HIBERNATE

JAVA PERSISTENCE FRAMEWORK

Anil Joseph

Created By Anil Joseph(anil.jos@gmail.com)

Agenda

- Introduction to ORM
- Understanding the Hibernate features and Architecture
- Mapping persistence classes
- Implement CRUD operations with Hibernate
- Advanced Queries and HQL
- Transactions
- Concurrency
- Advanced Mappings
- Hibernate with JPA

Created By Anil Joseph(anil.jos@gmail.com)

Software


JDK 1.7 or higher


Eclipse IDE for Java EE Developers


Hibernate


MySQL Database Server


MySQL Workbench


Apache Tomcat

Problems with mapping objects and relational data

Granularity

- Refers to the relative size of types we are working with
- Domain models support coarse-grained, fine grained & simple types
- SQL supports 2 level, tables and columns

Subtypes

- Object oriented languages support inheritance and polymorphism
- Relational databases do not have this concept.

Identity

- Object equality is checked by either "==" or the ***equals*** method
- Database row identity is expressed with the primary key

Associations

- Object associations are directional with references or pointers
- Table associations are non-directional with foreign keys

Created By Anil Joseph(anil.jos@gmail.com)

Object/Relational Mapping(ORM)

ORM provides an ***automated and transparent mechanism*** of persisting objects to the database.

ORM aims to enable the Application Developer deal with underlying persistence only in terms of business entities.

Created By Anil Joseph(anil.jos@gmail.com)

Object/Relational Mapping(ORM)

Mechanisms to map the database schemas to the business entities

Provides API's for CRUD operations

Defines a Language of API for performing queries on objects

Facility for transactions, lazy associations fetching and other optimizations

Created By Anil Joseph(anil.jos@gmail.com)

ORM Implementations

Java	.Net	NodeJS	Objective C
• Hibernate • EclipseLink • MyBatis • Apache OpenJPA	• ADO.Net Entity Framework • NHibernate • LINQ to SQL	• TypeORM	• Core Data

Created By Anil Joseph(anil.jos@gmail.com)

Hibernate

- Hibernate is an object-relational mapping tool for the Java programming language.
- Hibernate is an open-source software that is distributed under the GNU license.
- Hibernate is JPA compliant.

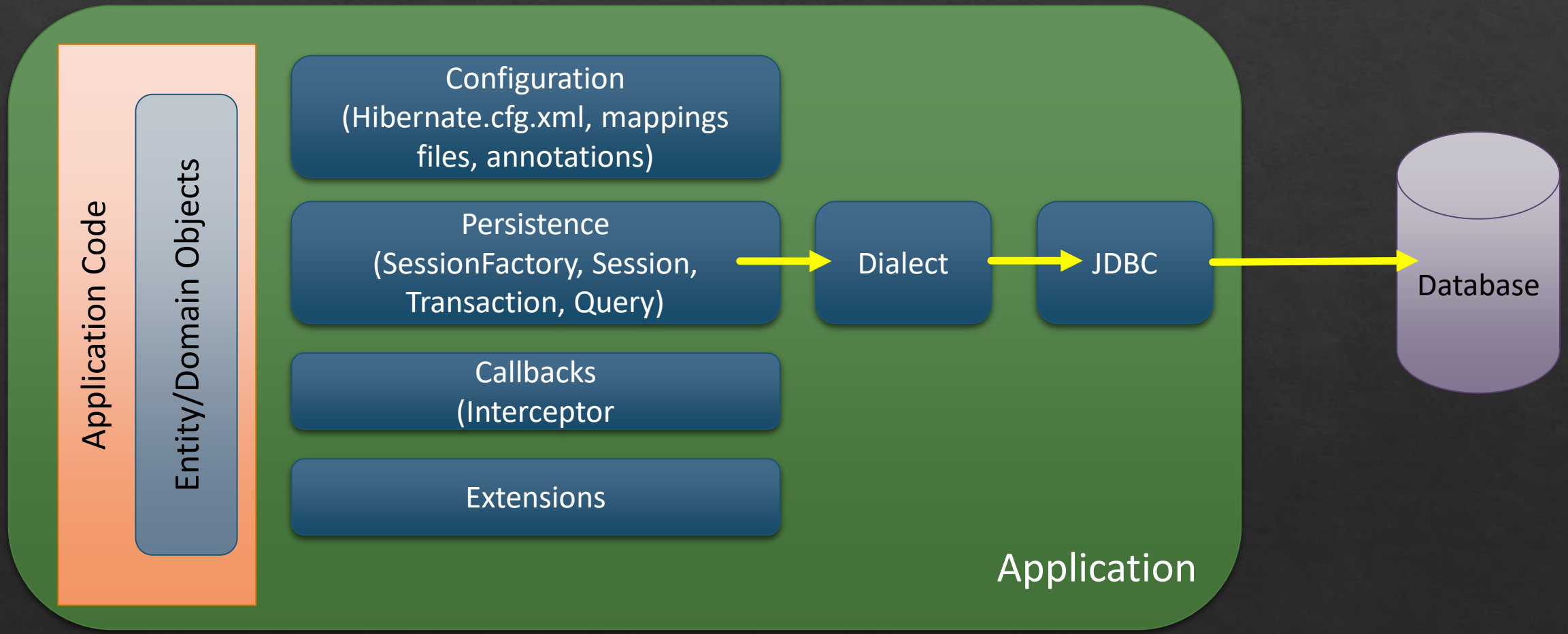
Created By Anil Joseph(anil.jos@gmail.com)

Hibernate Core Features

- Easy to use API's for persistence(CRUD)
- Comprehensive mappings options
- Optimized Queries depending on databases
- Query Execution Language(HQL)
- Transactions(Local and distributed)
- Lazy Loading
- Callbacks
- Caching
- Extensibility

Created By Anil Joseph(anil.jos@gmail.com)

Hibernate Architecture



Created By Anil Joseph(anil.jos@gmail.com)

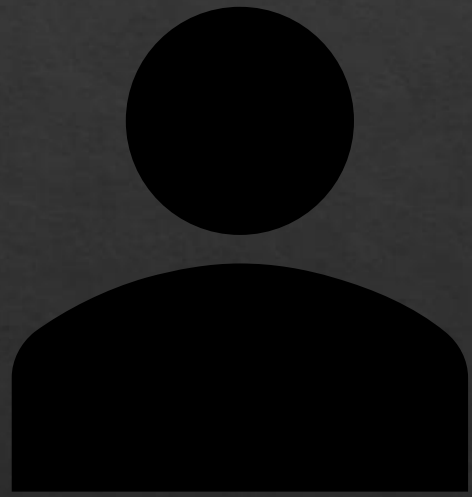
Hibernate Architecture

- Applications make use of hibernate through a set of interfaces.
- Hibernate makes itself accessible to the application at the following levels
  - Configuration (Allowing application to configure hibernate)
  - Persistence (Allowing application to persist its domain object over a session in a transaction. Or to retrieve using queries)
  - Callback (Allowing application to keep track of hibernate events)
  - Extension (Allowing application to add UserTypes, IdentifierGenerator)

Created By Anil Joseph(anil.jos@gmail.com)

Hand-on Exercise

- CRUD operations on an Message table
- Steps to be followed
  - Configure Hibernate
  - Create a Message Business Entity
  - Define the mapping between the table and business entity
  - Write the program to execute the CRUD operations



Created By Anil Joseph(anil.jos@gmail.com)

Mapping Entity to Database Tables

- Entity can be mapped to database tables using two ways
  - Hibernate mapping files(.hbm)
  - Annotations
- The annotations used in Hibernate are the JPA annotations

Created By Anil Joseph(anil.jos@gmail.com)

Mapping Annotations

@Entity

- Defined at the class level for an entity

@Table

- Defined at the class level for an entity to map to the database table

@Column

- Defined on a field or property to map to a database table column

Created By Anil Joseph(anil.jos@gmail.com)

Mapping Annotations

@Id

- Defined on a field or property to mark as the primary key

@Basic

- Defined on a field or property
- Default id no other annotation has been defined.

@Temporal

- Defined on a field or property for date types
- Date, Time , DateTime

Created By Anil Joseph(anil.jos@gmail.com)

Configuration

- The configuration object loads the Hibernate configuration file(hibernate.cfg.xml).
- The hibernate configuration comprises of
  - The database connectivity details
  - Entity mapping information(Hibernate Mapping Files or Entity classes in case of using Annotations)
  - Configuration for dialects, caching, session context etc.
- There would be different configuration for different databases.

Created By Anil Joseph(anil.jos@gmail.com)

Session Factory

The Session Factory creates the Hibernate Session objects

There is typically a single Session Factory instance for the whole application per database.

The Session Factory caches generated SQL statements and other mapping metadata that Hibernate uses at runtime.

Session Factory is a thread-safe object.

Created By Anil Joseph(anil.jos@gmail.com)

Session Interface

The Session is a light-weight interface that acts as a persistence manager in Hibernate.

It represents a single unit of work. In web invocation terms, can be mapped to one client request.

It is not thread safe and so should be used one per thread.

Provides the basic level of object caching in hibernate.

Provides the API's for the CRUD and Query operations

Created By Anil Joseph(anil.jos@gmail.com)

Transactions

- Hibernate uses its own transaction infrastructure.
- Hibernate transaction is an abstraction over low-level API, either JDBC or JTA.
- Transactions are required for any DML(insert, update and delete) operations.

Created By Anil Joseph(anil.jos@gmail.com)

Concurrency

- Multiple transactions that run concurrently can modify the same set of records.
- Such concurrent accesses have to be handled with proper locking mechanisms.
- Conventionally there are 2 mechanisms
  - Pessimistic locking
  - Optimistic locking
- Hibernate supports both models

Created By Anil Joseph(anil.jos@gmail.com)

Concurrency

- Concurrency is managed by
  - Pessimistic Locking
  - Optimistic Locking
- Pessimistic Locking
  - The database row is locked and fetched by the transaction.
  - No other transaction can modify the record until the lock is released.
- Optimistic Locking
  - This does not involve any database write locks.
  - Rows are fetched and modified in the usual manner
  - However on update to the database checks are done to verify if another transaction has modified the record.

Created By Anil Joseph(anil.jos@gmail.com)

Pessimistic Locking: Lock Options

- LockMode.None—Don’t go to the database unless the object isn’t in either cache.
- LockMode.Read—Bypass both levels of the cache, and perform a version check to verify that the object in memory is the same version that currently exists in the database.
- LockMode.Upgrade—Bypass both levels of the cache, do a version check (if applicable), and obtain a database-level pessimistic upgrade lock, if that is supported.
- LockMode.UpgradeNoWait—The same as UPGRADE, but use a SELECT...FOR UPDATE NOWAIT, if that is supported. This disables waiting for concurrent lock releases, thus throwing a locking exception immediately if the lock can’t be obtained.

Created By Anil Joseph(anil.jos@gmail.com)

Optimistic Locking

- Hibernate supports 2 mechanisms
  - Version
  - Optimistic Locking Configuration
    - All
    - Dirty

Created By Anil Joseph(anil.jos@gmail.com)

Optimistic locking in application (Versions)

- When using Version, a column should be added to the table and mapped to a property in the Entity class

```
<version name="version" column="VERSION" />
```

- Whenever the object is changed, hibernate will automatically update the version number

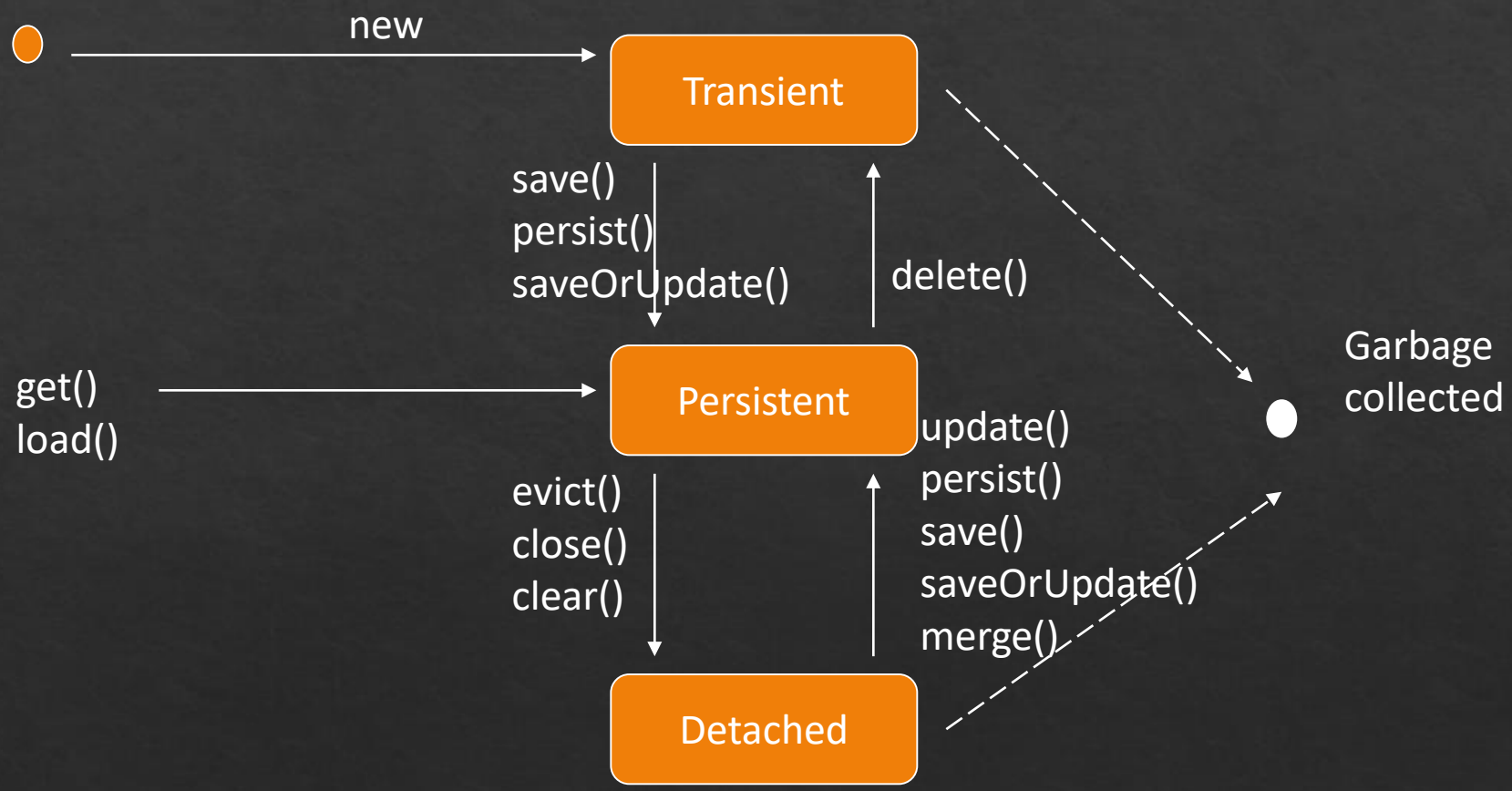
Created By Anil Joseph(anil.jos@gmail.com)

Lifecycle of persistent object

- The domain objects (Entity) are referred to as persistent objects.
- A persistent object can be in one of the three states
  - Transient
  - Persistent
  - Detached

Created By Anil Joseph(anil.jos@gmail.com)

Lifecycle of persistent object



Entity States

- Transient
  - An object is transient if it has been instantiated using the ***new*** operator, and it is not associated with Hibernate Session.
  - It has no persistent representation in the database.
- Persistent
  - A persistent object is associated with the Hibernate session.
  - A persistence instance has a representation in the database and an identifier value.
  - Hibernate will detect any changes made to an object in persistent state and synchronize the state with the database when the unit of work completes

Created By Anil Joseph(anil.jos@gmail.com)

Entity States

- Detached
  - A detached instance is an object that was persistent, but is now disassociated from the session now.
  - Detached instances have to be reattached to a Session to synchronize the changes to the database.
  - Detach objects facilitate a programming model for long running units of work that require ***user think-time***.

Created By Anil Joseph(anil.jos@gmail.com)

Mappings in Details

- Classes that represent domain objects/database tables can be of 2 types
  - Entity
  - Value Type/Embeddable types

Created By Anil Joseph(anil.jos@gmail.com)

Entity

- An object of entity type has its own database identity (primary key value).
- An entity has its own lifecycle; it may exist independently of any other entity.
- Entities are mapped as ***class*** in the configuration file or using the @**Entity** annotation.

Value Types

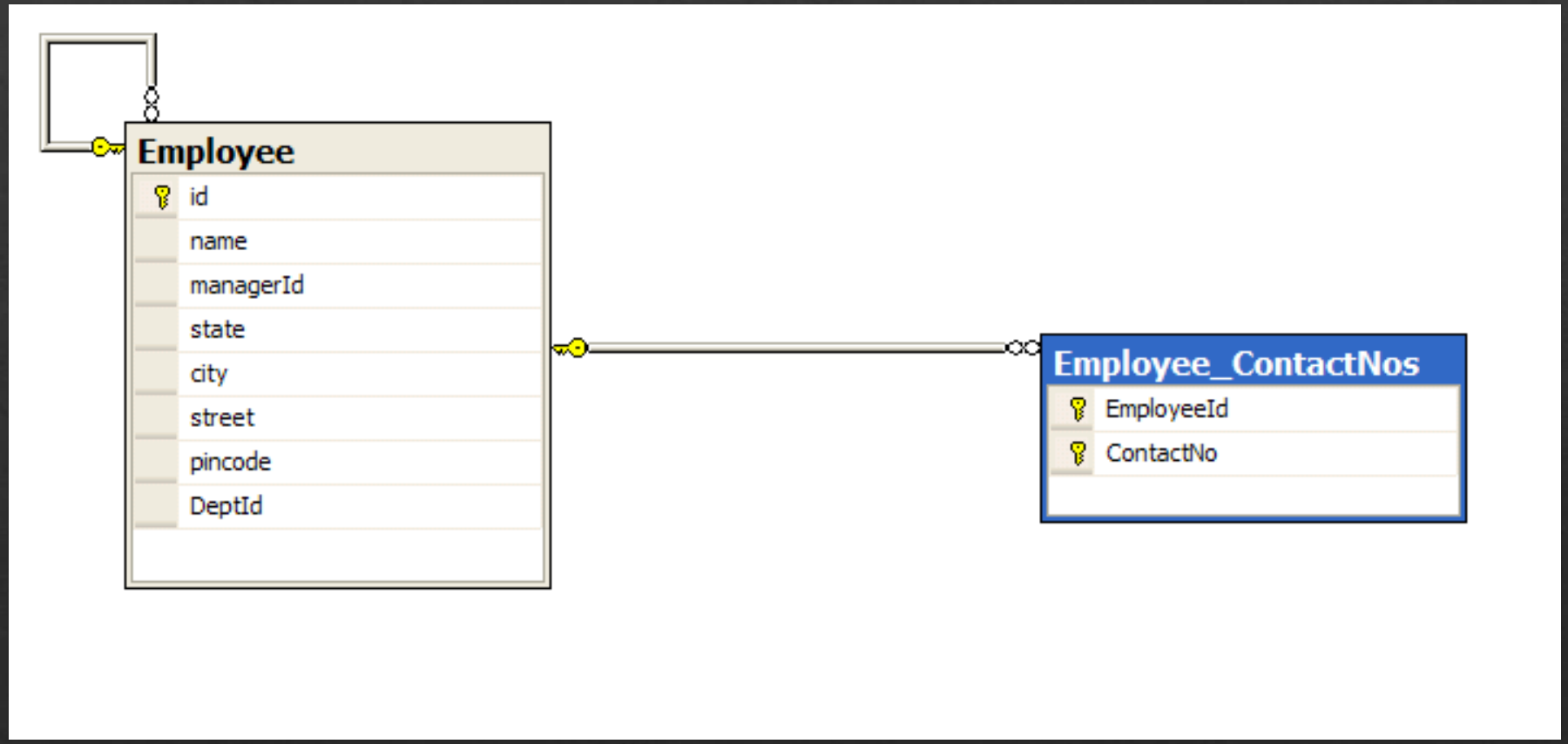
- An object of value type has no database identity; it belongs to an entity
- The lifespan of a value-type instance is bounded by the lifespan of the owning entity.
- Value types are mapped as ***component*** in the mapping file or using the **@Embedded** and **@Embeddable** annotations.
- When tables have composite keys they are represented in an Entity class as an Embeddable

Mapping: Value type Collections

- Mappings to collections(value type) are provided by the following types
  - Set
    - A collection that does not allow duplicates
  - Bag
    - A collection with duplicates
  - List
    - An ordered collection

Created By Anil Joseph(anil.jos@gmail.com)

Mapping with Set



Created By Anil Joseph(anil.jos@gmail.com)

Mapping with Set(Annotations)

```
@ElementCollection(fetch=FetchType.EAGER)
@CollectionTable(name=" Employee_ContactNos",
      joinColumns=@JoinColumn(name=" EmployeeId "))
@Column(name=" ContactNo ")
private Set<String> contactNos = new HashSet<>();
```

Created By Anil Joseph(anil.jos@gmail.com)

Mapping with Set(XML)

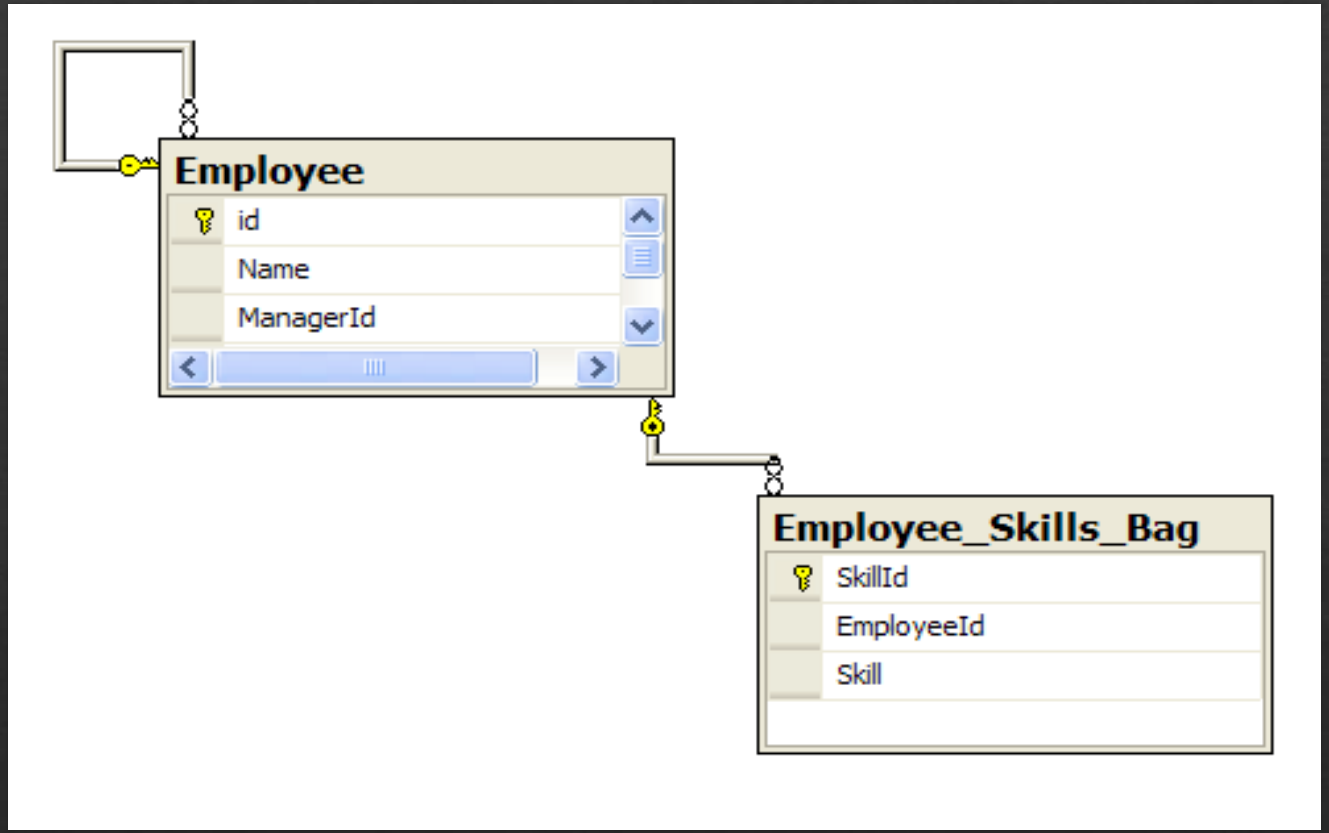
- Entity
  - private Set<string> contactNos;
  - get & set Methods

- Mapping

```
<set name=" contactNos" table="Employee_ContactNos" lazy="true">
    <key column="EmployeeId"></key>
    <element column="ContactNo"></element>
</set>
```

Created By Anil Joseph(anil.jos@gmail.com)

Mapping with Bag



Created By Anil Joseph(anil.jos@gmail.com)

Mapping with Bag(XML)

- Entity
  - private List<String> skills;
  - get & set Methods

- Mapping

```
<idbag name="skills" lazy="true" table="Employee_Skills_Bag">
        <collection-id type="Integer" column="SkillId">
            <generator class="sequence"/>
        </collection-id>
  <key column="EmployeeId"/>
  <element type="String" column="Skill" not-null="true"/>
</idbag>
```

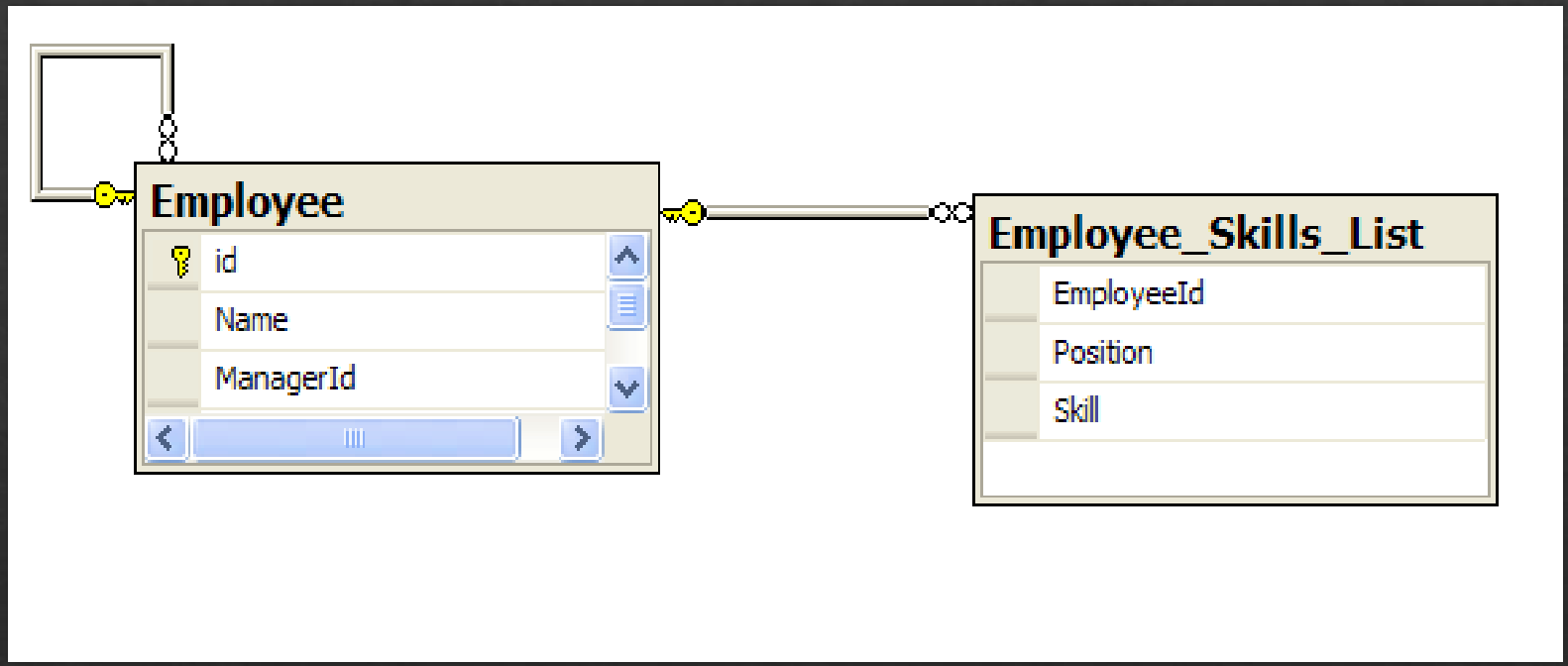
Created By Anil Joseph(anil.jos@gmail.com)

Mapping with Bag(Annotations)

```
@ElementCollection(fetch=FetchType.EAGER)
@CollectionTable(name=" Employee_Skills_Bag",
      joinColumns=@JoinColumn(name="EmployeeId"))
@Column(name=" Skill")

@CollectionId(generator="sequence",    columns={@Column(name="SkillId")},
      type=@Type(type="Integer"))
private List<String>  skills = new ArrayList<>();
```

Created By Anil Joseph(anil.jos@gmail.com)

Mapping with List(ordered collection)



Created By Anil Joseph(anil.jos@gmail.com)

Mapping with List: XML

- Entity
  - private List<string> skills;
  - get & set Methods

- Mapping

```
<list name="skills" lazy="true" table="Employee_Skills_List">
 <key column="EmployeeId"/>
 <index column="Position"/>
 <element type="String" column="Skill" not-null="true"/>
</list>
```

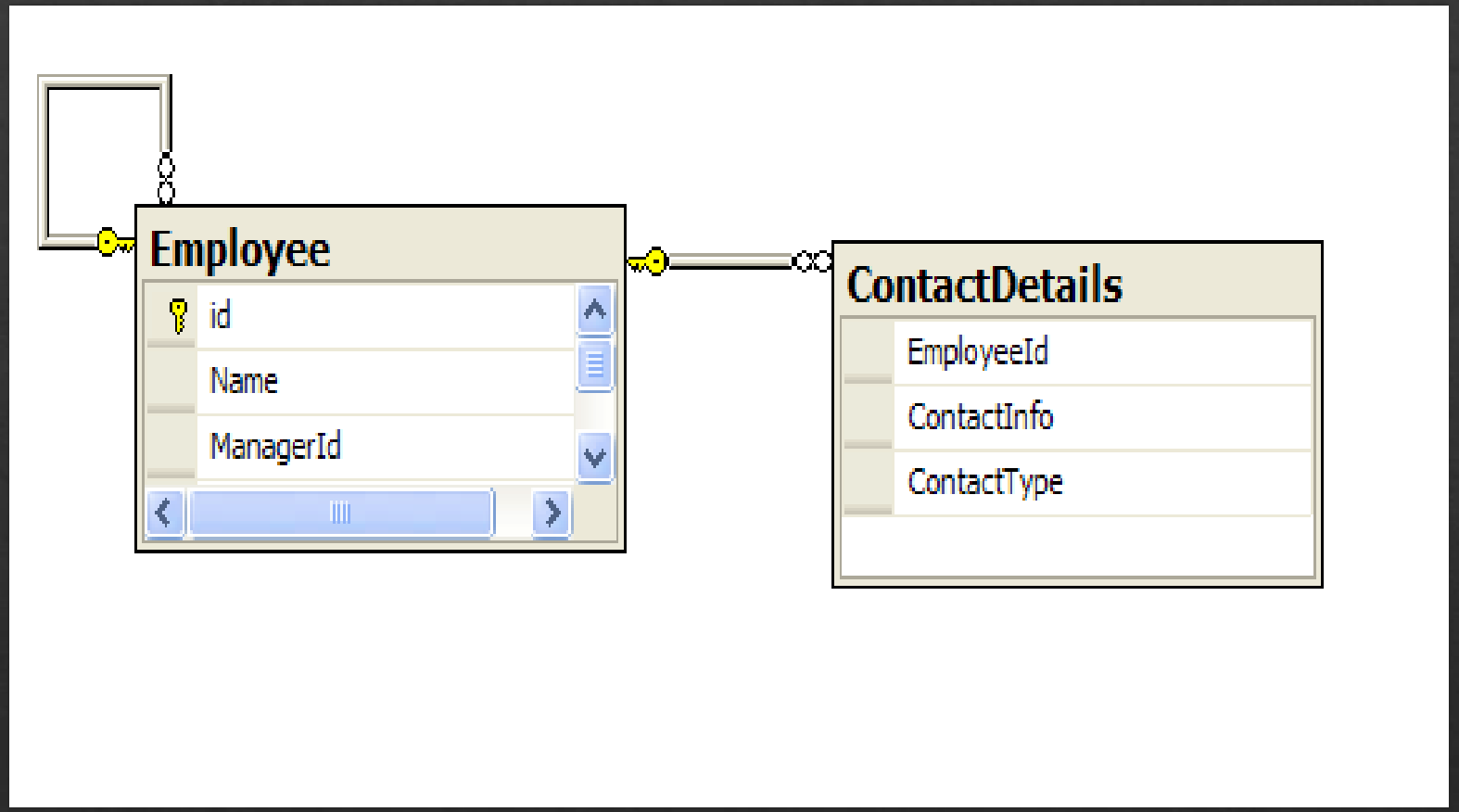
Created By Anil Joseph(anil.jos@gmail.com)

Mapping with List(Annotations)

```
@ElementCollection(fetch=FetchType.EAGER)
@CollectionTable(name=" Employee_Skills_List ",
      joinColumns=@JoinColumn(name="EmployeeId"))
@Column(name=" Skill")

@OrderColumn(name="position")
private List<String>  skills = new ArrayList<>();
```

Created By Anil Joseph(anil.jos@gmail.com)

Mapping with Set(Components)



Created By Anil Joseph(anil.jos@gmail.com)

Mapping with Set(Components)

- Value Type Class

```
public class ContactDetails
{
        private String contactInfo;
        private String contactType;

        //get & set methods
}
```

Created By Anil Joseph(anil.jos@gmail.com)

Mapping with Set(Components)

- Entity Class

```
private Set<ContactDetails> contacts;
```

Mapping with Set(Components): XML

- Mapping

```
<set name="Contacts" table="ContactDetails" lazy="true">
    <key column="EmployeeId"></key>
    <composite-element class="ContactDetails">
     <property name="ContactInfo" column="ContactInfo"/>
     <property name="ContactType" column="ContactType"/>
    </composite-element>
 </set>
```

Created By Anil Joseph(anil.jos@gmail.com)

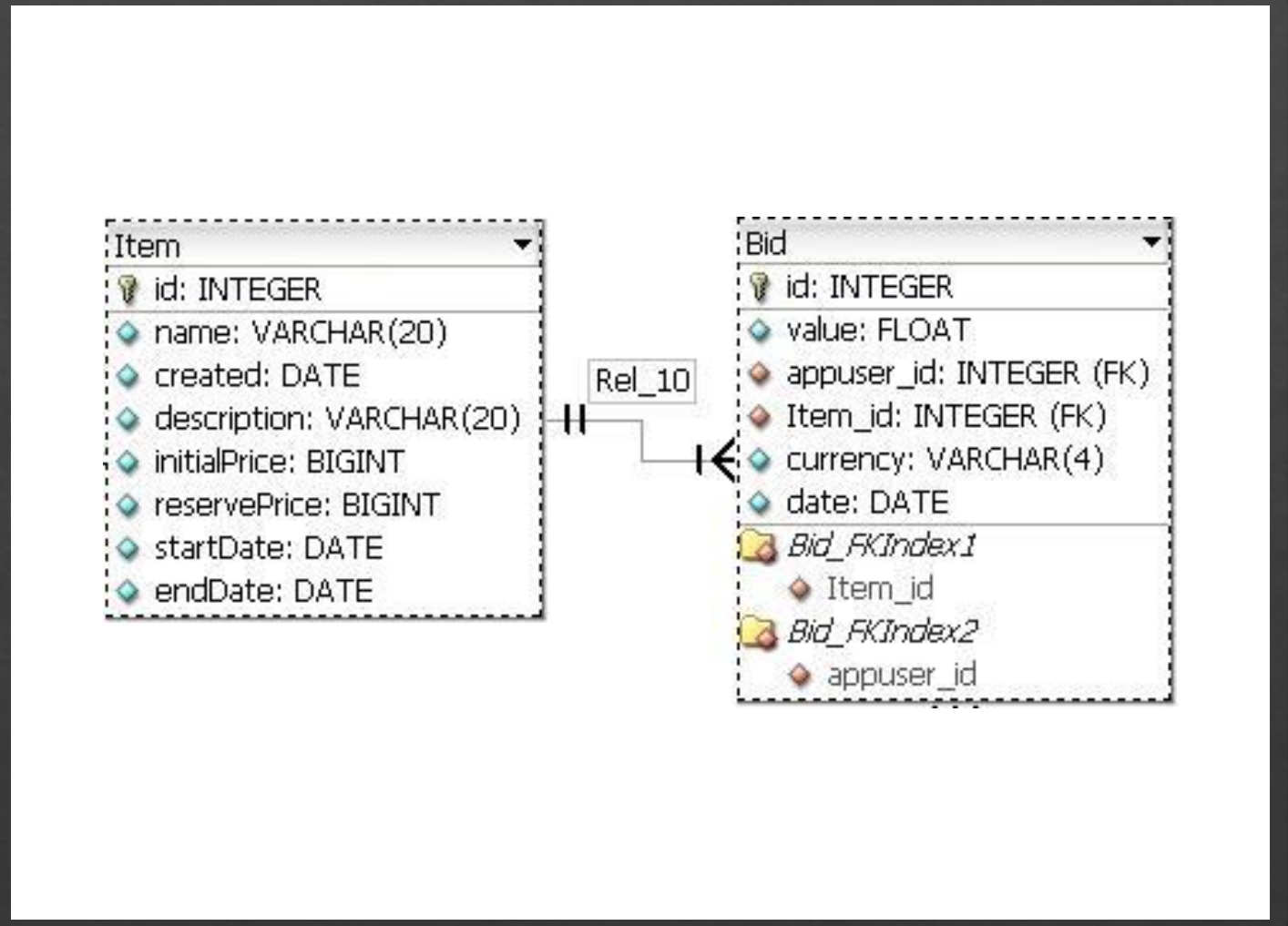
Mapping Entity Associations

- Hibernate allows us to define the associations between ***entities*** as
  - Unidirectional
  - Bidirectional
  - Multiplicity
    - One to One
    - One to many
    - Many to Many

Created By Anil Joseph(anil.jos@gmail.com)

One-to-Many

- Consider the following example with 2 tables of a auction application. Item and Bid tables.
- An item in an auction will have multiple bids placed on it.
- The association type here is one-to-many.



Created By Anil Joseph(anil.jos@gmail.com)

One-to-Many Mapping: Annotations

- In the Bid entity
  - @ManyToOne
  - @JoinColumn(name="itemId")
  - **private Item item;**

- In the Item entity
  - @OneToMany(mappedBy="item")
  - **private Set<Bid> bids**

Created By Anil Joseph(anil.jos@gmail.com)

One-to-Many Mapping: XML

- For unidirectional navigation, the item class will hold a collection of bids
- The relationship will be mapped using the <set> element

Item.hbm.xml

```
<set name="bids“ table=“Bid” inverse=true>
        <key column="Item_ID"/>
        <one-to-many class="Bid"/>
</set>
```

Created By Anil Joseph(anil.jos@gmail.com)

One-to-Many Mapping: XML

- For bidirectional navigation, the bid class will hold the item for which the bid is created.
- The item property can be mapped as follows

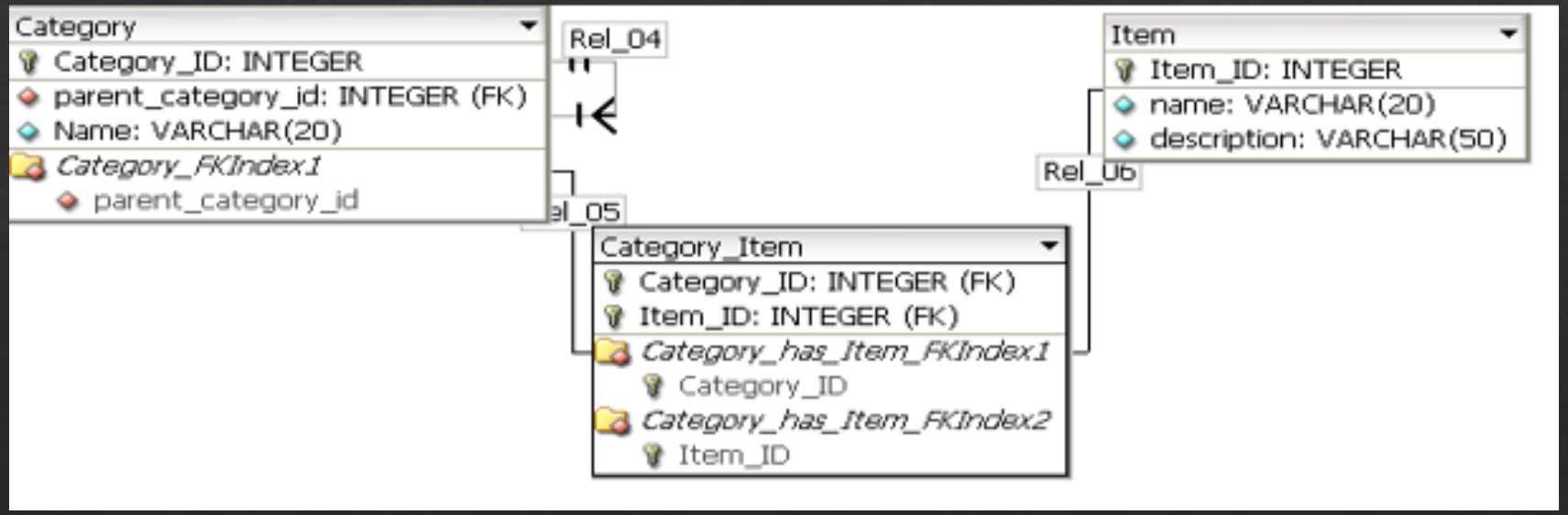
Bid.hbm.xml

```
<many-to-one name=“item“ column="Item_ID“
        class=“Item”
        not-null=“true"/>
```

Created By Anil Joseph(anil.jos@gmail.com)

Many-to-Many

- Considering the auction example where every item in an auction can belong to multiple categories, while a category has multiple items.
- A many to many relation should ideally make use of a link table
- The link table will have foreign keys to the primary keys of the original table



Created By Anil Joseph(anil.jos@gmail.com)

Many-to-Many Mapping: Annotations

- In the Category Entity
  - @ManyToMany
  - @JoinTable(name="CATEGORY_ITEM", joinColumns=@JoinColumn(name="categoryId"), inverseJoinColumns=@JoinColumn(name="itemId"))
  - **private Set<Item> items;**
- In the Item entity
  - @ManyToMany(mappedBy="items")
  - **private Set<Category> categories;**

Created By Anil Joseph(anil.jos@gmail.com)

Many-to-Many Mapping: XML

- The category class in this case will hold reference to the Item in the form of collection of value types

Category.hbm.xml

```
<set name="items"
        table="CATEGORY_ITEM"
                lazy="true"
                cascade="save-update“>
        <key column="category_id"/>
        <many-to-many class="Item" column="Item_ID"/>
</set>
```

Created By Anil Joseph(anil.jos@gmail.com)

Many-to-Many Mapping: XML

- Item class holds reference to categories

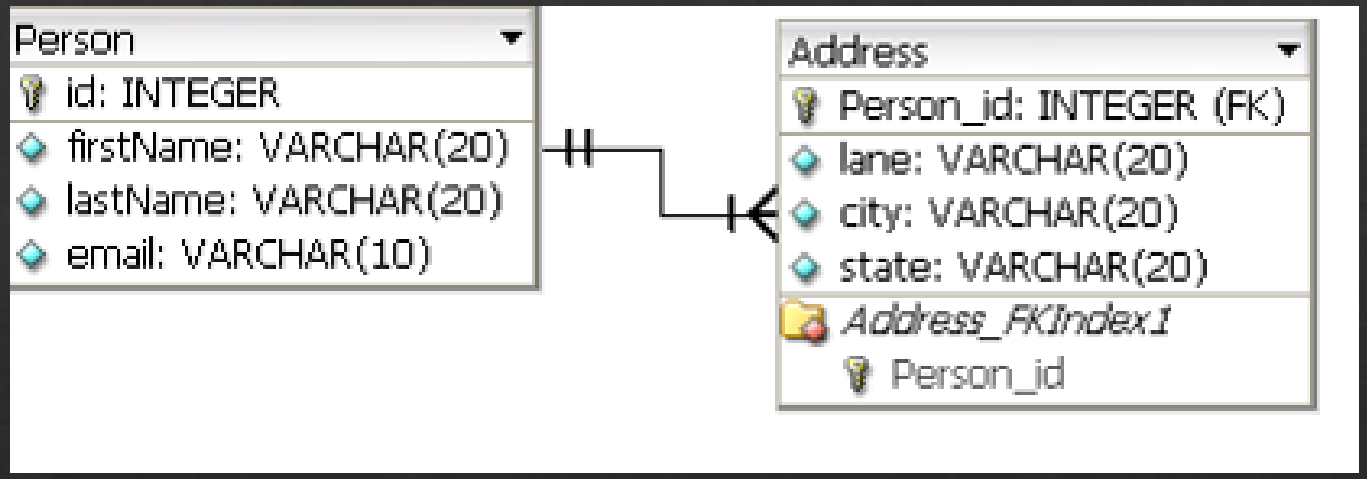
Item.hbm.xml

```
<set name=“categories"
        table="CATEGORY_ITEM"
                lazy="true“
                cascade="save-update“>
        <key column=“Item_ID"/>
        <many-to-many class=“Category" column=“Category_ID"/>
</set>
```

Created By Anil Joseph(anil.jos@gmail.com)

One-to-One

- The one-to-one relation can also be established through primary key association
- The primary key will also act as a foreign key in this case



Created By Anil Joseph(anil.jos@gmail.com)

One to One Mapping: Annotations

- In the Address Entity
  - @OneToOne
  - @PrimaryKeyJoinColumn
  - **private Person person;**
- In the Person Entity
  - @OneToOne(mappedBy=“person")
  - **private Address address;**

Created By Anil Joseph(anil.jos@gmail.com)

One to One Mapping: Annotations

- To ensure that newly saved Address instances are assigned the same primary key as that of the person, a special identifier-generator needs to be used

```
<class name="com.sample.hibernate.mode.Address"
                                                table="Address">
        <id name="id" column="Person_id" >
                <generator class="foreign">
                        <param name="property">person</param>
                </generator>
        </id>
        <one-to-one name=“person"
                class="com.sample.hibernate.mode.Person"
                                                constrained=“true"/>

        </class>
```

Created By Anil Joseph(anil.jos@gmail.com)

Inheritance Mapping

- Consider the auction application the customer has multiple billing options.
  - *Credit card*
  - *Bank account*
- This would be designed as a inheritance hierarchy

Created By Anil Joseph(anil.jos@gmail.com)

Inheritance Mapping: Strategies

Table per concrete class(Table per class)

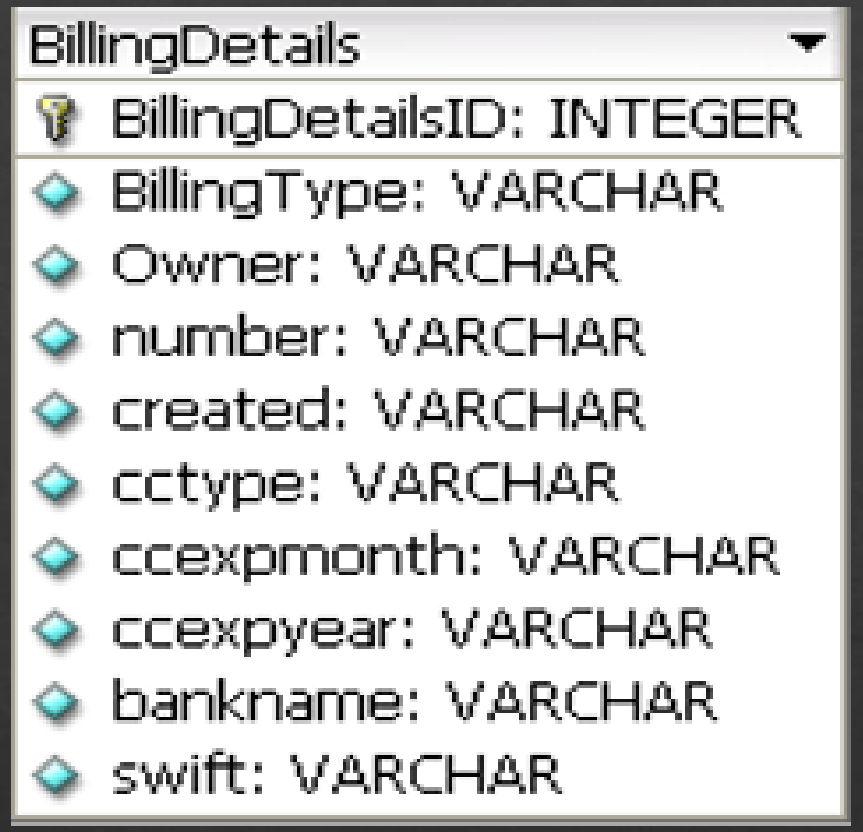
Table per class hierarchy(Single Table)

Table per sub class(Joined)

Created By Anil Joseph(anil.jos@gmail.com)

Mapping (Table per class hierarchy)

- A single table representing all the details
- Although there is a single table the classes will still be separate classes
- This strategy support polymorphic behavior



Created By Anil Joseph(anil.jos@gmail.com)

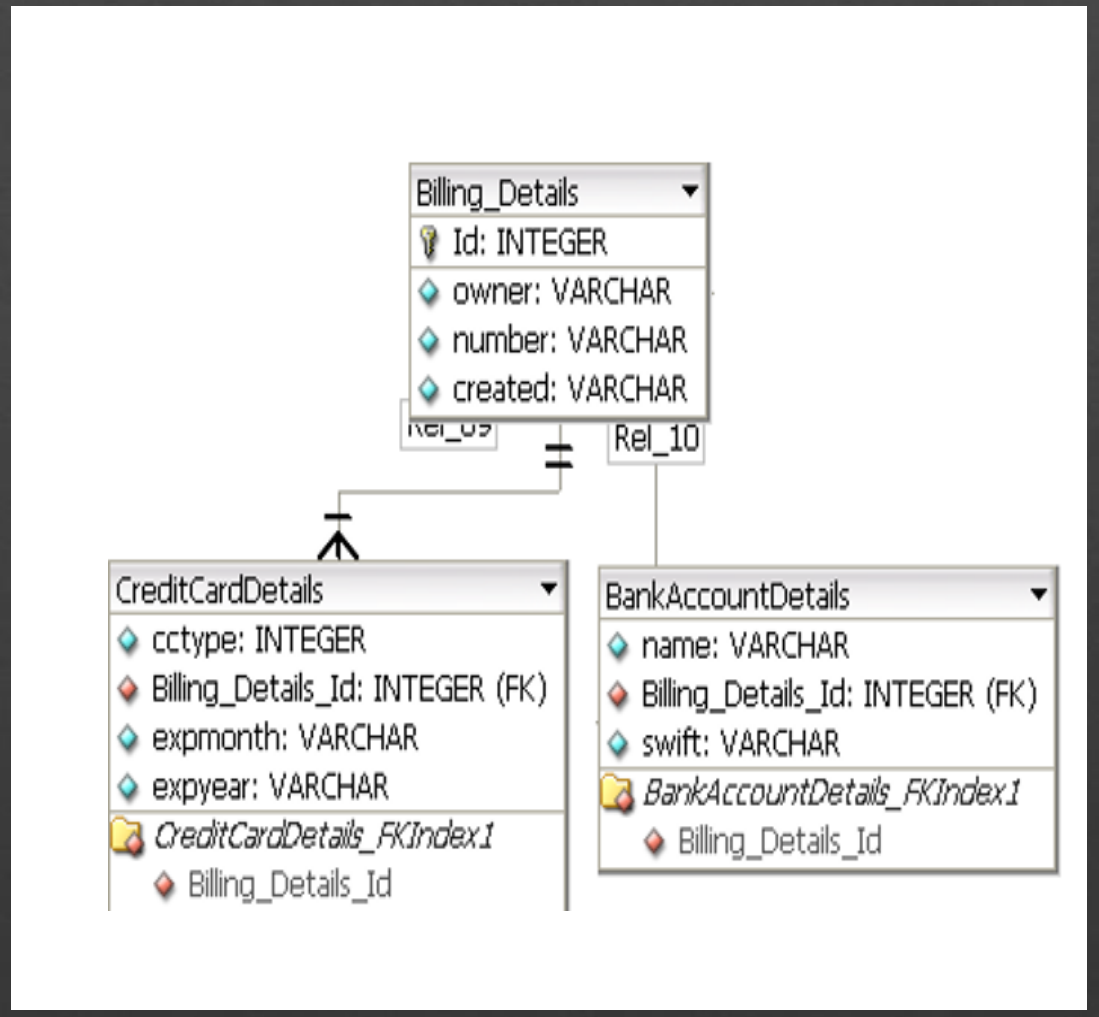
Mapping (Table per class hierarchy)

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE) @DiscriminatorColumn(  name="billingType",
        discriminatorType=DiscriminatorType.STRING ) @DiscriminatorValue("BD")
@Table(name="BillingDetails")
 public class BillingDetails { … }

@Entity
@DiscriminatorValue("CC")
public class CreditCard extends  BillingDetails  { … }

@Entity
@DiscriminatorValue("BA")
public class BankAccount extends  BillingDetails  { … }
```

Created By Anil Joseph(anil.jos@gmail.com)

Mapping(Table per subclass)

- The tables contain only the required columns and a foreign key to the BillingDetails
- *The foreign key also acts as the primary key of the table*



Created By Anil Joseph(anil.jos@gmail.com)

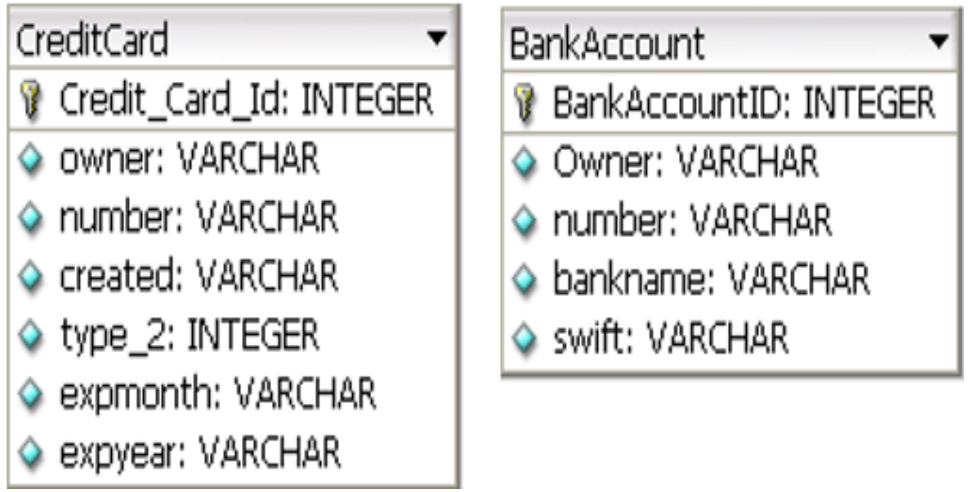
Mapping (Table per subclass)

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@Table(name="BillingDetails")
public class BillingDetails { … }

@Entity
@PrimaryKeyJoinColumn(name="Billing_Details_Id")
@Table(name="CreditCard")
public class CreditCard extends  BillingDetails  { … }

@Entity
@PrimaryKeyJoinColumn(name=" Billing_Details_Id ")
@Table(name="BankAccount")
public class BankAccount extends  BillingDetails  { … }
```

Created By Anil Joseph(anil.jos@gmail.com)

Mapping (Table per concrete class)

- In this case, the user name and details are same, only the type specific details differ
- This model has limited support for polymorphic associations and polymorphic joins.



Created By Anil Joseph(anil.jos@gmail.com)

Mapping (Table per concrete class)

```
@Entity
@Inheritance(strategy=InheritanceType. TABLE_PER_CLASS)
public class BillingDetails { ... }

@Entity
public class CreditCard extends  BillingDetails  { ... }

@Entity
public class BankAccount extends  BillingDetails  { ... }
```

Created By Anil Joseph(anil.jos@gmail.com)

Hibernate type system

- When working with hibernate, we have to deal with two data types
- The data type of the underlying database
- The Java data types
- The Java types need to be mapped to the data types in the database
- In the mapping files, the types are mapped as follows
  - <property name="email" column="EMAIL" **type="string"**/>

Built-in mapping types

- Hibernate comes with in built-in mapping types for various java and SQL types

Mapping type	Java type	SQL type
integer	Int or java.lang.Integer	INTEGER
Long	Long or java.lang.Long	BIGINT
Short	Short or java.lang.Short	SMALLINT
Double	Double or java.lang.Double	DOUBLE
Big_decimal	Java.math.BigDecimal	NUMERIC
Character	Java.lang.String	CHAR(1)
String	Java.lang.String	VARCHAR
Byte	Byte or java.lang.Byte	TINYINT
Boolean	Boolean or java.lang.Boolean	BIT
Yes_no	Boolean or java.lang.Boolean	CHAR(1) (‘Y’ or ‘N’)
True_false	Boolean or java.lang.Boolean	CHAR(1) (‘T’ or ‘F’)

Built-in mapping types

- The built in data type for dates and times are

Mapping type	Java type	SQL type
Date	Java.uti.Date or java.sql.Date	DATE
Time	Java.util.Date or java.sql.Time	TIME
Timestamp	Java.util.Date or java.sql.Timestamp	TIMESTAMP
Calendar	Java.util.Calendar	TIMESTAMP
Calendar_date	Java.util.Calendar	DATE

Built-in mapping types

- The built in data type for large object mappings are

Mapping type	Java type	SQL type
Binary	Byte[]	VARBINARY (or BLOB)
Text	Java.lang.String	CLOB
Serializable	Any java class that implements Serializable	VARBINARY (or BLOB)
Clob	Java.sql.Clob	CLOB
Blob	Java.sql.Blob	BLOB

- The java.sql.Blob or java.sql.Clob may be the most efficient way of handling large objects
- But these may not be supported by drivers for all databases
- If a Blob or Clob is referenced in a persistent class, then they will only be available in a transaction and not when it is in detached state

Basic Queries

Queries

- Queries are most important for writing good data access code
- Hibernate provides extensive support for querying using its Query interface
- Queries are in terms of classes and properties and not tables and columns
- Hibernate also supports execution of native SQL queries

Queries

- Queries can be created on a session
  - Query hqlQuery = session.createQuery(“from Person”);
- Queries support pagination by limiting results
  - Query query = session.createQuery(“from User”);
  - Query.setFirstResult(0);
  - Query.setMaxResults(10);

Queries

- Queries support criteria for filtering searches based on criteria
  - Session.createCriteria(Category.class).add(Expression.like(“name”, “Laptop%”));
- List() method executes the created query and returns the results as a list
  - List result = session.createQuery(“from User”).list();

Named Queries

- Named queries help in avoiding HQL queries to be located within Java files
- The queries to be used in an application can be externalized in an XML mapping file
- The desired query can be referred to by the name

Named Queries

- getNamedQuery() method obtains a Query instance from a named query

```
session.getNamedQuery("findItemsByDescripti
on").
    setString("description", description).list();
```

```
        <query
name="findItemsByDescription"><![CDATA[fro
m Item item where item.description like
:description]]>
        </query>
```

- Named queries allow easier optimization of queries

Advanced Queries

The query API

- The *Query* interface of hibernate is used for retrieving the data from database
- *Criteria* interface is for specifying query criteria
  - *Query hquery = session.createQuery()*
  - *Criteria crit = session.createCriteria()*

Binding Parameters

- For passing parameters to a query, the following approaches can be used
  - *Named Parameter*
  - *Positional Parameter*
- *Named Parameter*
  - *Allows you to specify the name of the expected parameter*
    - *String query = “from item where item.description like :searchString”*
  - *In this case “searchString” is the named parameter*
    - *Query qu = session.createQuery(query).setString(“searchString”, searchString);*

Binding Parameters

- Positional Parameter
  - Allows you to specify the parameter position using “?” like prepared statements
    - String query = “from item where item.description like ? And item.date > ?”
    - Query qu = session.createQuery(query).setString(0, searchString).setDate(1,minDate);

Binding Parameters

- As seen in the above examples, the setString() and setDate() methods were used, Hibernate also allows setting of various other types as well as entity
  - Session.createQuery(“from item where item.seller = :seller”).setEntity(“seller”, seller);

Ordering query results

- For ordering the results obtained from queries, hibernate provides order by clause
  - “from User u order by u.username”
  - “from User u order by u.username desc”

- The Criteria API also provides ordering
  - Session.createCriteria(User.class).addOrder(Order.asc(“lastname”));

Association

- The hibernate query language supports performing queries on associations
- *Join* can be used to combine data in two (or more) relations
- HQL provides four ways of expressing joins
  - An ordinary join in the from clause
  - A *fetch* join in the from clause
  - A *theta-style* join in the where clause
  - An *implicit* association join

Association

- A *fetch* join can be used to specify that an association should be eagerly fetched

```
From Item item
join fetch item.bids
    Where item.description like “%gc%”;
```

Association

- An alias can be to make query less verbose

```
From Item item
join fetch item.bids
    Where item.description like “%gc%”;

Session.createCriteria(Item.class).
    setFetchMode(“bids”, FetchMode.EAGER).
    list()
```

Implicit association

- HQL supports multipart property paths for querying

```
From User u where u.address.city=“city”
```

```
Session.createCriteria(User.class).
add(Expression.eq(“address.city”,”city”));
```

Theta Style Join

- Theta style joins are to perform joins between two un-associated classes

  From user , category

- This will return an Object[] with 2 elements

Aggregation

- The hibernate query language supports performing various aggregation of the results
- The supported aggregation functions are
  - avg(...), sum(...), min(...), max(...)
  - count(*) count(...), count(distinct ...)
  - count(all...)
- Select count(*) from Item
- Select sum(amount) from Bid

Grouping

- Any property or alias appearing in an HQL outside the aggregate function can be used for grouping results
  - Select u.lastname, count(u) from User u group by u.lastname
- Here the results will be grouped according the user's lastname
- Grouping can be restricted using the *having* clause
  - Select u.lastname, count(u) from User u group by u.lastname having user.lastname like 'A%'

Sub-queries

- In the databases that support sub-selects, hibernate supports sub-queries
- Sub-queries allow a select query to be embedded in another query
  - From User u where 10 < (select count(i) from u.items I where i.successfulBid is not null)
- This returns all the users who have bid successfully more than 10 times

Native queries

- Hibernate provides support for making use of native SQL queries instead of HQL
- The SQL query results return the entity instance just like hibernate queries

Scalar Queries

- SQL queries can be used to return a list of scalars

```
String sql = “select * from appuser”
    ses.createSQLQuery(sql)
        .addScalar("username", Hibernate.STRING)
        .addScalar("name",Hibernate.STRING)
        .list();
```

- This will return the scalar values as List of Object[]

Native queries

- SQL query may return multiple entity instances
- SQL query may return multiple instances of same entity
- Hibernate needs to distinguish between different entities
- Hibernate uses a naming scheme for the result column aliases to correctly map column values to the properties of particular instances

Entity Queries

- Hibernate can also allow you to obtain Entity instances using SQL queries
- The SQL queries will need to be supplied with the Entity that represents the table

```
String sql = “Select * from address”;


Session.createSQLQuery(sql).
    addEntity(Address.class);
```

Handling associations

- When an entity has a many-to-one relation with another, then it is required to return the association or else a column not found error will be returned
- A * notation will return the association

```
String sql = "select * from appuser";


ses.createSQLQuery(sql).
    addEntity(Appuser.class).
    list();
```

Returning Non-Managed entities

- SQL can be used to return non-managed entities
- The entity instances returned may not reflect the latest values over a session

String sql = "select lane, city from address where city like ?";

```
ses.createSQLQuery(sql)
.setResultTransformer(Transformers.
aliasToBean(Address.class)).
setString(0, name)
.list();
```

Named SQL Query

- SQL queries can be written externally and accessed in the code
- Named queries allow more flexibility to modify queries

```
<sql-query name="cityQuery">

    <return-scalar column="city" type="string"/>
    <return-scalar column="lane" type="string"/>
    <![CDATA[
    SELECT a.city AS city,
          a.lane AS lane
    FROM Address a WHERE a.city LIKE :cityName
    ]]>
</sql-query>
```

Stored Procedure

- Hibernate 3 introduces support for queries via stored procedures and functions
- The stored procedure/function must return a resultset as the first out-parameter to be able to work with Hibernate.

Stored Procedure

```
CREATE OR REPLACE FUNCTION selectAllEmployments
   RETURN SYS_REFCURSOR
   AS
   st_cursor SYS_REFCURSOR;
BEGIN
    OPEN st_cursor FOR
    SELECT EMPLOYEE, EMPLOYER,
    STARTDATE, ENDDATE,
    REGIONCODE, EID, VALUE, CURRENCY
    FROM EMPLOYMENT;
RETURN st_cursor;
END;
```

Stored Procedure

- To use the stored procedure, map it through a named query

```
<sql-query name="selectAllEmployees_SP" callable="true">
    <return alias="emp" class="Employment">
    <return-property name="employee" column="EMPLOYEE"/>
    <return-property name="employer" column="EMPLOYER"/>
    <return-property name="startDate" column="STARTDATE"/>
    <return-property name="endDate" column="ENDDATE"/>
    <return-property name="regionCode" column="REGIONCODE"/>
    <return-property name="id" column="EID"/>
    <return-property name="salary">
    <return-column name="VALUE"/>
    <return-column name="CURRENCY"/>
    </return-property>
    </return>
    { ? = call selectAllEmployments() }
</sql-query>
```

Custom SQL

- Hibernate3 can use custom SQL statements for create, update, and delete operations
- The class and collection persisters in Hibernate already contain a set of configuration time generated strings (insertsql, deletesql, updatesql etc)
- The mapping tags <sql-insert>, <sql-delete>, and <sql-update> override these strings

Custom SQL

- The custom SQL script can also invoke a stored procedure if any
  - <sql-insert callable="true">{call createUser (?, ?)}</sql-insert>
  - <sql-delete callable="true">{? = call deleteUser (?)}</sql-delete>
- The positioning of parameters are important and should be as per what hibernate expects

Caching

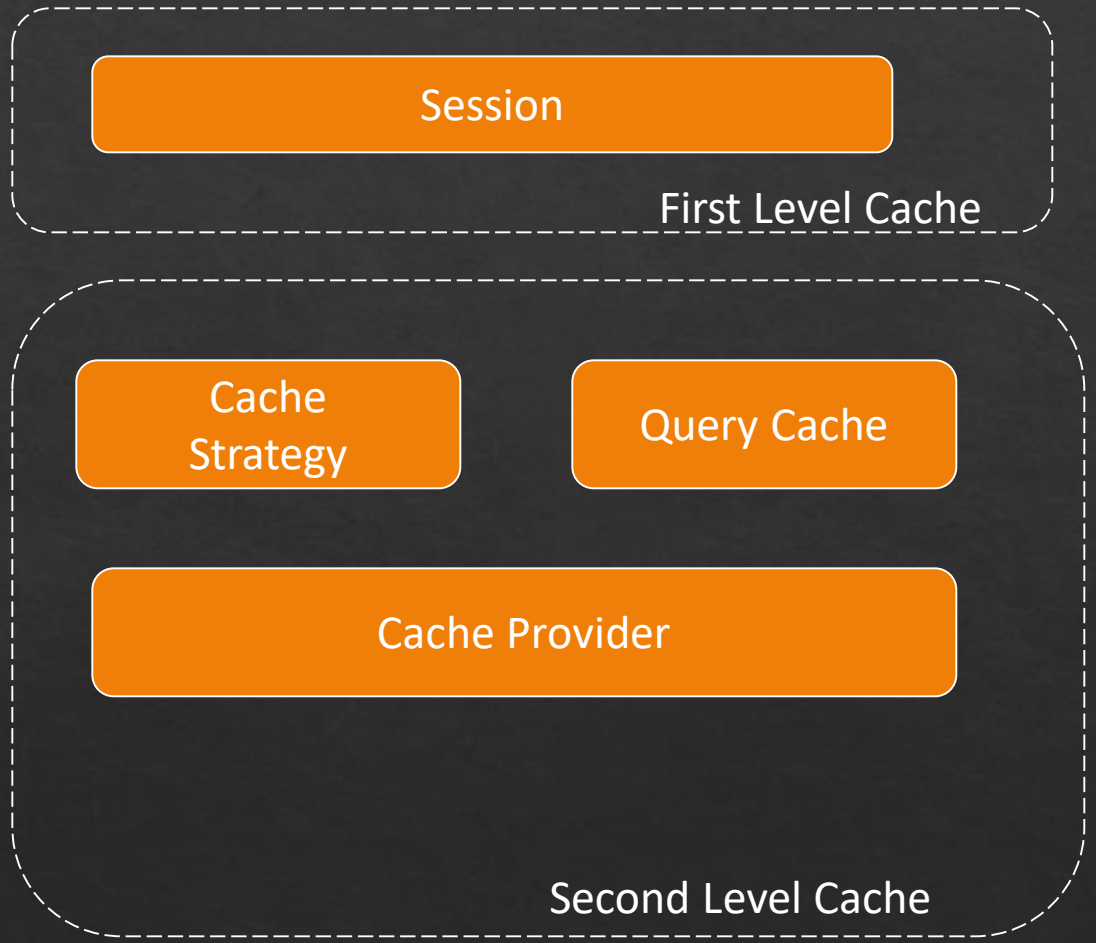
Hibernate Cache

- One of the benefits of using Hibernate is its ability to cache the instances
- Caching of data in entity instances can provide great performance benefits
- Hibernate provides configurable caching, where each class can be separately enabled for caching depending on its usage
- Hibernate provides caching at multiple levels

Hibernate Cache Architecture

- The hibernate cache provides caching at two levels
- The first level cache is the Session Cache which is enabled by default and can not be disabled
- The second level cached can be chosen between
  - Process Cache
  - Clustered Cache

Hibernate Cache architecture



First Level Cache

- All the persistence instances that are in Persistent state stay in the First Level Cache
- Changes made in a unit of work are cached in the first level cache
- Changes made to object are written to DB only on flush()
- You can opt to not keep data in first level cache by using evict() method
- Not useful for application that deal with bulk or large quantity of data. In such case, we are better off using plain SQL or stored procedures

Second Level Cache

- The second level cache is at a higher level and allow data to be shared across the application if opting for Process cache
- The cluster cache is across multiple servers and relies on remote communication to synchronize the cache
- The second level cache stores data in the form of Values

Second Level Cache

- A Class can be enabled for caching by indicating it in the mapping file

```
<class name="BillingDetails“  table="Billing_Details">
<cache usage="read-write"/>
```

References

Books

- Hibernate in Action
- Java Persistence with Hibernate

Online Tutorials

- https://www.tutorialspoint.com/hibernate/
- https://www.journaldev.com/3793/hibernate-tutorial

Created By Anil Joseph(anil.jos@gmail.com)

Training Enquiries

ANIL.JOS@GMAIL.COM

+91 98331 69784

+91 9833 169784

Created By Anil Joseph(anil.jos@gmail.com)

Thank You

Created By Anil Joseph(anil.jos@gmail.com)