity type. If Discriminator is String the Discriminator value default is Entity name.
- Example using XML:

Person Details.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.nareshit.pojo">
<class name="Person" table="Person_Details" discriminator-value="person">
<id name="personId" column="Person_ID">
<generator class="increment" / ></id>
<discriminator column="usertype" type="string" />
<property name="firstName" length="15"/>
<property name="lastName" length="15"/>
<subclass name="Employee" extends="Person" discriminator-value="emp">
<property name="salary" length="10"/>
<property name="deptNo" length="10"/>
<property name="deptName" length="20"/>
<property name="joiningDate" type="java.util.Date" />
</subclass>
</class>
</hibernate-mapping>
```

- Using Annotation:

Person.java

```
package com.nareshit.pojo;
import javax.persistence.*;
import org.hibernate.annotations.GenericGenerator;
@Entity
@Table(name="Person_Details")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="usertype", discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue(value="Person")
public class Person {
        @Id
        @GeneratedValue(generator="myGenerator")
        @GenericGenerator(name="myGenerator",strategy="increment")
    private int personId;
     @Column(name="firtName",length=20)
    private String firstName;
  @Column(name="lastName",length=20)
  private String lastName;
  ...
```

Employee.java

```
package com.nareshit.pojo;
import java.util.Date;
import javax.persistence.*;
@Entity
@Table(name="Person_Details")
@DiscriminatorValue("emp")
public class Employee extends Person {
@Column(name="salary")
private double salary;
@Column(name="joiningDate")
private Date joiningDate;
@Column(name="deptNo")
private int deptNo;
@Column(name="deptName")
private String deptName;
...
```

PERSON_DETAILS

Table :

PERSON_ID	USERTYPE	FIRSTNAME	LASTNAME	SALARY	DEPTNO	DEPTNAME	JOININGDATE
1	emp	ramu	A	12000	12	Sales	12-DEC-12 12.00.00.000000 AM

⇨ Table per sub-class Hierarchy:

- In this approach every class of Inheritance hierarchy contains Its own database table. But the tables of sub-class will maintain one-to one relationship with table of super class.
- To configure super class, we can use <class> tag and to configure sub class we can use <joined-subclass> tags in Hibernate mapping file.
- The <joined-subclass> tag is used to map the sub class with super class, Using the primary key and foreign key relation.
- Using XML

PersonDetails.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.nareshit.pojo">
<class name="Person" table="Person_Details">
<id name="personId">
<generator class="increment"/>
</id>
<property name="firstName" length="15"/>
<property name="lastName" length="15"/>
<joined-subclass name="Employee"
table="Employee_Details">
<key column="empNo"/>
<property name="deptNo" length="10"/>
<property name="deptName" length="15"/>
<property name="salary" length="15"/>
<property name="joiningDate" />
</joined-subclass>
<joined-subclass name="Customer"
 table="Customer_Details">
<key column="customerId" />
<property name="orderName" length="15"/>
<property name="orderDate"  length="15"/>
</joined-subclass>
</class>
</hibernate-mapping>
```

- Using Annotation:
  Person.java

```
@Entity
@Table(name="person_details")
@Inheritance(strategy=InheritanceType.JOINED)
public class Person {
        @Id
        @GenericGenerator(name="myGenerator",strategy="increment")
        @GeneratedValue(generator="myGenerator")
        private Integer personId;
        private String firstName,lastName;
        public Integer getPersonId() {
                return personId;
        }
        public void setPersonId(Integer personId) {
                this.personId = personId;
        }
        public String getFirstName() {
                return firstName;
        }
        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }
        public String getLastName() {
                return lastName;
        }
        public void setLastName(String lastName) {
                this.lastName = lastName;
        }}
```

```
@Entity
@Table(name="person_details")
@Inheritance(strategy=InheritanceType.JOINED)
public class Person {
	@Id
	@GenericGenerator(name="myGenerator",strategy="increment")
	@GeneratedValue(generator="myGenerator")
	private Integer personId;
	private String firstName,lastName;
	public Integer getPersonId() {
		return personId;
	}
	public void setPersonId(Integer personId) {
		this.personId = personId;
	}
	public String getFirstName() {
		return firstName;
	}
	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}
	public String getLastName() {
		return lastName;
	}
	public void setLastName(String lastName) {
		this.lastName = lastName;
	}}
..}
```

Customer.java

```
package com.nareshit.pojo;
import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;
@Entity
@Table(name="customer_details")
@PrimaryKeyJoinColumn(name="customerId")
public class Customer extends Person {
	private String orderName;
	private Date orderDate;
...}
```

- Both Employee and Customer class are child of Person thus while specifying the mapping we used @PrimaryKeyJoinColumn to map it to parent table.
- **@PrimarykeyJoinColumn** – The Annotation specifies a primary key column that is used as a foreign key to join another table.

⇨ Table per Concrete class:

- Table per concrete class one per Pojo basis.
- In this case of Table per Concrete class there will be multiple tables in the database having no relations to each other. There are two ways to map the table with table per concreate class strategy.
  1. By union-subclass element
  2. By self-creating the table for each class.

⇨ Project example of Inheritance Mapping:

@Entity

```
@Table(name=”APPROVAL_DETAIL”)
@Audited
@AuditTable(value=”H$APPROVAL_DETAIL”)
@Inheritance(strategy= Inheritance.SINGLR_TABLE)
@DiscriminatorColumn(name=”APPROVAL_TYPE”, discriminatorType=DiscriminatorType.STRING)
@DiscriminatorOption(force=true)
public class ApprovalDetails extends BaseEntity{
@Id
@GeneratedValue(generator=”approvalDetailIdGenerator”, strategy=GenerationType.SEQUENCE)
@SequenceGenerator(name=”approvalDetailIdGenerator”, sequenceName=”APPROVAL_DETIAL_ID_SEQ”,
allocationSize=1)
private Long approvalDetailId;
@OneToOne(cascade = CascadeType.REFRESH)
@JoinColumn(name=”APPROVAL_CONTACT_ID”, referenceColumnName=”CONTACT_ID”, nullable=false)
Contact contact;
… }
```

ExtensionApproval.java

```
@Entity
@Audited
@AuditTable(value=“H$APPROVAL_DETAIL”)
@DiscriminatorValue(“EXTENSION”)
public class ExtensionApproval extends ApprovalDetails{
@Column(name=”REVIEW_ID”)
private Long reviewId;
@ManyToOne(fetch=FetchType.LAZY)
@JoinColumn(name=”EXTENSION_ID”,referencedColumnName=”EXTENSION_ID”)
private Extension extension;
…}
```

ReviewApproval.java

```
@Entity
@Audited
@AuditTable(value=“H$APPROVAL_DETAIL”)
@DiscriminatorValue(“REVIEW”)
public class ReviewApproval extends ApprovalDetails{
@ManyToOne(fetch=FetchType.LAZY)
@JoinColumn(name=”REVIEW_ID”, referencedColumnName=”REVIEW_ID”)
AbstractReview review;
…}
```

⇨ Association Mappings:

1. One-to-One
2. One-to-Many
3. Many-to-one
4. Many-to-Many

p.k ---> p.k (one-to-one)

f.k(unique) ---> p.k (one-to-one)

p.k ----> f.k (one-to-many)

f.k-----> p.k (many-to-one)

table1---> linktable --->table2(many-to-many)

⇨ One-To-One:

In this association one entity object exactly mapped with or associated with one object of another entity.
Example- Association is the Relationship between an Address & Person. And contact and ApprovalDetail.
In Hibernate one-to-one relationship between entities can be created by 2 different techniques.

1. Using shared primary key
   In this technique hibernate will ensure that it will use a common primary key value in both the table. This way primary key of entity PERSON can safely be assumed the primary key of entity Address also. The Example demonstrated shared primary key.
2. Using Foreign Key Association:
   In this association a foreign key column is created in the owner entity.
   For example – If we make PERSON as owner entity then a extra column ADDRESS_ID will be created in PERSON table. This column will store the foreign key for ADDRESS table.

One-to-One (pk-pk) UniDirectionalXmlMappingExample

SQL Script:

SQL>create table person (personId number(10,0) not null, name varchar2(12 char), age number(10,0), email varchar2(15 char), primary key (personId));

SQ>create table address (personId number(10,0) not null, h_No varchar2(20 char), city varchar2(20 char), state varchar2(20 char), country varchar2(20 char), primary key (personId));

Person.java

```
package com.nareshit.pojo;
public class Person {
private int personId,age;
private String name,email;
...}
```

Address.java

```
package com.nareshit.pojo;
public class Address {
private int personId;
private String h_No,city,state,country;
private Person person;
...}
```

Person.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Person" table="person">
<id name="personId">
<generator class="increment"/>
</id>
<property name="name" length="12"/>
<property name="age" length="3"/>
<property name="email" length="15"/>
</class>
</hibernate-mapping>
```

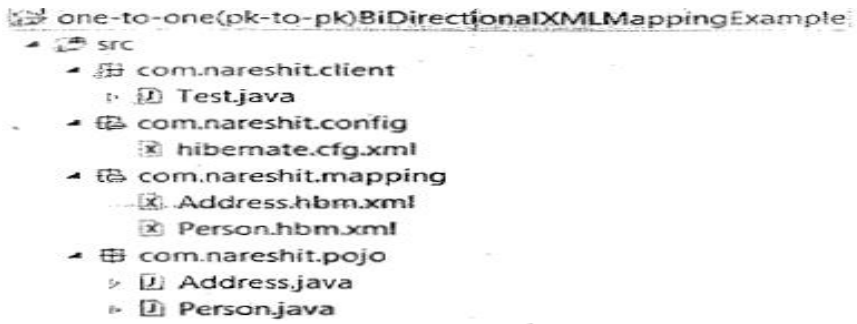
Address.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Address" table="address">
<id name="personId">
<generator class="foreign">
<param name="property">person</param>
</generator></id>
<property name="h_No" length="20"/>
<property name="city" length="20"/>
<property name="state" length="20"/>
<property name="country" length="20"/>
<one-to-one name="person" class="com.nareshit.pojo.Person"/>
</class>
</hibernate-mapping>
```

Note:- In the above example code Address object identity value is nothing but associated person class object identity value. For this we need to use foreign algorthim.

Foreign alg collects identity value from associated object and makes it has the identity value of current object. only foreign alg is capable of Collecting the identity value from associated object and assigns identity value of current object.

Example 2: One –to –One (pk-to-pk) BiDirectional XMLMappingExample



Person.java

```
package com.nareshit.pojo;
public class  Person {
private int  personId,age;
private String name,email;
private Address address;

...}
```

Address.java

```
package com.nareshit.pojo;
public class  Address {
private int  personId;
private String h_No,city,state,country;
private Person person;
...}
```

Person.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Person" table="person">
<id name="personId">
<generator class="increment"/>
</id>
<property name="name" length="12"/>
<property name="age" length="3"/>
<property name="email" length="15"/>
<one-to-one name="address"
class="com.nareshit.pojo.Address" />
</class>
</hibernate-mapping>
```

Address.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Address" table="address">
<id name="personId">
<generator class="foreign">
<param name="property">person</param>
</generator>
```

```
</id>
<property name="h_No" length="20"/>
<property name="city" length="20"/>
<property name="state" length="20"/>
<property name="country" length="20"/>
<one-to-one name="person" class="com.nareshit.pojo.Person" />
</class>
</hibernate-mapping>
```

Example 3: One –to –One (pk - to-pk) Bidirectional Annotation Mapping Example

Person.java

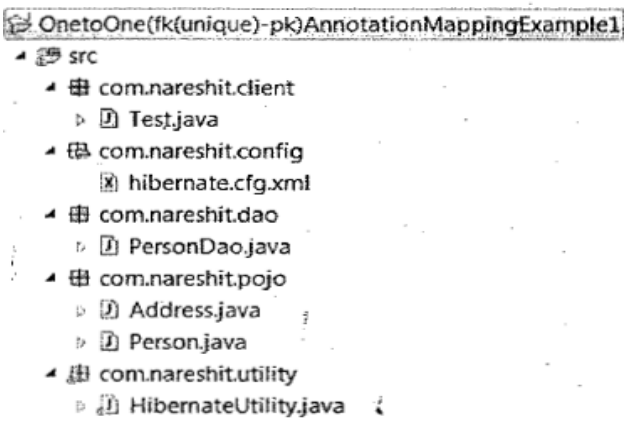
```
package com.nareshit.pojo;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;
import org.hibernate.annotations.GenericGenerator;
@Entity
@Table(name="person")
public class Person {
        @Id
        @GeneratedValue(generator="myGenerator")
        @GenericGenerator(name = "myGenerator",          strategy = "increment")
         @Column(length=12)
private int personId;
private int age;
@Column(length=20)
private String name,email;
@OneToOne(cascade=CascadeType.ALL)
@PrimaryKeyJoinColumn
private Address address;                    ...}
```

Address.java

```
package com.nareshit.pojo;
import javax.persistence.*;
import org.hibernate.annotations.GenericGenerator;
import org.hibernate.annotations.Parameter;
@Entity
@Table(name="Address")
public class Address {
        @Id
        @GenericGenerator(name = "myGenerator",
        strategy = "foreign",
        parameters=@Parameter(name = "property",
        value = "person"))
        @GeneratedValue(generator="myGenerator")
private int personId;
        @Column(length=20)
private String h_No,city,state,country;
@OneToOne(mappedBy = "address",cascade=CascadeType.ALL)
private Person person;                                                    ...}
```

One-to-One Using foreign key association :-

1. In this association, a foreign key column is created in the owner entity. For example, **if we make** PERSON as owner entity, then a extra column "ADDRESS_ID" will be created in PERSON table. This column will store the foreign key for ADDRESS table. The relational model is shown below:



Person.java

```
package com.nareshit.pojo;
import javax.persistence.*;
import org.hibernate.annotations.GenericGenerator;
@Entity
@Table(name="person")
public class Person {
        @Id
        @GenericGenerator(name="myGenerator",strategy="increment")
        @GeneratedValue(generator="myGenerator")
        private int personId;
        @Column(length=12,name="name")

        private String personName;
        private byte age;
        @Column(length=15,name="email")
        private String email;
@OneToOne(cascade=CascadeType.ALL)
@JoinColumn(name="address_Id",unique=true,nullable=false)
private Address address;
public Person(){
}                                                    ...}
```

Address.java

```
package com.nareshit.pojo;
import javax.persistence.*;
@Entity
@Table(name="Address")
public class Address {
@Id
private int addressId;
@Column(length=15,name="h_No")
private String h_No;
@Column(length=20,name="city")
private String city;
@Column(length=20,name="state")
private String state;
@Column(length=20,name="country")
private String country;                              ...}
```

In one –to-one (fk- to -pk) association, refer the Address entity in Person class as follows:

```
1 @OneToOne
2 @JoinColumn(name="address_Id")
3 privateAddress address;
```

If no **@JoinColumn** is declared on the owner (Person) entity, the defaults apply. A **join column(s)** will be created in the owner **entity** table and its name will be the concatenation of **the name of the** relationship in the owner side, _ (underscore), and the name of the primary key **column(s) in the** owned side.

⇨ Project example of One-To-One:

@Entity
@Table(name=”APPROVAL_DETAIL”)
@Audited
@AuditTable(value=”H$APPROVAL_DETAIL”)
@Inheritance(strategy= Inheritance.SINGLR_TABLE)
@DiscriminatorColumn(name=”APPROVAL_TYPE”, discriminatorType=DiscriminatorType.STRING)
@DiscriminatorOption(force=true)
public class ApprovalDetails extends BaseEntity{
@Id
@GeneratedValue(generator=”approvalDetailIdGenerator”, strategy=GenerationType.SEQUENCE)
@SequenceGenerator(name=”approvalDetailIdGenerator”, sequenceName=”APPROVAL_DETIAL_ID_SEQ”, allocationSize=1)
private Long approvalDetailId;
@OneToOne(cascade = CascadeType.REFRESH)
@JoinColumn(name=”APPROVAL_CONTACT_ID”, referenceColumnName=”CONTACT_ID”, nullable=false)
Contact contact;
… }

Note – No related code in Contact Domain.

⇨ One-To-Many:

In this association One parent object is associated with multiple objects. One Department has multiple employees. One Memo object has multiple Contact objects.

One-to-Many relations can be developed in four ways: - Set, List, Bag, Array, Map.

Bag – When we use List as Child object holder in parent class and as we know list holds insertion order so to but database is not aware of this insertion order so to maintain insertion order with respect to List in mapping file, we use <bag> and in database there should be a separate column to maintain insertion order.

Example:

Department.java

```
public class Department{
private int deptNo;
private String name;
private Set<Employee> empoyees;
…}
```

Employee.java

Public class **Employee**{
Private empName;
private String name;
private doulbe salary;
…}

Mapping file:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="net.codejava.hibernate">

<class name="Department" table="DEPARTMENT">
<id name=" deptNo" column=" DEPT_NO">
<generator class="native"/>
</id>
<property name="name" column="NAME" />
<set name=" empoyees" inverse="true" cascade="all">
<key column=" DEPT_NO" not-null="true" />
<one-to-many class="Employee"/>
</set>
</class>
</hibernate-mapping>
```

Note- In database level we will store parent primary key in child table and with respect to parent_id there must be multiple object of child.

```
create database stockdb;
use stockdb;

CREATE TABLE `category` (
  `category_id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(45) NOT NULL,
  PRIMARY KEY (`category_id`)
);

CREATE TABLE `product` (
  `product_id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(45) NOT NULL,
  `description` varchar(512) NOT NULL,
  `price` float NOT NULL,
  `category_id` int(11) NOT NULL,
  PRIMARY KEY (`product_id`),
  KEY `fk_category` (`category_id`),
  CONSTRAINT `fk_category` FOREIGN KEY (`category_id`) REFERENCES `category` (`category_id`)
);
```

Note – Here Category is parent and Product is Child.

⇨ Many-To-One:

This type of association relates many objects of an entity to one Object of another Entity, means One child must have one parent. In this relation we take a parent object in child class. In this association table structure will be same as One-To-Many. In normal programming we generally implement **One-To-Many bidirectional** which is combination of **One-To-Many** & Many-To-One so will directly see the example of **Bidirectional**. In this case in Employee mapping file, we just need to write:

```
<many-to-one name="dept"
class="com.nareshit.pojo.Department"
column="dept_no"/>
```

Note- The @ManyToOne annotation is used to create the many-to-one relationship between the Employee and Department entities. The cascade option is used to cascade the required operations to the associated entity. If the cascade option is set to CascadeType.ALL then all the operations will be cascaded. For Instance, when we save a employee obejct then associated Department object will also save automatically.

⇨ One-To-Many Bidirectional:

In Bidirectional relationship one of the sides (and only one) has to be the owner. The owner is responsible for the association column update. To declare a side as not responsible for the relationship the attribute **mappedBy** is used. mappedBy refers to the property name of the association on the owner side. As we can see we don't have to declare the join since it has already been declared on the owner side.

Inverse:

Inverse keyword is responsible for managing the insert /update of the foreign key column. An inverse keyword has the boolean value "true/false". The default value of this keyword is false.

If inverse keyword value is false then parent class is responsible for saving/updating the child and its relationship.

If inverse keyword is true then then an associated subclass is responsible for saving/updating itself.

Note- An inverse keyword is always used with the **One-To-Many** & **Many-To-One.**

Example:

ReviewMemo.java

```
@Entity
@Table(name="MEMO")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="MEMO_TYPE")
@Audited
@AuditTable(value="H$Memo")
public class ReviewMemo extends BaseEntity{
…
@OneToMany(fetch= FetchType.EAGER, mappedBy="reviewMemo", cascade=CascadeType.All,
orphanRemovel=true)
private Set<Contact> contacts;
…}
```

Then In Contact domain:

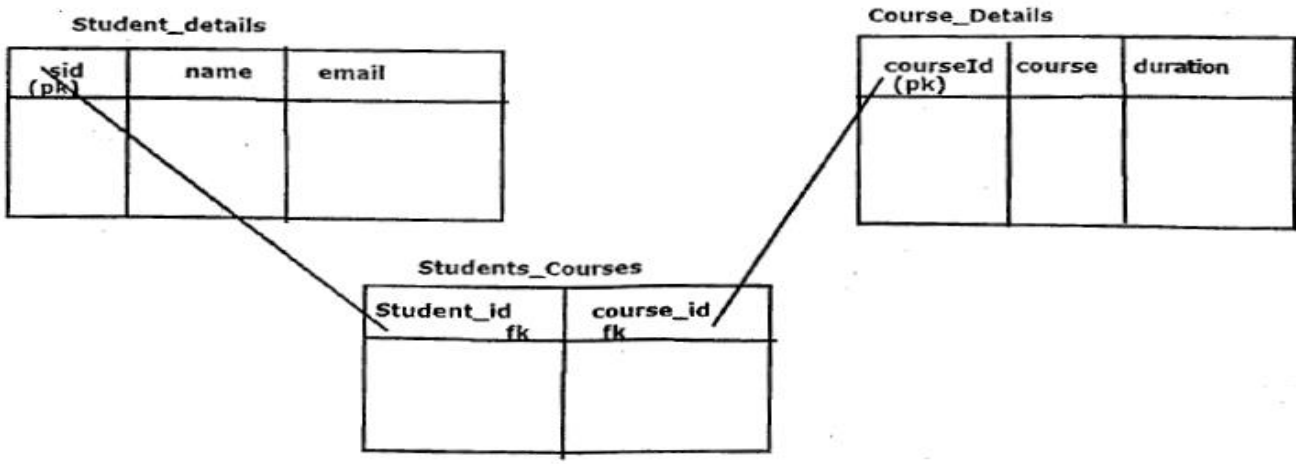
```
@ManyToOne(fetch=FetchType.LAZY)
@JoinColumn(name="MEMO_ID", referenceColumnName="MEMO_ID", updatable=false)
ReviewMemo reviewMemo;
```

⇨ Many-To-Many:

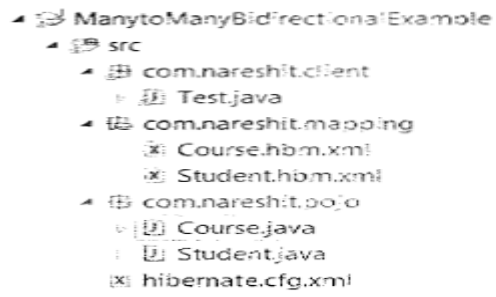
In this association many objects of one Entity are associated with many object of another Entity.

Student_details entity maps to multiple objects of coures_details.

Database not supports this kind of relationship so we have to use a separate table to maintain relationships.



Many-To-Many Mapping Example (XML Mapping)



Student.java

```
package com.nareshit.pojo;

import java.util.Set;

public class  Student {
private int  sid;
private String name,email;
private Set<Course> courses;  ...}
```

Course.java

```
package com.nareshit.pojo;

import java.util.Set;

public class  Course {
private int  courseId;
private String courseName,duration;
private Set<Student> students;          ...}
```

Student.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Student"
table="student_details">
<id name="sid">
<generator class="assigned"/>
</id>
<property name="name" length="12"/>
<property name="email" length="20"/>
<set name="courses" table="Students_Courses" cascade="all">
<key column="student_id"/>
<many-to-many class="com.nareshit.pojo.Course"
column="course_id"/>
</set>
</class>
</hibernate-mapping>
```

Course.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Course"
table="course_details">
<id name="courseId">
<generator class="assigned"/>
</id>
```

```
<property name="courseName" length="15"/>
<property name="duration" length="15"/>
<set name="students" table="Students_Courses" cascade="all">
<key column="course_id"/>
<many-to-many class="com.nareshit.pojo.Student"
column="student_id"/>
</set>
</class>
</hibernate-mapping>
```

⇨ Project example of Many-To-Many:

```
@Enity
@Table(name=”REGION”, schama=”REFDATA”)
@NamedQueries({
@NamedQuery(name=”Region.getAllRegion”, query=”select r from PACRegion r where r.committeeId=
:committeeId”),
@NamedQuery(name=”Region.getRegionById”, query=”select r from PACRegion r where r.regionId =
:regionid”)
})
@Cache(usage=CacheConcurrencyStrategy.READ_ONLY)
public class PACRegion extends BaseEntity{
@Id
@Column(name=”REGION_ID”)
private Long regionId;
@Column(name=”SHORT_NAME”)
private String shortName;
@Column(name=”LONG_NAME”)
private String longName;
@Column(name=”REGION_CD”)
private String regionCode;
@Column(name=”COMMITTEE_ID”)
private Long committeeId;

@ManyToMany(fetch= FetchType.EAGER, cascade= CascadeType.ALL)
@JoinTable(name=”COMMITTEE_REGION_MAP”, joinColumns=@JoinColumn(name=”REGION_ID”),
inverseJoinColumns=@JoinColumn(name=”COMMITTEE_ID”))
private Set<GoverningCommittee> governingCommittees;

…
@Override
public int hashCode(){
Final int prime = 31;
int result = 1;
result = prime * result + ((shortName == null) ? 0 : shortName.hashCode());
retrun result;}
@Override
public boolean equals(Object obj){
if(this == obj)
retun true;
if(obj == null)
return flase;
if(getClass() != obj.getClass())
return false;
PACRegion other = (PACRegion) obj;
If(shortname == null){
```

```
If(other.shortName != null){
return false;}
else if (!shortName.equals(other.shortName))
return false;
return true;
}
}
```

Public class **GoverningCommittee** extends Committee{
@OneToOne
@JoinColumn(name=”PARENT_ID”, referenceColumnName=”COMMITTEE_ID”)
Private ApprovalCommittee approval committee;

@ManyToMany(fetch= FetchType.EAGER, mappedBy=” **governingCommittees**”)
Private Set<PACRegion> regions;
}
Tables:
Table-**REGION**.

REGION_ID	SHORT_NAME	LONG_NAME
1	APAC	Asia Pacific

Table-**COMMITTEE**

COMMITTEE_ID	SHORT_NAME	LONG_NAME	REGION_ID	COMMITTEE_TYPE
1	NPAC_JAPAN	Japan NPAC	1	GOVERNING
2	NPAC_AMERICAS	Americas NPAC	2	GOVERNING
				JOINT_REVIEW

Table- **COMMITTEE_REGION_MAP**

COMMITTEE_REGION_MAP_ID	REGION_ID	COMMITTEE_ID
1	1	2

Note- To establish One-To-Many Relationship eg – ProcessFeedbac can have multiple Contact objects and if we don’t want to touch Contact table then in that case we simply create one mapping table as “PROCESS_CONTACT_MAP”.
Table - PROCESS_FEEDBACK

PROCESS_FEEDBACK_ID	COMMENTS
1	
2	
3	

Table - PROCESS_CONTACT_MAP

PROCESS_FEEDBACK_ID	CONTACT_ID
1	5
2	

In **ProcessFeedback** domain class we need add below code:

```
@OneToMany(fetch=FetchType.LAZY)
@JoinTable(name=”PROCESS_CONTACT_MAP”, joinColumn=@JoinColumn(name=”PROCESS_FEEDBACK_ID”),
inserviceJoinColumn=@JoinColumn(name=”CONTACT_ID”))
private Set<Contact> contacts;
```

⇨ Hibernate Bulk Operations:

1. HQL (Hibernate Query Language)
2. Criteria API
3. Native SQL

⇨ HQL (Hibernate Query Language)

- HQL can performs updates, delete, selects, on multiple rows of data (multiple objects) at a time.
- We can perform both select and non-select operations on a database by using HQL.
- An Object-oriented form of SQL is called HQL.
- HQL syntax is very much similar to SQL syntax.
- HQL queries are performed by using entities and their properties.
- SQL queries are formed by using db tables and their columns.
- SQL is a database dependent and HQL is a database Independent.
- Example-
  SQL – Select * from Employee;
  HQL-select e from com.test.pojo.Employee as e;/ from com.test.pojo.Employee/ from com.test.pojo.Employee e
- HQL & SQL keyword are not case sensitive.
- **Query query = session.createQuery(hqlQuery);**
- **Query.list();**
- We can use HQL named query – **session.getNamedQuery(String namedQueryName**); this method returns object of Query.

⇨ Query:

- Query is an Interface and its implemented class is QueryImpl
- Query is an Object-oriented representation of Hibernate Query.
- The object of Query can be obtained by **session.createQuery(Sting hql)** or **session.getNamedQuery(String namedQuery)** method.
- **public int executeUpdate()** is used to execute the update or delete query.
- **public List list()** method returns the results of select query as list.
- When HQL select query selects all the column values of database records then the generated list will be object of domain class.
  Example – List<Employee> result = query.list();
- If HQL select query is selecting specific column values of a table then the generated list data structure contains Java.lang.Object[] array.
  Example – **Query query = session.createQuery("select e.name,e.salary from Employee e");**
  List<Object[]> result = query.list()
- When we are preparing the HQL queries we can place 2 types of parameters
  1. Position parameter (?)
  2. NamedParameter(:example)
- Query query = session.createQuery("select e from Employee e where e.name like=? And e.salary>=?")
  q.setParameter(0, "ramu");
  q.setDouble(1, 1600);
- NamedParamenter – **Query query = session.createQuery("select e.empNo, e.name from Employee as e where e.name like :name1 and e.salary>=:salary1");**
  query.setParameter("name1","Ramu");
  query.setDouble("salary1", 7000.0);
  List<Object[]> list = query.list();
- We can use both named parameter and positional parameter with in the same query. But We should write positional parameters first then named parameter.

⇨ Fetching strategies:

- There are four fetching strategies:
  1. fetch- "join" - Disable the lazy loading, always load all the collections and entities.

2. fetch – “select” (@Fetch(FetchMode.SELECT)) (default) - Lazy load all the collections and entities
3. batch-size=”N” – Fetching up to ‘N’ collections or entities, Not records.
4. fetch-“select” – Group its collection into a sub select statement.

⇨ Criteria API:

- Criteria API provides an object-oriented way to retrieve persistent Object.
- We can execute only select statements by using criteria we cannot execute Update, Delete statements, Using criteria.
- Criteria api also include aggregation methods.
- Criteria is an Interface present in org.hibernate package used to represent the query in object oriented manner.
- To get Criteria object we use following methods – session.createCriteria(Class entityClass);
  Example –
  Session session = sessionFactory.openSession();
  Criteria crit = session.createCriteria(Employee.class);
  Criterion nameCriterion = Restrictions.eq(“name”, “Vikash”);
  Crit.add(nameCriterion);
  List<Employee> result = crit.list();
- Projections – If we want to select particular properties (column) to perform aggregate functions use projections.
  Session session = sessionFactory.openSession();
  Criteria crit = session.createCriteria(Employee.class);
  Projection nameProjection = Projections.property(“name”);
  Projection salaryProjection = Projections.propery(“salary”);
  ProjectionList pList = Projections.projectionList();
  pList.add(nameProjection);
  pList.add(salaryProjection);
  crit.setProjection(pList);
  List<Object[]> result = crit.list();
- Query query = session.createSQLQuery(sqlQueryString);
- We can add entity in SQL query using **query.addEnity(Empoyee.class);**

⇨ Hibernate caching:

1. First level cache/ session level cache / Local cache
2. Session level cache/ session factory level cache/ Global cache.

⇨ First level cache/ Session level cache/ Local cache

- First level cache in Hibernate is enabled by default, its opens with session and closes with session. We cannot disable it forcefully. First level cache is associated with session object.
- First level cache is associated with session object and other session objects in application cannot see it.
- The scope of cache object is of session. Once session is closed cached objects are gone forever.
- First level cache is enabled by default and we cannot disable it.
- When we query an entity first time, it is retrieved from database and stored in first level cache associated with hibernate session.
- If we query same object again with same session object it will be loaded from cache and no sql query will be executed.
- The loaded entity can removed from session using **evict()** method. The next loading of this entity will again make a database call if it has been removed using evict() method.
- The whole session cache can be removed using **clear()** method. It will remove all the entities stored in cache.
- Its programmer responsibility to refresh the session time to time using **refresh()** method.

⇨ Session level cache/ session factory level cache/ Global cache.

This is sessionFactory level cache, first step to define this cache is to select the vendor general we use EHCache.

⇨ By default a transaction must be atomic, consistent, isolated and durable (ACID).

SPRING

⇨ What is Spring?

- Spring is light weight, open source, loosely coupled, dependency injection, aspect oriented java application framework.
- Spring is light weight reason spring framework size is very less and spring container can be activated as in memory containers without any physical server support.

- Spring is non-invasive framework, because the classes of spring app can be developed without extending or implantation spring API classes/interface.
- Spring container are light weight containers. Because they can be activated anywhere by just creating objects for certain predefined classes.
  Note – Servlet container and JSP container are heavyweight containers.
- Spring is loosely coupled because we can use one or another module of spring without having dependency with other module.

⇨ Pojo/Java Bean:

- Pojo – The class that can be executed in JDBC environment without taking support of third party libraries is called POJO classes.
- Java Bean – The class that follow some standards is called Java Bean
  1. Must be public class
  2. Must have private members variables
  3. Must have public getters/setters
  4. Must have 0-param constructor directly or indirectly
  5. Must implements Java.io.serizable

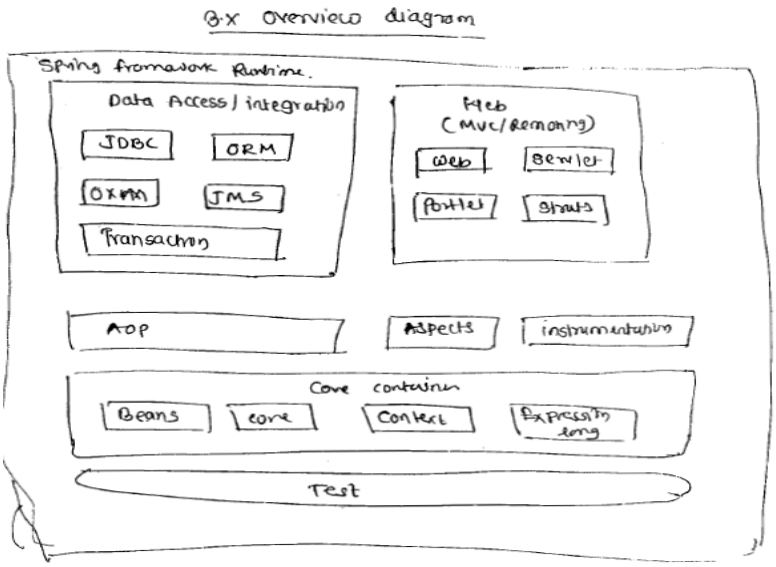
  Note – Every Java Bean is Pojo but every Pojo class need not to be Java Bean.

❖ If Bean class is managed by Spring container, then this bean is called as Spring Bean.

⇨ Spring modules:

1. Spring core
2. Spring AOP
3. Spring MVC
4. Spring J2EE (Service) / Context
5. Spring Data Access / DAO
6. Spring Test.

⇨ Spring 3.x Architecture:



SPRING CORE

⇨ Why Has-A relationship is better than Is-A relationship?

❖ **Reason – 1**

If we apply Is-A relationship between two classes then object of subclass can get functionality of an only one super class. It is not possible to get functionality for more than one super class. Because java allow only one class to extends in our child class.

- If we apply Has-A relationship then it is possible to get functionality of more than one class in our child class.

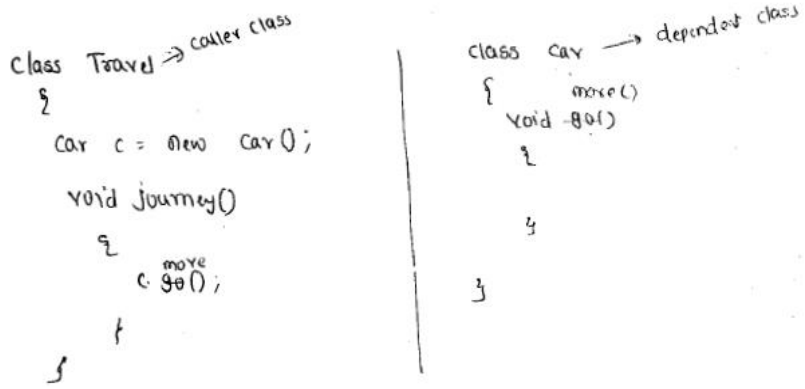
❖ **Reason – 1**

- If we apply Is-A relationship then all functionality of superclass will be inherited into subclass.
- If we want some functionality then remaining functionality should be made as a private. If another subclass wants functionality of super class, then it is not possible.
- If we have Has a relationship then we no need to make the functionality of a class as private. We can call any required functionality by using an object.
- In Is-A relationship when we apply testing on subclass then its superclass is also get tested and if any problem occurs in subclass testing will be fail for super class as well.

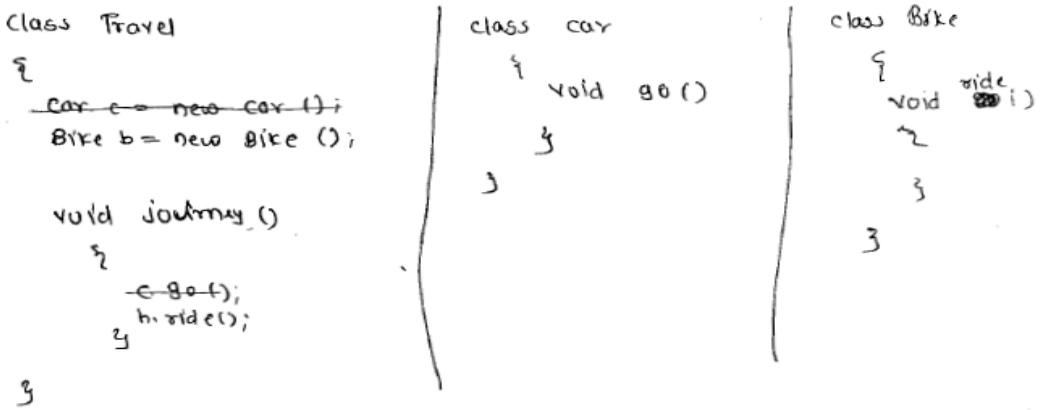
⇨ What is Tightly coupling?

Tightly coupling between two classes will occur in the following situation

- If method name is changed in the dependent class name, then we need to modify caller class also.



- If a caller class wants to change its dependency to another similar type of class then also we need to change or modify caller class.



In the above example when we want to change dependency from car class to Bike class then we have to change caller class called **Travel**. This is called Tightly coupling.

⇨ Loosely coupling:

- In order to overcome tight coupling between objects in spring framework we use dependency injection mechanism using POJO/POJI model. In the above example Traveler, Car are Tightly coupled. If we want to achieve loose coupling between the objects Traveler and Car we need to re-write the program like:

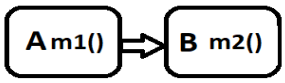
```
class Traveler
{
   Vehicle v;
   public void setV(Vehicle v)
   {
     this.v = v;
   }

   void startJourney()
   {
     v.move();
   }
}
```

```
Interface Vehicle
{
  void move();
}
```

```
class Bike implements Vehicle
{
    public void move()
    {
        // logic
    }
}
```

In the above example, Spring container will inject either Car object or Bike object into the Traveler by calling setter method, so if Car object is replaced with Bike, then no change are required in Traveler class. This means there is loosely coupling between Traveler and Vehicle object. Actually setter method will be activated by XML file or in case of Annotation when we autowired Vehicle property then Spring will automatically injects its implementation class object to the vehicle reference variable.

- **We can explain this Loosely coupling in another word:**



How **Class A** can communicate to **Class B** or how **Class A** can use method **m2()** present in **Class B**, for this Java provides two methods:

1. **Inheritance (Is-A)** – We can extends **Class B** in **Class A**.
2. **Composition (Has-A)** – We can create object of **Class B** in **Class A**.

In both the cases **Class A** is tightly coupled with **Class B**, i.e., Degree of dependency is high. If we modify or delete the **Class B** then there will be impact or just assume in project **Class B** has used multiple places around 100 places and if we delete or modify the **Class B** then there will be huge impact on project.

So, to overcome this problem Spring provides IOC (Inversion of control) container to achieve loosely coupling. So how can we do that? We do not need to inherit the Class B into **Class A** or create the object of **Class B** into **Class A**; we just need to tell **IOC** container hey **IOC** we need object of into **Class A** then **IOC** will create the object of **Class B** and Inject into **Class A**. This process is called as **Dependency Injection**.

⇨ How we are able to develop loosely coupled classed in Spring framework?

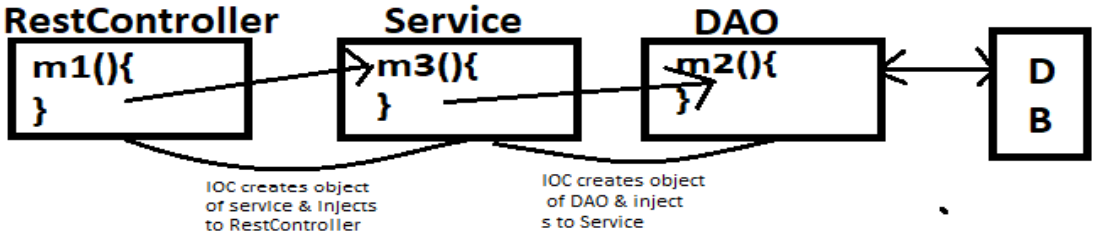
By using IOC container and Dependency Injection.

⇨ What is Spring Bean?

Normal java bean class managed by IOC container called Spring bean.

⇨ Dependency Injection:

When underlying server or container injects the value to the resources or dependent object into target object is called as Dependency Injection.

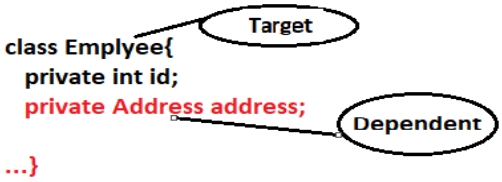


Note – Here all the classes should be Spring Bean.

- There are 4 types of Dependency Injection in Spring –
  1. Setter Injection
  2. Constructor Injection
  3. Interface Injection
  4. Lookup method/ method injection

⇨ Setter Injection:

- Injecting dependent bean into target bean using target bean setter method is called as Setter injection.
- Below is image of illustration of Target and dependent class:



- In Setter injection Target bean object will created first.
- Partial injection is possible in setter injection.
- In this injection Target method must contain setter method of dependent property.

⇨ Constructor Injection:

- Injecting dependent bean into target bean using target bean constructor is called as constructor injection.
- Dependent bean created first then target bean will be created.
- If we have only one constructor in the spring bean then writing @Autowired is optional.
- In xml we use <constructor-arg/> tag to inject the values in Target class.
- In Spring bean class if both constructor and setter injection applied for same property then constructor injection will be overridden by setter injection because constructor injection will happen at the object creation time and setter after object creation.
- For parameterizes construction injection we can use - **<constructor-arg value="myuserName" index="0" />** and **<constructor-arg value="mypassword" index="1" />**
- To inject object, we use ref - **<constructor-arg ref="sb" />**

⇨ Field Injection:

- In Spring based project if we use @Autowired annotation at variable level then IOC will perform field Injection.
- To perform field injection IOC will use Reflection API.
- As our variables are declared as private IOC can not access private variables directly from outside of class.
- To access private variable outside of class we can use Reflection API.

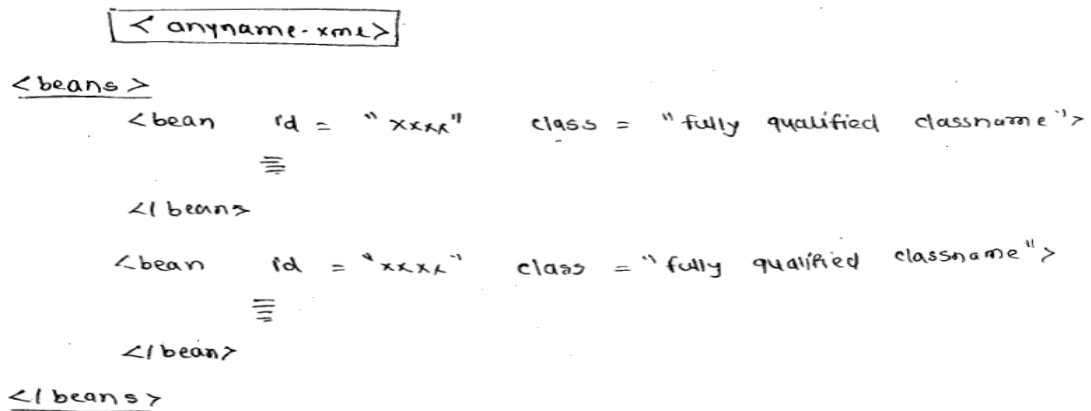
Note – We are going to Autowiring **ByType** mechanism then there is a chance of ambiguity problem to resolve this issue we can use **@Primary** annotation. And another solution we have that is @Qualifier in this method we pass component name from component. It works on ByName mechanism.
Eg –
@Autowired
@Qualifier(“petrolEngine”)
private IEngine eng;

⇨ Setter vs. Constructor Injection:

- Setter injection gets preference over constructor injection when both are specified
- Constructor injection cannot partially initialize values
- *Circular dependency can be achieved by setter injection*
- Security is lesser in setter injection as it can be overridden
- Constructor injection fully ensures dependency injection but setter injection does not
- Setter injection is more readable

⇨ Spring Configuration file:

- <any_name>.xml
- Only class (class may be concrete, abstract and final) is configured using spring xml file not Interface.
- In order to tell classes of our application to the Spring IOC container we configure our class in Spring configuration file. However, if we work with Spring with Annotation then We just need to use @ComponentScan annotation on the root package class or we can define base package with component.
  Eg – If our base package it com.learn and on top of this we have com.learn.controller and other then if we create a class eg- Application.java and on top of this Application class if we define @ComponetScane then Spring will automatically creates spring beans of all the classes which recedes under the base package and sub-package.
- We can specify base package name manually in different formats:
  @ComponentScan(“com.in28minutes.springboot”) /
  @ComponentScan({"com.in28minutes.springboot.basics.springbootin10steps","com.in28minutes.springboot.somethingelse"}) /
  @ComponentScan({"com.in28minutes.package1","com.in28minutes.package2"})
- XML based - <context:component-scan base-package="com.in28minutes" />
- <context:component-scan base-package="com.in28minutes.package1, com.in28minutes.package2" />
- In Web application we configure our servlet class in deployment descriptor file (web.xml). Similarly, in spring application we configure spring beans in a spring configuration file.
- A spring configuration file contains root element **<beans>** & each class is configured using **<bean>** tag.



- Using <bean> tag only classes can be configured not Interfaces.

- A class can be concreate, abstract or it may be final.
- We can configure same class for multiple times with different ids.

```
<bean id = "id1" class = "com.sathya.beans.Travel" />
<bean id = "id2" class = "com.sathya.beans.Travel" />
```

- Id should not be duplicate, Id should not contain special character it can have only alpha-numeric chars.
- In a bean class if we define a constructor for injecting dependency then we need to configure <constructor-arg>
- If a setter method is defined for injection, then we need to configure <property> tag.
- In <property> tab name attribute is mandatory but in <constructor-arg> tab name attribute is not allowed.
- A Spring bean can have more than one constructor, because a constructor can be overladed.
- If bean class contain many constructor then we need to configure bean class for multiple times in xml file.

  Eg –

```
Public class A
{
    Private int x;
    Private int y;
    Public A(int x)          // with 1 argument
    {
        this.x = x;
    }
    Public A (int x, int y) // with 2 argument
    {
        this.x = x;
        this.y = y;
    }
}
<beans>
    <bean id = "a1" class = "A">
        <constructor-arg value = "100" />
    </bean>
    <bean id = "a2" class = "A">
        <constructor-arg value = "50" />
        <constructor-arg value = "70" />
    </bean>
</beans>
```

- If dependency of bean class is reference type, then IOC container injects object.
- In the configuration file we use “ref” attribute to configure the referenced type of dependency.
- <property> tag has the below attributes:
  1. name
  2. value
  3. ref
- **name** attribute is mandatory and **value** & **ref** we can use only one at a time.
- We can configure ref as tag as well, when ref as tag then it should be either local of parent or bean has an attribute.
- If we configure local attribute with <ref> tag then spring container will search for dependent in same xml file.

- If we configure Parent attribute with <ref> tag then Spring Container checks for dependent bean in Parent xml.

⇨ Type of dependency in a bean:

- A bean can have a maximum of 3 types of dependencies:
  1. Primitive Type.
  2. Referenced Type.
  3. Collection Type.

⇨ Spring container:

- A container is class which provides runtime supports for other classes.
- In Spring framework, a spring IOC container means it is an object of implementation class of either BeanFactory Interface or ApplicationContext interface.
- ApplicationContext interface is a sub interface of a BeanFactory, So ApplicationContext container is better than BeanFactory.
- XMLBeanFactory is implementation class of BeanFactory interface.

```
BeanFactory factory = new XmlBeanFactory();
```

- ClassPathXmlApplicationContext is an implementation class of ApplicationContextInterface.
- While creating Spring IOC container object we need to pass spring configuration file as a parameter so spring container reads Bean definition from xml file.
- In Spring configuration file to configure collection type we use - <list>, <set>, <map>,<props>

⇨ Application context:

- ApplicationContext is sub-interface of BeanFactory.
- The 3 important implementation classes of ApplicationContext interface are:
  1. ClassPathXmlApplicationContext
  2. FileSystemApplicationContext
  3. XmlWebApplicationContext
- XmlWebApplicationContext will be used in Spring MVC applications.
- ClassPathXmlApplicationContext loads the Spring Configuration file from class path location.
- If a class which is loading the xml file and the xml file are in two different locations then we use FileSystemXMLApplicationContext for loading the xml file.

⇨ Circular Dependency issue:

- If A&B are two Beans where A depends on B and B depends on A and in the both Beans constructor injection is defined then circular dependency Problem occurs.
- We can solve this circular dependency by using setter injection.

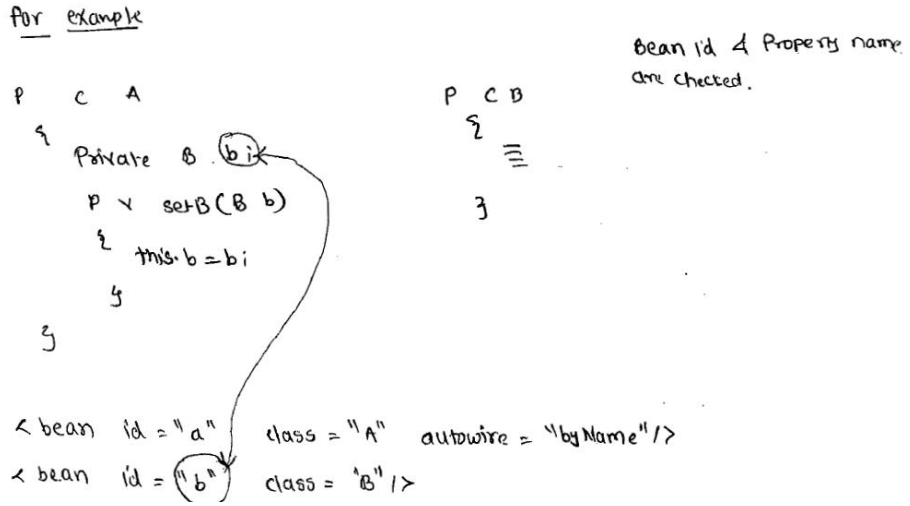
⇨ Bean Autowiring:

- The process of combining our bean class with spring container is called bean wring.
- For bean wiring we need to configure our beans with there dependencies in xml file.
- Bean Autowiring – It is dependency injection mechanism in which Spring container automatically injects the dependencies of dependent properties.
- In Bean Autowiring a spring container automatically injects the dependencies (Java class object and its properties) of the Spring Bean (Java class) without <property> or <constructor-arg> in spring configuration file.
- By default, Autowiring in bean is disabled if we want to enabled then we need to add autowire attribute in <bean> tag.
- The possible values of autowire attribute are:
  1. no (default)
  2. byName

3. byType
4. constructor

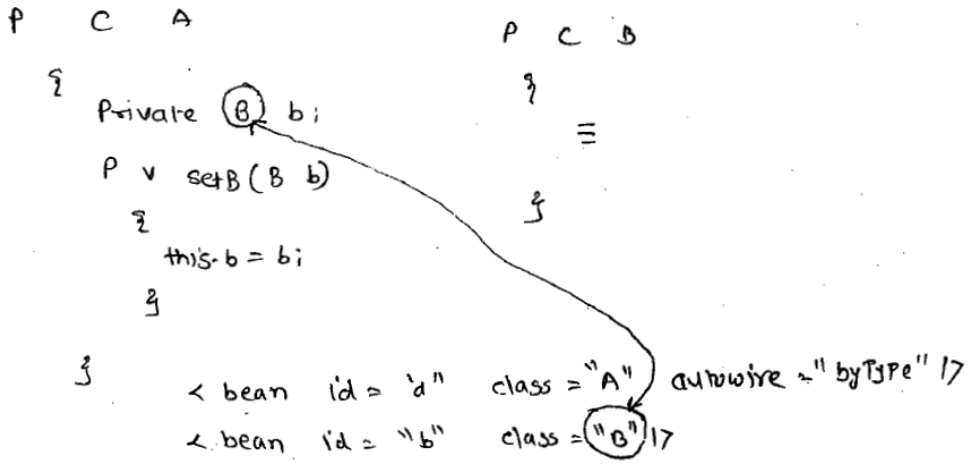
⇨ byName:

- If autowire strategies is byName then a Spring container verifies a bean id in xml is matched with property name or not if matched then container injects an object of bean class by calling the setter method.



⇨ byType:

- In this Autowiring strategi Spring container verify a bean class in xml is matched with property type or not if matched then container injects by calling setter method.



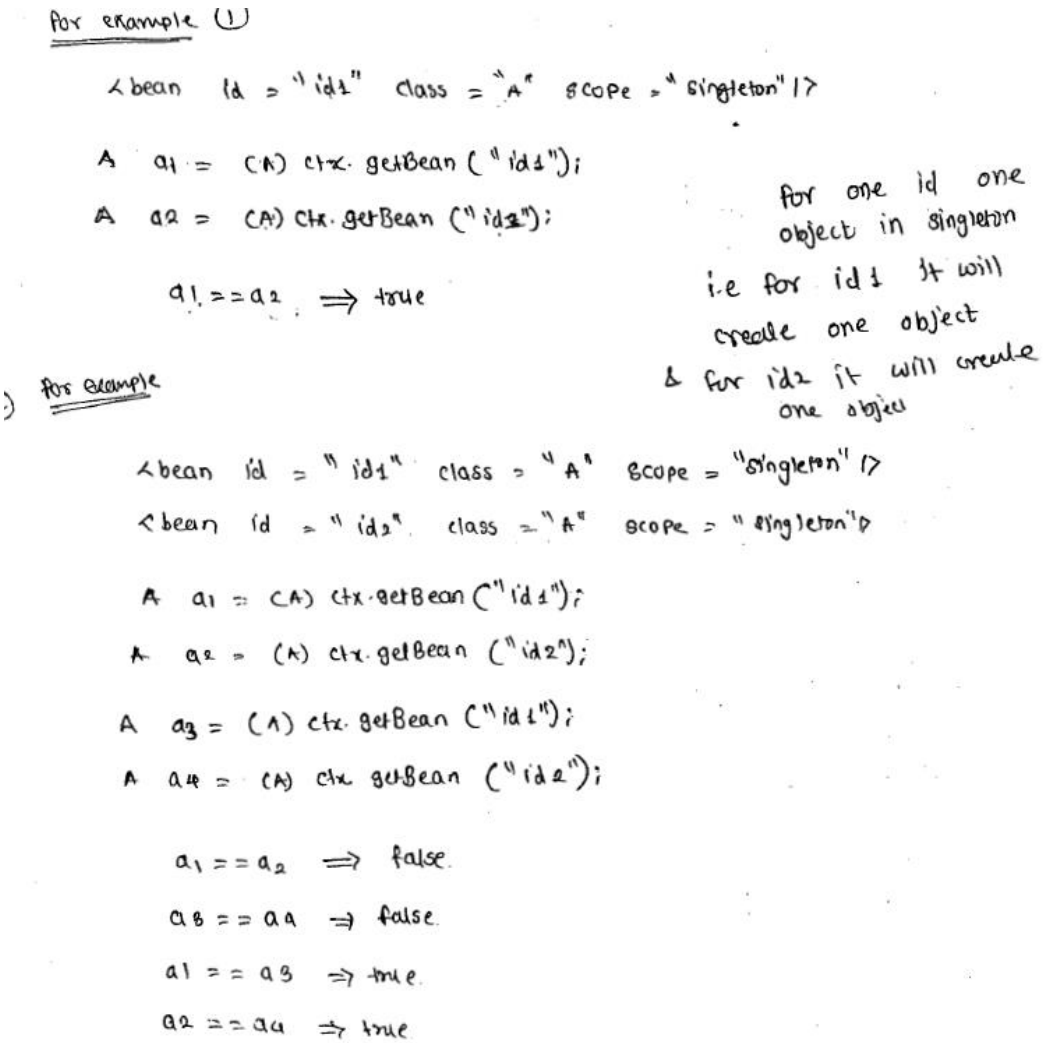
- If a bean class in xml is matched with property type for more than one's then spring container throws org.springframework.beans.factory.unsatifiedDependency exception.
- I a bean class in xml is not matched with property type then container will not inject that dependency so that property remains with default values as null.

⇨ Constructor:

- If Autowiring strategy is constructor then spring container checks a bean class in xml is matched with property type or not if matched then container injects the dependency by calling constructor of the bean.
- Then Autowiring strategy byType and constructor has same verification strategy bit dependency injection type is different.
- The Autowiring strategy byName and byType has same injection type i.e setter but verification is different.
- If default constructor is not present then Spring container will throw Exception.
- Note – **Autowiring is only applicable for reference type but not applicable for primitives type and collection type.**

⇨ Bean Scopes:

- A scope indicates lifetime of an object or value with in a container.
- In order to set scope for bean we need to configure scope attribute along with <bean> tag.
- If we do not configure scope attribute then by default spring container reads the scope as singleton.
- Then available scopes for bean are:
  1. singleton
  2. prototype
  3. request - used in MVC
  4. session – used in MVC
  5. global session – used in portlet
- global session scope can be used in spring with portlet
- request and session scope can be set to a bean only in MVC application.
- **If we set bean scope as a singleton then spring container will create one object of that bean class with respect to the id and return same object for multiple times.**



Note – Spring container make bean object as singleton with respect to id it means when we call multiple times getBean() method by passing the same id, the container returns same object of the bean.

- If the bean scope is **prototype**, then Spring container creates and returns a new object whenever getBean() method is called.

⇨ Making a java class as singleton:

- To configure Java Singleton class in Spring configuration file we need to configure our singleton class like following:

```
<bean id = "id1" class = "A" factory-method = "getInstance" scope=
                                                    "Prototype" />
```

- Even though scope is prototype but the bean class is always singleton because spring container also calls factory-method only to read the object.

⇨ Look-up method injection:

- Suppose in an application we have two bean classes A & B where A depends on B.
- A is a singleton bean and B is a prototype bean.
- In class A we have taken reference variable of B as dependency and then we define setter or constructor for injection dependency.

```
<bean id = "id1" class = "A" scope = "singleton">
    <property name = "b" ref = "id2" />
</bean>
<bean id = "id2" class = "B" scope = "Prototype" />
```

- In the above even though the scope of class B is Prototype but container injects one object of class B only to class A because dependency of a class is injected for once, whenever its object is created.
- **When a prototype bean is dependent on singleton bean then prototype scope bean also started to behave like singleton bean. Because on every call only one object is going to return by container**. So, overcome this problem we use lookup method injection.
- If we want to inject multiple objects of prototype bean to one object of singleton bean then instead of constructor or setter injection, we need to apply lookup method injection. Example –

```
abstract public class A
{
    public abstract B getObjectOfB();
    public void m1()
    {
        getObjectOfB().m2();
    }
}
```

- At runtime spring container creates a proxy class by extending class A and in that proxy it overrides abstract method and returns a new object of class B.

```
<bean id = "id1" class = "A" scope = "singleton">
    <lookup-method name = "getObjectOfB" bean = "id2" />
</bean>
<bean id = "id2" class = "B" scope = "Prototype" />
```

- How can we inject an object created using proxy design pattern into object created using singleton design pattern – Using lookup method injection (So, Spring is using Proxy design pattern in case of lookup method injection).

⇨ Core module with annotation:

- The annotation of core module divided into three types:
  1. Stereo type annotation
  2. Autowiring annotation
  3. Miscellaneous annotation
- Framework has given 4 types of annotations:
  1. @Component
  2. @Service
  3. @Repository
  4. @Controller
- @Component annotation is base annotation for base for all remaining 3 annotations - @Service, @Repositiory and @Controller
- The classes which are added with stereo type annotation are going to be visible to spring container in a "component-scanning"
- Bean Autowiring facility provided by spring container for automatically injecting dependencies of reference type by without explicitly configuring it id.
- Autowiring facility is only applicable for dependencies of reference types.
- Annotation for Autowiring:
  1. @Autowiring
  2. @Qualifier
- These autowired annotation is applicable for fields (variables) methods and for constructors.
- Stereo annotations are class level annotation it means they are only applicable for classes.
- If we add @Autowired annotation then a spring container checks for a bean class in xml is matched with property type or not. If matched then injects that bean object by calling either setter or a constructor. If not matched then throws an exception.
- **required attribute** – if we want to tell the container that if bean class in xml is not matched with property type, then don't throw an exception, then we need to add **required** element to the @Autowited annotation.
- By passing required=false we are telling spring container that a dependency is optional.
- If construction injection is defined then even though required = false container throws an exception. To overcome this, we need to define a default constructor.
- **@Qualifier** – If bean class in a xml is matched for more than one with property type then ambiguity occurs and container throws an exception, to resolve this we use @Qualifier annotation.
- @Scope – This annotation is used to set the scope for a bean class.

```
@Service
@Scope (value = "prototype")

Public class A
  {
  }
```

- **@PostConstruct & @PreDestroy** – These annotations are used to tell the container about **custom init** method and custom destroy method.

```
@Service
Public class A
{
   @PostConstruct
   Public void setUp()
   {
      // initialization logic
   }
   @PreDestroy
   Public void tearDown()
   {
      // destruction logic
   }
}
```

- **@Value** – This annotation is given for injecting a value to a primitive type or simple type property, by reading it from a resource container.
- @Primary – When multiple classes are implementing a common interface then to tell the container give more priority to one implementation class we use @Primary annotation on top of the class.

⇨ Difference between BeanFactory container and ApplicationContext container:

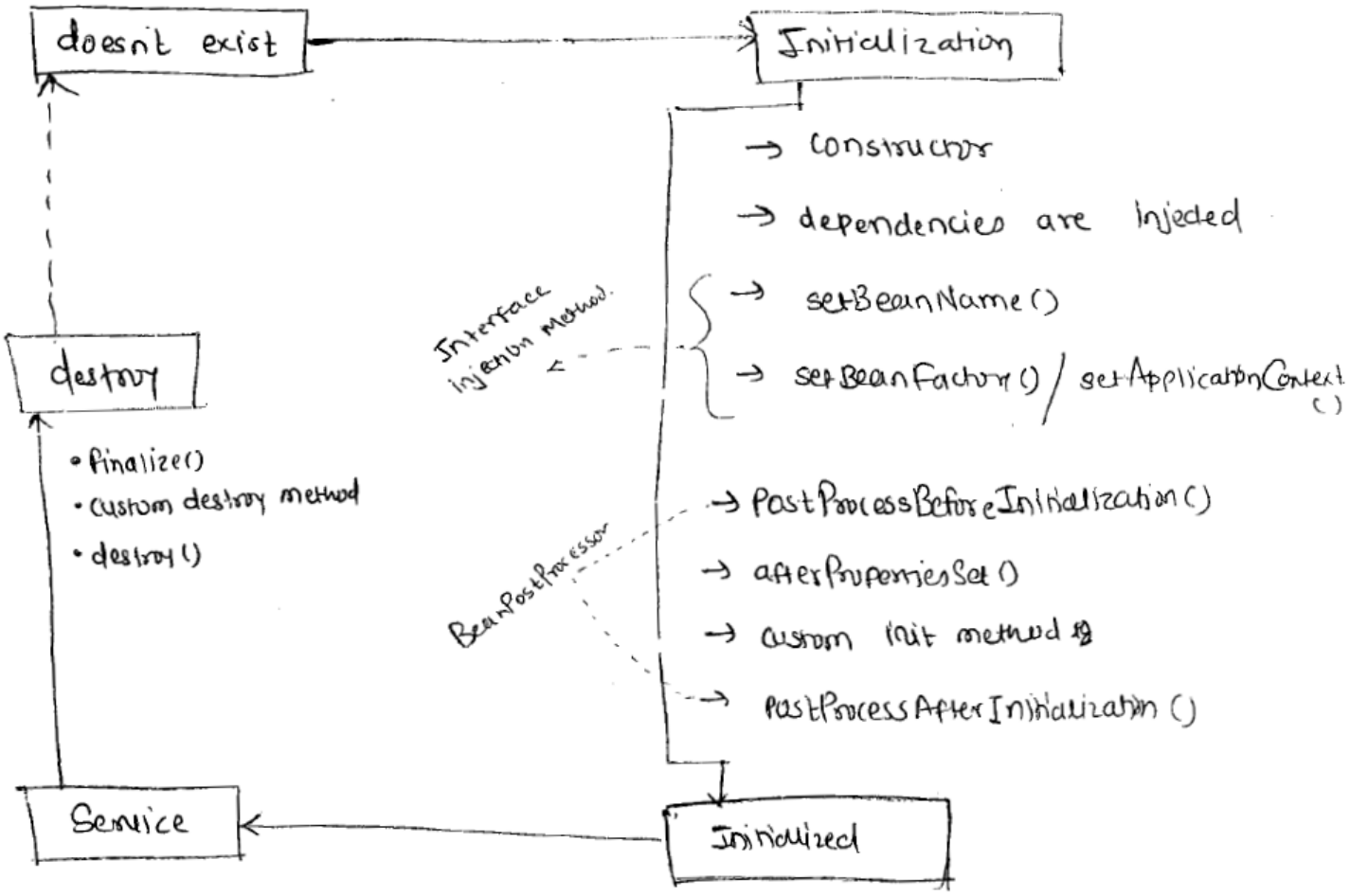
Diff. bet^n BeanFactory & ApplicationContext Container

Never used in real project	always used.
BeanFactory	ApplicationContext
VIMP ① BeanFactory is basic container & it cannot Pre instantiated Singleton beans.	① ApplicationContext is an advanced container and it can Preinstantiate Singleton bean.
② BeanFactory container only supports Singleton & Prototype supports.	② ApplicationContext Container supports all beans scope.
③ SpEL is not supported by BeanFactory.	③ SpEL is supported by ApplicationContext.
IMP ④ In BeanFactory we need to explicitly add an object of a BeanPostProcessor class to the BeanFactoryContainer.	④ In ApplicationContext, we need to configure the BeanPostProcessor class in xml. but we no need to add an object explicitly.

vimp
⑤ a BeanFactory container can create a Bean objects and injects dependency. but it cannot provide other additional services like transaction, scheduling, or messaging etc.

⑤ ApplicationContext container will inject dependencies and also provides additional services to the beans.

⇨ Bean Life cycle:



- In Spring framework, we say that Spring container creates a Bean object but spring container internally calls JVM only for creating an object.
- After instantiating bean class by JVM the JVM calls constructor of the Bean class so that initiating begins.
- After constructor is executed the spring container injects the dependencies required to bean Object.
- The spring container checks whether a bean class implements **BeanNameAware** interface or not, if yes then calls **setBeanName**() method.
- A Spring container checks a bean class implements **BeanFactoryAware** or **ApplicationContextAware** interface or not.
- If **BeanFactoryAware** interface is implanted then calls **setBeanFactory**() method and if **ApplicationContextAware** interface is implanted then calls **setApplicationContext**().
- If **BeanPostPorcessor** class is created then container calls **postPorcessorBeforeInitilization**().
- If Spring bean class implements **InitializingBean** interface then container calls **afterpropertesSet**().
- If a custom **init** method is defined then it will be called.
- **postProcessorAfterInitialization**() of a **BeanPostProcessor** class is called.
- Now the bean initialized and ready to provide the service.

- Before removing a bean object from container checks whether a bean class implements **DisposableBean** interface or not. If yes then it calls **destroy**() method.
- If a custom destroy method is defined then it will be called.
- Before a bean object is going to be garbage collector JVM call finalize() method. Now the bean object is completely destroyed.

⇨ Initializing a Bean class:

Before an object of a Bean class is going to become into usable state if wen want to execute any initializing logic then framework has given 2 options:

1. We can implements our Bean class from **InitializingBean** Interface.
2. We can define a custom **init** method in Bean class.

- If we implements initializing interface then we need to override its abstract method **afterPropertiesSet**()
- As in this case we are implementing interface so the this is invasive approach to make it non-invasive we have to configure **init** method.
- To tell Spring container that a method of our Bean class as **init()** method we need to configure **init-method** attribute with bean tag in xml.
- Before an object of a bean class is going to be garbage collector if we want to execute some destruction logic to the bean by IOC container then the framework has given option for defining the destruction logic.
  1. By implanting our bean class from DisposalBean interface
  2. By defining accustom destroy method
- If we implement Disposal interface then we need to override its abstract method **destroy**() for defining the destruction logic.
- If we want to make our class non-invasive class then we can define a custom method and can configure that method in **destroy-method** in bean tag.

⇨ BeanPostProcessor Interface:

- In a Spring application if a multiple beans are having common initialization logic and if it is define in each bean separately then duplication of the code occurs in an application.
- Instead of duplicating common initializing logic we need to create a separate class by implementing BeanPostProcessor interface to define a common initialization logic.
- As part initialization of spring bean container 1st calls postporcessorBeforeInitialation() method after that initialization method defined in the bean class will be called.