HIBERNATE 5.0

Rajeev Gupta M. Tech CS
Java Trainer & Consultant

Workshop topics

- **Hibernate-What it is ?**
- **JPA- What it is?**
- **ORM and Issues**
- **Hibernate Hello World CRUD**
- **Primary key generation strategy**
- **More annotations**
- **Hibernate Object life cycle**
- **Hibernate Architecture**
- **Relation mapping**
  - **Many-to-one mapping**
  - **one-to-one mapping**
  - **Many-to-Many mapping**
- **Inheritance in Hibernate**
  - **Single Table Strategy**
  - **Table Per Class Strategy**
  - **Joined Strategy**
- **HQL and the Query Object**
  - one class to two tables
  - Select and Pagination in HQL
  - Named Quaries
  - Criteria API
- **Hibernate caching**
- **Hibernate JPA**
- **Hibernate performance**

Workshop topics

- **Hibernate-What it is ?**
- **JPA- What it is?**
- **ORM and Issues**
- **Hibernate Hello World CRUD**
- **Primary key generation strategy**
- **More annotations**
- **Hibernate Object life cycle**
- **Hibernate Architecture**
- **Relation mapping**
  - **Many-to-one mapping**
  - **one-to-one mapping**
  - **Many-to-Many mapping**
- **Inheritance in Hibernate**
  - **Single Table Strategy**
  - **Table Per Class Strategy**
  - **Joined Strategy**
- **HQL and the Query Object**
  - one class to two tables
  - Select and Pagination in HQL
  - Named Quaries
  - Criteria API
- **Hibernate caching**
  - First Level
  - Second Level Cache
  - **Hibernate optimization**
  - **Hibernate Batch Processing**

Gavin King

Öffentlich geteilt - 10.12.2013

#Hibernate

Just because you're using Hibernate, doesn't mean you have to use it for *everything*. A point I've been making for about ten years now.

Hibernate sucks!

... because it's slow

'The problem is sort of cultural [..] developers use Hibernate because they are *uncomfortable* with SQL and with RDBMSes. You should be very comfortable with SQL and JDBC before you start using Hibernate - Hibernate builds on JDBC, it doesn't replace it. That is the cost of extra abstraction [..] save yourself effort, **pay attention to the database at all stages of development.'**

- Gavin King (creator)

What is Hibernate?

- Hibernate is an ORM (Object Relational Mapping) tool.
- Maps the Relational Model in the database to the Object Model in Java.
- Tables are mapped to classes.
- Columns are mapped to JavaBean properties.
- Replaces SQL with HQL, a database-independent query language that navigates object relations rather than table

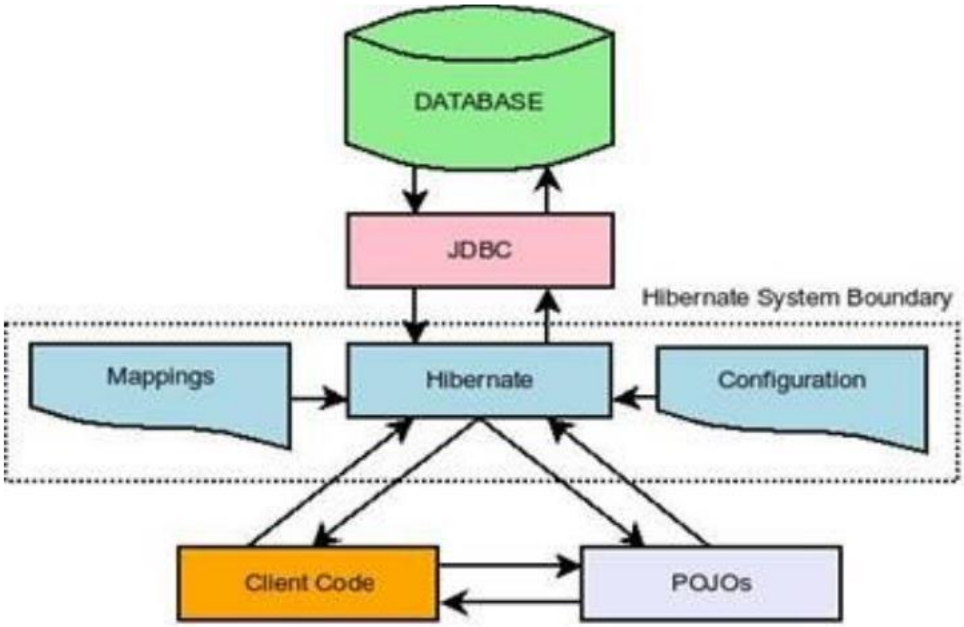
Why Hibernate?

JDBC is the bike of the persistence world. It's great for what it does, and for some jobs it works fine. But as our applications become more complex, so do our persistence requirements. We need to be able to map object properties to database columns and have our statements and queries created for us, freeing us from typing an endless string of question marks. We also need features that are more sophisticated:

- *Lazy loading*—As our object graphs become more complex, we sometimes don't want to fetch entire relationships immediately. To use a typical example, suppose we're selecting a collection of `PurchaseOrder` objects, and each of these objects contains a collection of `LineItem` objects. If we're only interested in `PurchaseOrder` attributes, it makes no sense to grab the `LineItem` data. This could be expensive. Lazy loading allows us to grab data only as it's needed.
- *Eager fetching*—This is the opposite of lazy loading. Eager fetching allows you to grab an entire object graph in one query. In the cases where we know that we need a `PurchaseOrder` object and its associated `LineItems`, eager fetching lets us get this from the database in one operation, saving us from costly round-trips.
- *Cascading*—Sometimes changes to a database table should result in changes to other tables as well. Going back to our purchase order example, when an `Order` object is deleted, we also want to delete the associated `LineItems` from the database.

If ORM is the Solution, is JDBC a Problem?

- No, no, JDBC is perfectly alright
- ORM provides a layer of convenience for the programmer.
- Does this convenience come with a price of performance?
- ORM sits in between your application and JDBC providing the missing link between the object oriented model and relational database model of programming. In fact this so called ORM interacts with the JDBC to talk to the database ultimately.

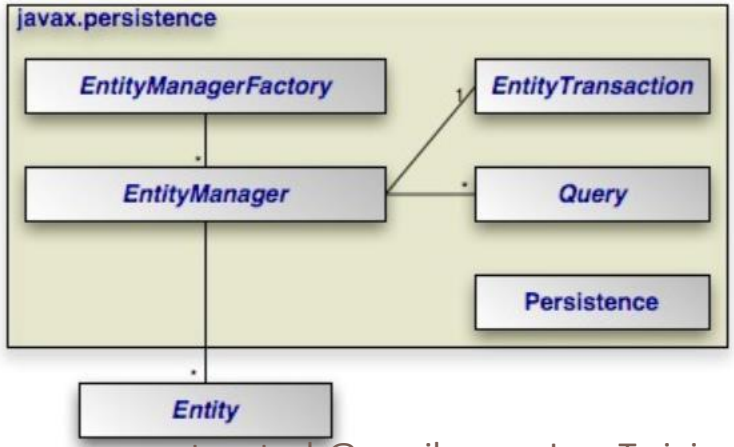


Workshop topics

- Hibernate-What it is ?
- JPA- What it is?
- ORM and Issues
- Hibernate Hello World CRUD
- Primary key generation strategy
- More annotations
- Hibernate Object life cycle
- Hibernate Architecture
- Relation mapping
  - Many-to-one mapping
  - one-to-one mapping
  - Many-to-Many mapping
- Inheritance in Hibernate
  - Single Table Strategy
  - Table Per Class Strategy
  - Joined Strategy
- **HQL and the Query Object**
  - one class to two tables
  - Select and Pagination in HQL
  - Named Quaries
  - Criteria API
- **Hibernate caching**
  - First Level
  - Second Level Cache
  - **Hibernate optimization**
  - **Hibernate Batch Processing**

What is JPA?

- JPA is the Java Persistence API, the entity persistence model for EJB 3.0
- Standardized persistence framework which is implemented by Hibernate (or TopLink, Cayenne, etc.)
- JPA Annotations and persistence.xml provide vendor independent configuration
- EntityManager provides vendor independent access to persistence
- Replaces vendor specific query languages (HQL) with standard



Why JPA?

- JPA is the standard, and standards are good!
- Using JPA does not tie you to Hibernate.
- JPA gives you most of the features of plain old Hibernate, except:
  - No criteria queries in JPA 2.0
    - Criteria query is a neat feature of Hibernate that constructs query using Java-based combinators instead of alternate query language, getting the benefit of IntelliSense and Eclipse's refactoring tools.
  - JPA doesn't have Hibernate's DeleteOrphan cascade type
    - Delete Orphan is a useful annotation that directs Hibernate to deletes entities in a collection if the parent is deleted, preventing orphaning.
  - JPA doesn't have an equivalent to Hibernate's ScrollableResults
- But, all of these features are accessible to an otherwise fully JPA application!

Workshop topics

- Hibernate-What it is ?
- JPA- What it is?
- ORM and Issues
- Hibernate Hello World CRUD
- Primary key generation strategy
- More annotations
- Hibernate Object life cycle
- Hibernate Architecture
- Relation mapping
  - Many-to-one mapping
  - one-to-one mapping
  - Many-to-Many mapping
- Inheritance in Hibernate
  - Single Table Strategy
  - Table Per Class Strategy
  - Joined Strategy
- **HQL and the Query Object**
  - one class to two tables
  - Select and Pagination in HQL
  - Named Quaries
  - Criteria API
- **Hibernate caching**
  - First Level
  - Second Level Cache
  - **Hibernate optimization**
  - **Hibernate Batch Processing**

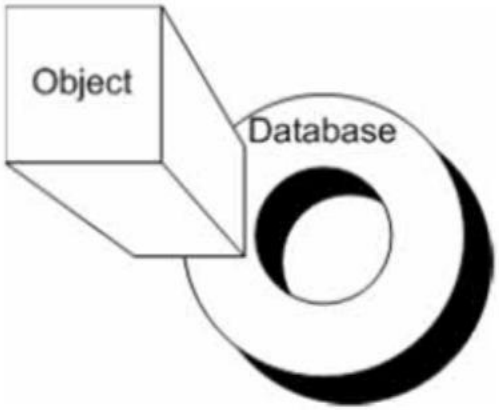
ORM

Object Relational

- Mapping object to relational world is not easy, as there are points of mismatches

- There are numbers of impedance mismatches between OO world and DB worlds, and ORM tool must address and provide solutions for that

- Identity
- Granularity
- Associations
- Navigation
- Inheritance
- Data type mismatch



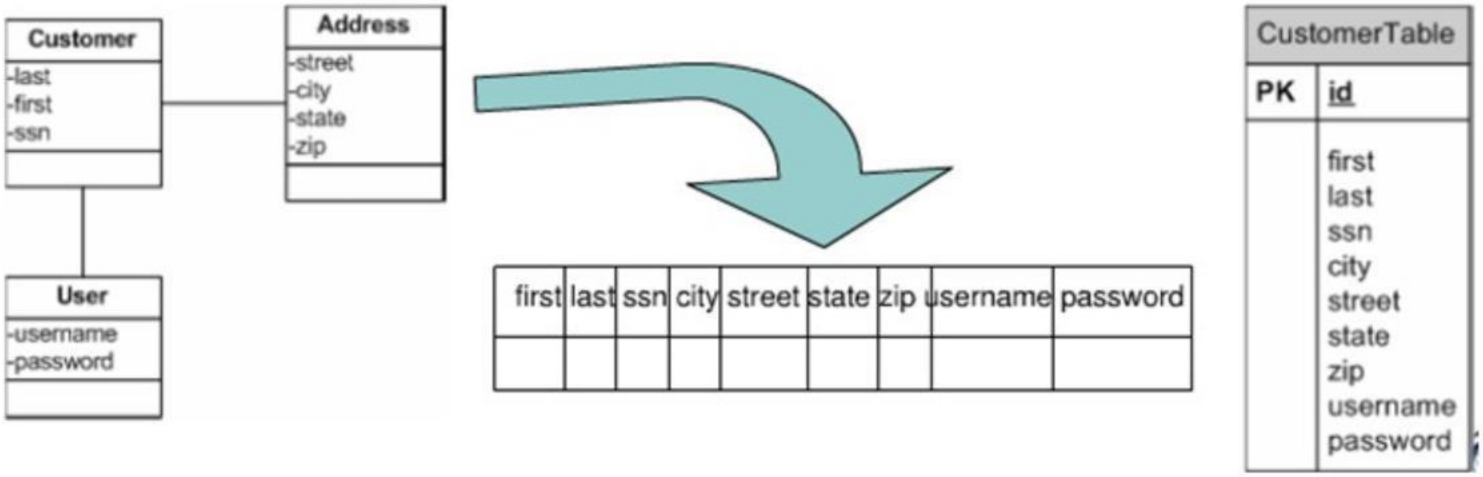
Identity

- A row is uniquely identified form all other rows by its primary key
- An object's identity does not always translate well to the concept of primary key of DB world
- In Java, What is the meaning of identity of an object?
- Its data or its place in memory?

```
accountA.equals(accountB);
accountA==accountB
```

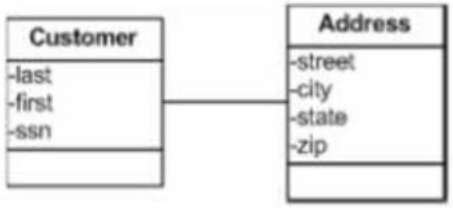
Granularity

- Objects and tables are at different levels of granularity
- Table structure are often de-normalized in order to support better performance, so table rows can map to multiple objects, how to map them?



Associations

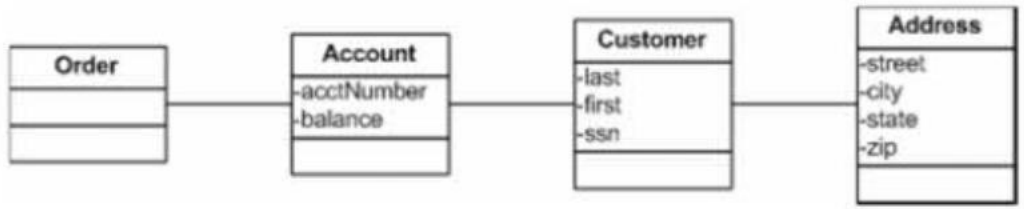
- Associations in Java are either unidirectional or bidirectional and can be represented by one object having pointer of other object.
- Relation between database tables is based on join. table relationship is always bidirectional, while for object it may be unidirectional/bidirectional



Options to consider
Does address know the customer?
Does the customer have >1 address?
Can an address be owned by >1 customer?

Navigation and associations traversal

- Navigating Java Objects to get property information often require traversal of the object graph.
  - This can be quite expensive if the objects are distributed or need to be created before traversed.
  - Consider getting the address to ship and order



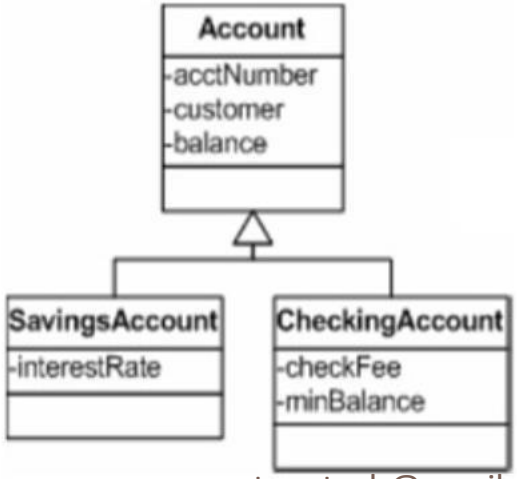

```
anAcct = anOrder.getAccount();
aCust = anAcct.getCustomer();
orderAddress = aCust.getAddress();
shipTo(orderAddress);
```

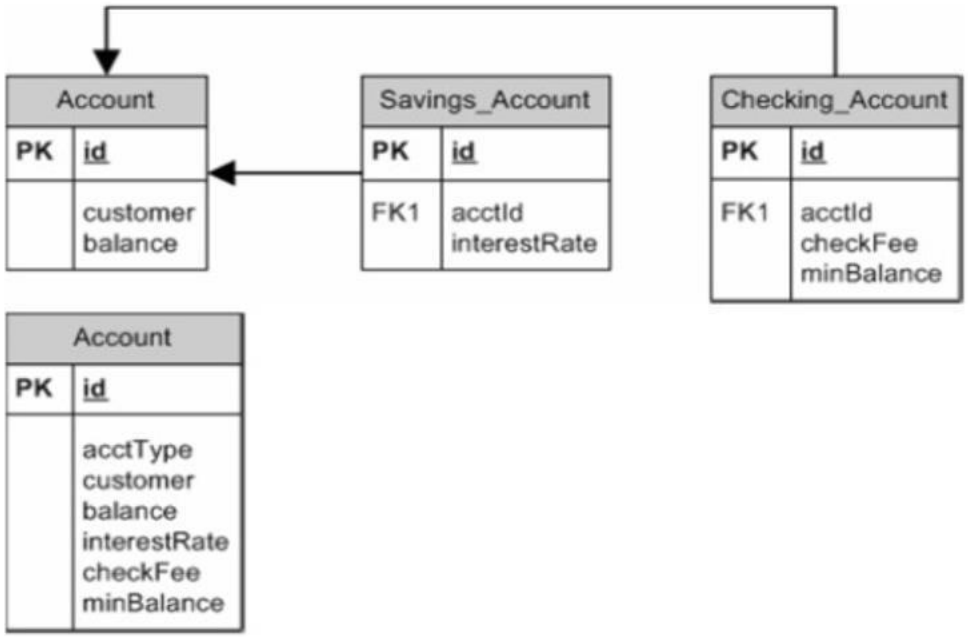
- While navigation in database is handled with the help of single join.

Inheritance

- Inheritance and Polymorphism are important concepts in OO world
- Database don't have equivalent concepts
- Now question is how to map it to database world?

Inheritance examples



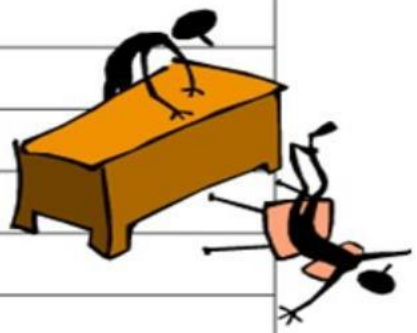


Data Types

- There is mismatch between database data type and object oriented data types.
- Question is how ORM tools can help us to map is correctly?

Some Java Data Types vs. SQL Data Types

Java Data Type	*SQL Data Type*
String	VARCHAR
String	CHARACTER
String	LONGVARCHAR
BigDecimal	NUMERIC
BigDecimal	DECIMAL
Boolean, boolean	BIT
Integer, byte	TINYINT
Integer, short	SMALLINT
Integer, long	BIGINT



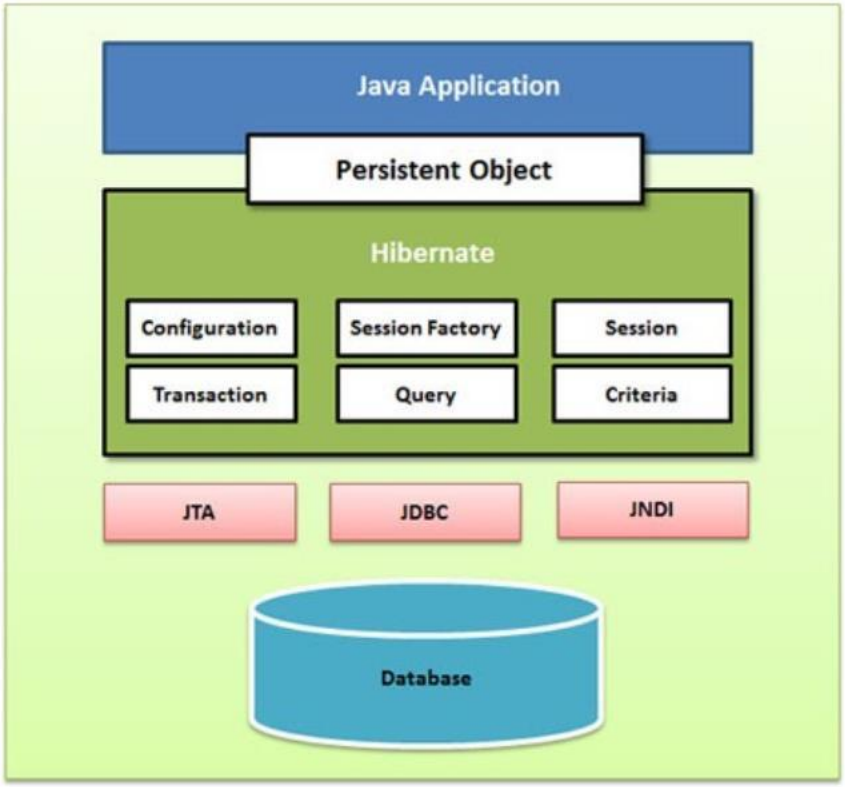
Workshop topics

- **Hibernate-What it is ?**
- **JPA- What it is?**
- **ORM and Issues**
- **Hibernate Hello World CRUD**
- **Primary key generation strategy**
- **More annotations**
- **Hibernate Object life cycle**
- **Hibernate Architecture**
- **Relation mapping**
  - **Many-to-one mapping**
  - **one-to-one mapping**
  - **Many-to-Many mapping**
- **Inheritance in Hibernate**
  - **Single Table Strategy**
  - **Table Per Class Strategy**
  - **Joined Strategy**
- **HQL and the Query Object**
  - one class to two tables
  - Select and Pagination in HQL
  - Named Quaries
  - Criteria API
- **Hibernate caching**
  - First Level
  - Second Level Cache
  - **Hibernate optimization**
  - **Hibernate Batch Processing**

Hibernate Hello World

Steps

1. Write a POJO class representing the table
2. Write a hbm file
3. Write cfg file/ write annotated POJO class
4. Write a class to test CRUD code



Write a Annotated POJO class

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

@Entity
public class Customer {
    @Id
    private int customerId;
    private String customerName;
    private String customerAddess;
    public int getCustomerId() {
        return customerId;
    }
    public void setCustomerId(int customerId) {
        this.customerId = customerId;
    }
    public String getCustomerName() {
        return customerName;
    }
```

Note: annotations can also be applied to getters

Rule to be Entity

Entity class must be:

1. Annotated with @javax.persistence.Entity
2. @javax.persistence.Id annotation must be used to denote primary key
3. Must have a no-arg constructor that has to be public or protected.
4. Must be a top-level class.
5. Entity class must not be final.
6. No methods or persistent instance variables of the entity class may be final
7. May implements Serializable interface

Hibernate.cfg.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- database connection setting -->
        <property name="connection.url">jdbc:mysql://localhost:3306/fooraj123</property>
        <property name="connection.username">root</property>
        <property name="connection.password">root</property>
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</property>

        <!-- disable the second level cache -->
        <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
        <property name="show_sql">true</property>

        <property name="format_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">update</property>
        <!-- jdbc connection pool build in  -->
        <property name="connection.pool_size">1</property>
        <!-- <property name="current_session_context_class">thread</property> -->
        <mapping class="com.demo.Customer" />

    </session-factory>
</hibernate-configuration>
```

Hibernate Dependencies

```
<dependencies>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>6.0.5</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.2.9.Final</version>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>2.8.2</version>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-api</artifactId>
  <version>2.8.2</version>
</dependency>
</dependencies>
```

<property name="current_session_context_class">thread</property>

As explained in this forum post, 1 and 2 are related. If you set `hibernate.current_session_context_class` to thread and then implement something like a servlet filter that opens the session - then you can access that session anywhere else by using the `SessionFactory.getCurrentSession()`.

`SessionFactory.openSession()` always opens a new session that you have to close once you are done with the operations. `SessionFactory.getCurrentSession()` returns a session bound to a context - you don't need to close this.

If you are using Spring or EJBs to manage transactions you can configure them to open / close sessions along with the transactions.

You should never use "one session per web app" - session is not a thread safe object - cannot be shared by multiple threads. You should always use "one session per request" or "one session per transaction"

- http://stackoverflow.com/questions/8046662/hibernate-opensession-vs-getcurrentsession
- https://forum.hibernate.org/viewtopic.php?p=2384979&sid=8367751b54bf160003b867f858393398#p2384979

Adding record

```
SessionFactory factory = new MetadataSources(
        new StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build())
        .buildMetadata()
                .buildSessionFactory();

Session session=factory.openSession();
Transaction tx=session.getTransaction();
try {
    tx.begin();
    Customer customer=new Customer();
    customer.setCustomerName("ravi");
    customer.setCustomerAddess("delhi");
    session.save(customer);
    tx.commit();
}catch(HibernateException ex) {
    ex.printStackTrace();
    tx.rollback();
}
```

```
mysql> select * from customer;
+------------+-----------------+--------------+
| customerId | customerAddess  | customerName |
+------------+-----------------+--------------+
|        121 | delhi           | ravi         |
+------------+-----------------+--------------+
```

get & Update record

```
Session session=factory.getCurrentSession();

session.beginTransaction();
Customer customer=(Customer) session.get(Customer.class, 121);
customer.setCustomerAddess("noida");
session.update(customer);
session.getTransaction().commit();
```

```
Session session=factory.getCurrentSession();

session.beginTransaction();
Customer customer=(Customer) session.get(Customer.class, 121);
customer.setCustomerAddess("noida");
session.update(customer);
session.getTransaction().commit();
```

Display all records

```
Session session=factory.getCurrentSession();

session.beginTransaction();
List<Customer>cList=session.createQuery("from Customer").list();

for(Customer c:cList)
    System.out.println(c);
session.getTransaction().commit();
```

```
Hibernate:
    select
        customer0_.customerId as customerId0_,
        customer0_.customerAddess as customer2_0_,
        customer0_.customerName as customer3_0_
    from
        Customer customer0_
Customer [customerId=121, customerName=ravi, customerAddess=noida]
```

Workshop topics

- **Hibernate-What it is ?**
- **JPA- What it is?**
- **ORM and Issues**
- **Hibernate Hello World CRUD**
- **Primary key generation strategy**
- **More annotations**
- **Hibernate Object life cycle**
- **Hibernate Architecture**
- **Relation mapping**
  - **Many-to-one mapping**
  - **one-to-one mapping**
  - **Many-to-Many mapping**
- **Inheritance in Hibernate**
  - **Single Table Strategy**
  - **Table Per Class Strategy**
  - **Joined Strategy**
- **HQL and the Query Object**
  - one class to two tables
  - Select and Pagination in HQL
  - Named Quaries
  - Criteria API
- **Hibernate caching**
  - First Level
  - Second Level Cache
  - **Hibernate optimization**
  - **Hibernate Batch Processing**

Primary Keys with @Id and @GeneratedValue

- @GeneratedValue annotation is used to decide primary key generation strategies
- This takes a pair of attributes: strategy and generator. The strategy attribute must be a value from the javax.persistence.GeneratorType enumeration. If you do not specify a generator type, the default is AUTO

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy=)
    private int customerId;
    private String customerNa
    private String customerAd
    public int getCustomerId(
        return customerId;
    }
    public void setCustomerId
        this.customerId = cus
    }
```

AUTO : GenerationType - GenerationType
IDENTITY : GenerationType - GenerationType
SEQUENCE : GenerationType - GenerationType
TABLE : GenerationType - GenerationType
GenerationType - javax.persistence
Customer - com.model
customerAddess : String - Customer
customerId : int - Customer
customerName : String - Customer

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int customerId;
```

```
@Entity
public class Customer {
    @Id
    @TableGenerator (name="my_gen",table="customerpktable",pkColumnName="customerkey",
            pkColumnValue="customervalue",allocationSize=1)

    @GeneratedValue(strategy=GenerationType.TABLE,generator="my_gen")
```

Compound Primary Keys

- What if we need some business primary key?
- You must create a class to represent the primary key.
- Condition to be primary key class:
  - It must be a public class
  - It must have a default constructor
  - It must be serializable
  - It must implement hashCode() and equals() methods to allow the Hibernate code
  - **Compound Primary Keys with @Id, @IdClass, or @EmbeddedId**
  - **Consider that business requirment is that we need to represent customer primary key as a combination of customerId and**

Your three strategies for using this primary key class once it has been created are as follows:

- Mark it as `@Embeddable` and add to your entity class a normal property for it, marked with `@Id`.
- Add to your entity class a normal property for it, marked with `@EmbeddableId`.
- Add properties to your entity class for all of its fields, mark them with `@Id`, and mark your entity class with `@IdClass`, supplying the class of your primary key class.

Compound Primary Keys with @Id

1. Create an POJO annotated with @Embeddable (it must implements Serilizable interface and must have default

```
@Embeddable
public class CustomerKey implements Serializable{
    private static final long serialVersionUID = -4336329019606358832L;
    private int customerId;
    private String customerRegistrationId;
    public int getCustomerId() {
        return customerId;
    }
```

2. Use it in target class

```
@Entity
public class Customer {
    @Id
    private CustomerKey customerKey;
    private String customerName;
    private String customerAddess;
```

Compound Primary Keys @EmbeddedId

- Here, the primary key class cannot be used in other tables since it is not an @Embeddable entity, but it does allow us to treat the key as a single attribute of the Account class

```
@Entity
public class Customer {
    @EmbeddedId
    private CustomerKey customerKey;
    private String customerName;
    private String customerAddess;
```

- No need to mention @Embeddable on CustomerKey class

```
@Embeddable
public class CustomerKey implements Serializable{
    private static final long serialVersionUID = -4336329019606358832L;
    private int customerId;
    private String customerRegistrationId;
    public int getCustomerId() {
        return customerId;
    }
```

Workshop topics

- Hibernate-What it is ?
- JPA- What it is?
- ORM and Issues
- Hibernate Hello World CRUD
- Primary key generation strategy
- More annotations
- Hibernate Object life cycle
- Hibernate Architecture
- Relation mapping
  - Many-to-one mapping
  - one-to-one mapping
  - Many-to-Many mapping
- Inheritance in Hibernate
  - Single Table Strategy
  - Table Per Class Strategy
  - Joined Strategy
- HQL and the Query Object
  - one class to two tables
  - Select and Pagination in HQL
  - Named Quaries
  - Criteria API
- Hibernate caching
  - First Level
  - Second Level Cache
  - Hibernate optimization
  - Hibernate Batch Processing

More Annotations

- What if table name and field name is something else?
  - Use annotation @Table,@Column
  - @Tomporal
- @Lob?
- @Tomporal?
  - Storing date and time.
  - What if i want only to store date part not time?
    - @Temporal(TemporalType.DATE)
- @Transient?
  - You can not store static and transient data.
- @Basic ?
  - Used to add some properties to add to that field.
  - You can work apply without it.

```
@Entity(name="customer_entity")
@Table(name="customer_table")
public class Customer {
    @Id
    @Column(name="customer_Id")
    private int customerId;
    @Column(name="customer_Name")
    private String customerName;
    @Column(name="customer_Addess")
    private String customerAddess;
```

What is the difference bw applying name using @Table and @Entity

```
@Temporal(TemporalType.DATE)
private Date customerDob;
@Transient
private String customerPassword;
```

@Enumerated

```
public enum CreditCardType {
    VISA,
    MASTER_CARD,
    AMERICAN_EXPRESS
}
```

```
@Entity
@Table(name = "credit_card")
public class CreditCard {

    @Id
    private String number;
    private String expiryDate;
    private Integer controlNumber;
    private CreditCardType creditCardType;

    // Constructors, getters, setters
}
```

Mapping an Enumerated Type with `String`

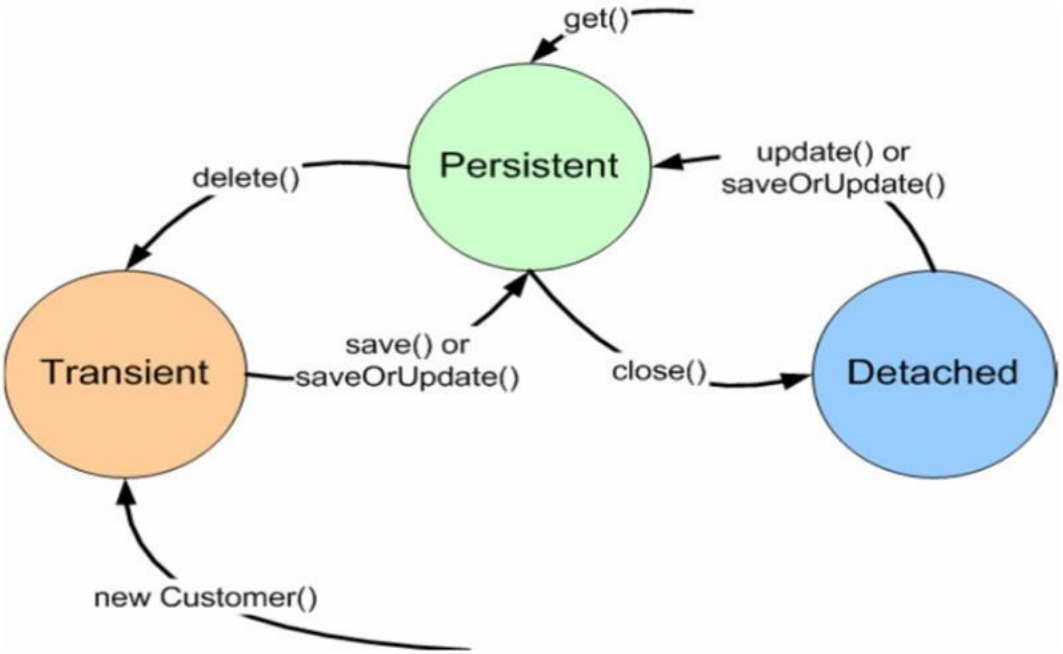
```
@Entity
@Table(name = "credit_card")
public class CreditCard {

    @Id
    private String number;
    private String expiryDate;
    private Integer controlNumber;
    @Enumerated(EnumType.STRING)
    private CreditCardType creditCardType;

    // Constructors, getters, setters
}
```

Workshop topics

- Hibernate-What it is ?
- JPA- What it is?
- ORM and Issues
- Hibernate Hello World CRUD
- Primary key generation strategy
- More annotations
- Hibernate Object life cycle
- Hibernate Architecture
- Relation mapping
  - Many-to-one mapping
  - one-to-one mapping
  - Many-to-Many mapping
- Inheritance in Hibernate
  - Single Table Strategy
  - Table Per Class Strategy
  - Joined Strategy
- HQL and the Query Object
  - one class to two tables
  - Select and Pagination in HQL
  - Named Quaries
  - Criteria API
- Hibernate caching
  - First Level
  - Second Level Cache
- Hibernate optimization
- Hibernate Batch Processing

Object lifecycle

- Given an instance of an object that is mapped to Hibernate, it can be in any one of three states:-
  - Transient
  - Persistence
  - Detached



More on Object Life Cycle

- **Transient objects** exist in memory, Hibernate does not manage transient objects or persist changes to transient objects. Hibernate Don't care about that object
- **Persistent objects** exist in the database, and Hibernate manages the persistence for persistent objects. If fields or properties change on a persistent object, Hibernate will keep the database representation up-to-date.
- **Detached objects** have a representation in the database, but changes to the object will not be reflected in the database, and vice versa.
- A detached object can be created by closing the session that it was associated with, or by evicting it from the session with a call to the session's evict() method.

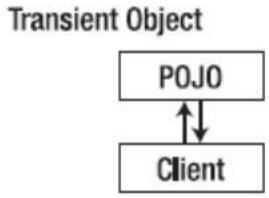


Figure 4-1. *Transient objects are independent of Hibernate.*

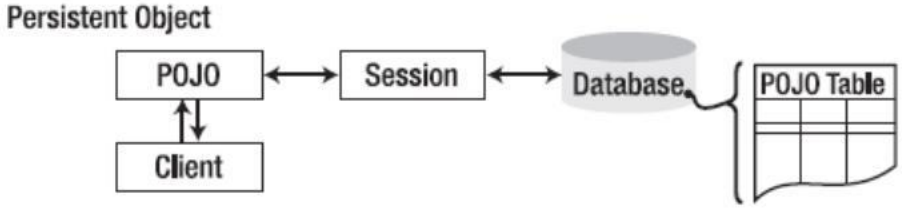


Figure 4-2. *Persistent objects are maintained by Hibernate.*

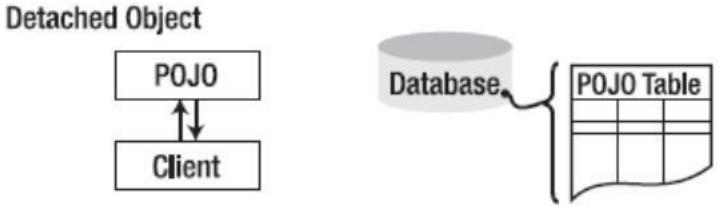
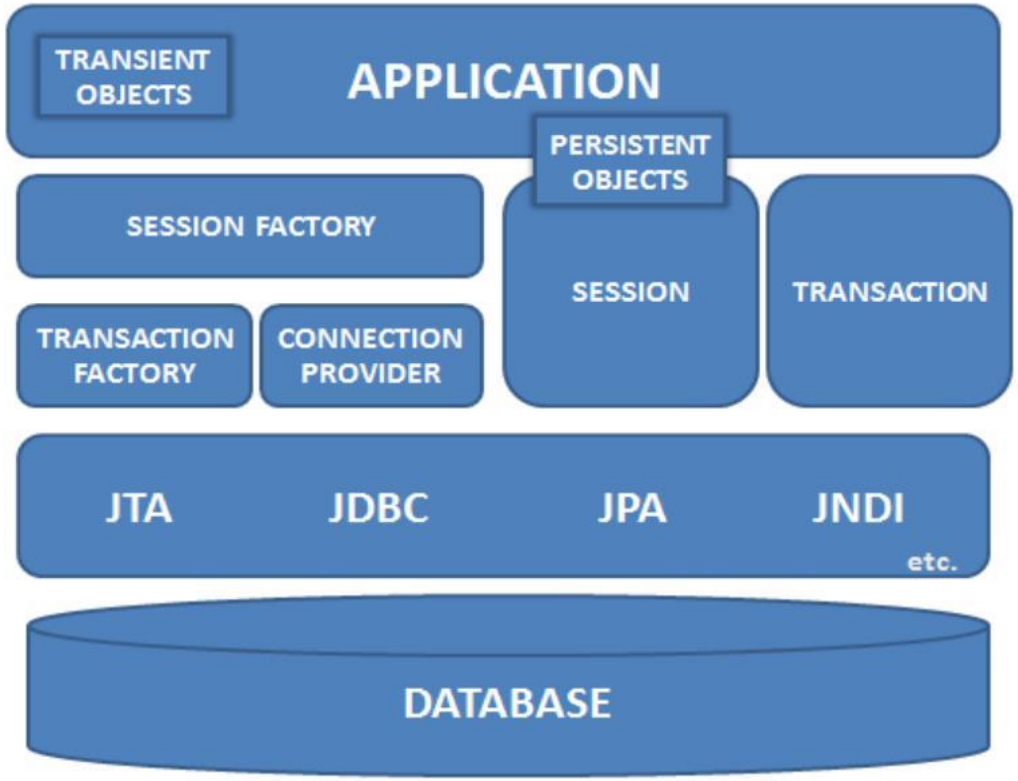


Figure 4-3. *Detached objects exist in the database but are not maintained by Hibernate.*

Workshop topics

- **Hibernate-What it is ?**
- **JPA- What it is?**
- **ORM and Issues**
- **Hibernate Hello World CRUD**
- **Primary key generation strategy**
- **More annotations**
- **Hibernate Object life cycle**
- **Hibernate Architecture**
- **Relation mapping**
  - **Many-to-one mapping**
  - **one-to-one mapping**
  - **Many-to-Many mapping**
- **Inheritance in Hibernate**
  - **Single Table Strategy**
  - **Table Per Class Strategy**
  - **Joined Strategy**
- **HQL and the Query Object**
  - one class to two tables
  - Select and Pagination in HQL
  - Named Quaries
  - Criteria API
- **Hibernate caching**
  - First Level
  - Second Level Cache
  - **Hibernate optimization**
  - **Hibernate Batch Processing**

Hibernate Architecture



Core Interfaces

- Used to perform basic CRUD and querying operations
  - **Session**
  - **SessionFactory**
  - **Configuration**
  - **Transaction interface**
  - **Query**
  - **Criteria**

SessionFactory

- The application obtains **Session** instances from a **SessionFactory** interface
- The SessionFactory is created from an configuration object.

```
SessionFactory factory=new Configuration().configure().buildSessionFactory();
```

- The SessionFactory is an expensive object to create
- It too is created at application start-up time
  - It should be created once and kept for latter use.
  - The SessionFactory object is used by all the threads of an applications
  - it is thread safe object
  - one SessionFactory object is created per database (where connecting to multiple sessions)

- The SessionFactory is used to create Session Objects

Session

- The Session object is created from the SessionFactory object

```
SessionFactory factory=new Configuration().configure().buildSessionFactory();
Session session=factory.openSession();
```

  - A Session object is lightweight and inexpensive to create.
  - Session object provides the main interface to accomplish work with the database
  - Does the work of getting a physical connection to the database (hopefully from a connection pool)
  - Session objects are not thread safe
  - Session objects should not be kept open for a long time
  - Application create and destroy these as needed. Typically they are created to complete a single unit of work.

- When modifications are made to the database, session objects are used to create a transaction object

Transaction

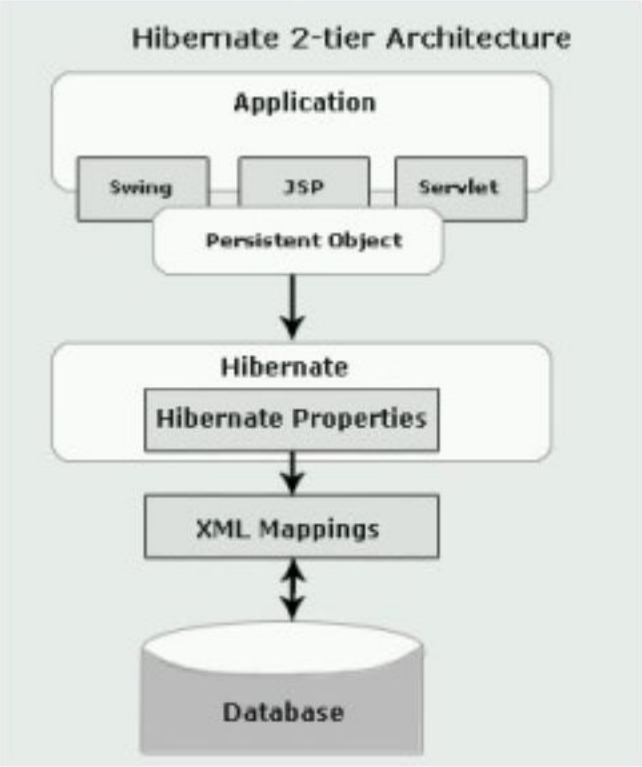
- Transaction objects are obtained from the session objects, when an modification to the database is needed.

```
Transaction trans=session.beginTransaction();
```

- The Transaction object provides abstraction for the underlying implementation
  - Hibernate with use whatever transaction implementation is available (JDBC, JTA etc.)
  - It is optional; allowing developer to use their own transactional infrastructure.
  - Transaction objects should be kept open for a short time.

How hibernate actually works with JDBC?

- By default Hibernate creates a proxy for each of the entity class in mapping file. This class contain the code to invoke JDBC.
- Proxies are created dynamically by sub-classing the entity object at runtime.
- The subclass has all the methods of the parent, so when a method on the entity object is called, the proxy loads up the data from the database and calls the method.



Persistent object states

- The POJO or persistent objects can be in one of the three states is defined in relation to a persistence context (that means it is loaded into the Hibernate Session object)
  - Transient
  - Persistent
  - Detached

Transient state

- The instance is not associated with any persistence context. It has no persistent identity or primary key value.
- Transient instances may be made persistent by calling save(), persist() or saveOrUpdate() method on Session object

save() vs pesist()

`Save()`

1. Returns generated Id after saving. Its return type is `Serializable`;
2. Saves the changes to the database outside of the transaction;
3. Assigns the generated id to the entity you are persisting;
4. `session.save()` for a detached object will create a new row in the table.

`Persist()`

1. Does not return generated Id after saving. Its return type is `void`;
2. Does not save the changes to the database outside of the transaction;
3. Assigns the generated Id to the entity you are persisting;
4. `session.persist()` for a detached object will throw a `PersistentObjectException`, as it is not allowed.

Different between get() and load()

- both functions are use to retrieve an object with different mechanism
- session.load()
  - It will always return a “proxy” (Hibernate term) without hitting the database.
  - proxy object look like a temporary fake object.
  - If no row found , it will throws an ObjectNotFoundException.
- session.get()
  - It always hit the database and return the real object,an object that represent the database row, not proxy.

Difference between the load() and get() methods is that while load() assumes that instance exists and may return a proxy and we cannot guarantee that instance actually exists in the database (it may have got deleted).

Hence get() must be used instead of load() to determine if an instance exists.

Also note that the load() will throw an exception if instance does not exist.

Detached State

- The instance was once associated with a persistence context/session but not is not attached to it may because the session was closed.
- It has a persistent identity and can have a corresponding row in the database.
- While for persistent instance Hibernate guarantees the relationship between persistent (database) identity and Java identity for detached instance there is no such guarantees .
- Detached instances may be made persistent by calling **update(), saveOrUpdate()**
- The state of a transient or detached instance may also be made persistent as a new persistent instance by calling **merge().**

update(),merge() differences

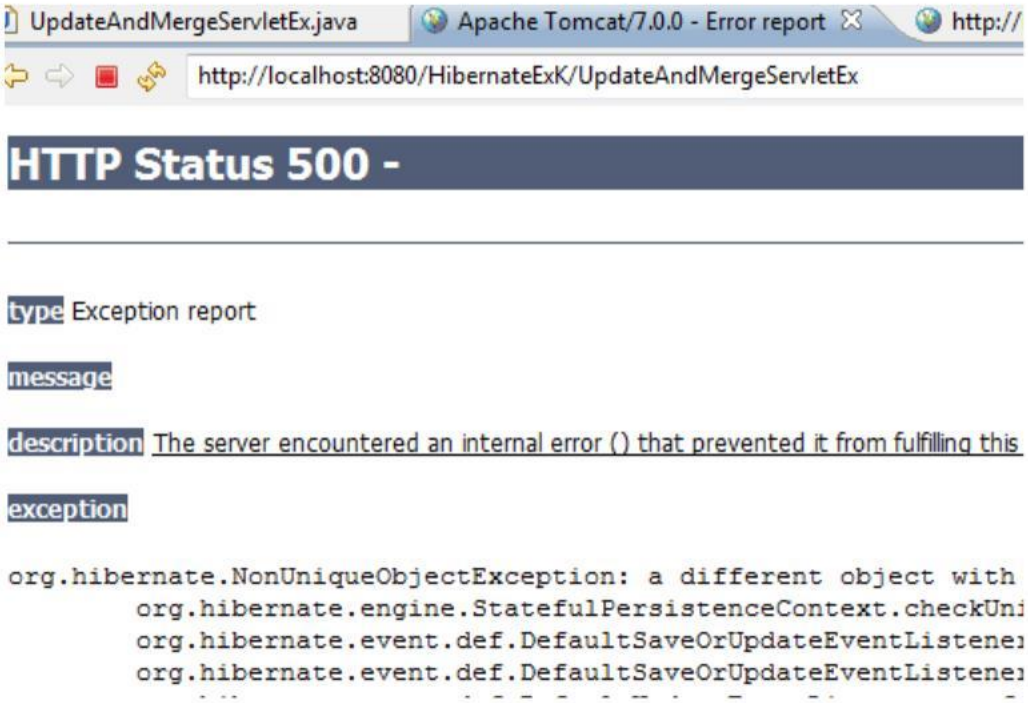
- **Update should be used to save the data when the session does not contain an already persistent instance with the same identifier.**

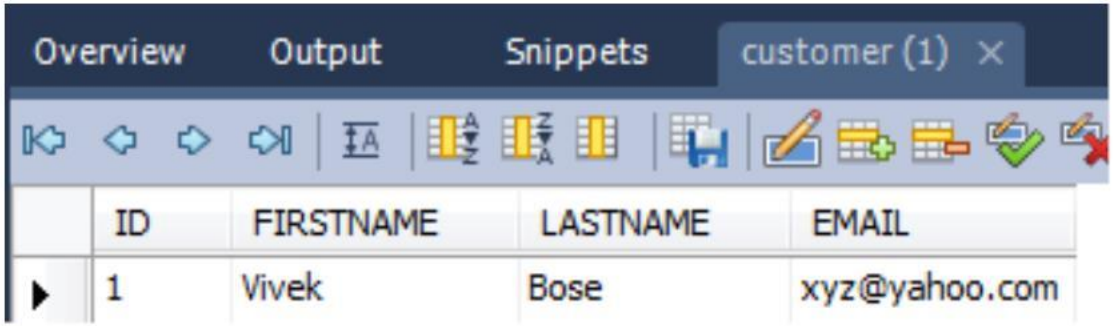
- **Merge should be used to save the modifications at any time without knowing about the state of a session**

- **To understand difference bw update() and merge() lets try to update values**

```
public class DemoUpdate extends HttpServlet {
    private static final long serialVersionUID = 1L;
    Session session = null;

    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {

        SessionFactory fac=new Configuration().configure().buildSessionFactory();
        Session session=fac.openSession();
        Customer c=(Customer) session.get(Customer.class, new Long(121));
        session.close();

        c.getId();
        c.setEmail("foo22.foo.com");
        Session session2=fac.openSession();
        Customer c1=(Customer) session2.get(Customer.class, new Long(121));
        session2.update(c);
        session2.flush();
        session2.close();
    }

}
```

will throws exception





When **session2.update(c);** **is** used instead of **session2.merge(c);**

This is if there is a persistent instance with the same identifier in the session, update() will throw an exception!

When **session2.merge(c);** is used the data gets updated because the instance that the session is holding gets its values from the detached instance when merge is called

delete() and refresh()

- **void delete(Object object) throws HibernateException**
  - Remove a persistent instance from the datastore. The argument may be an instance associated with the receiving Session or a transient instance with an identifier associated with existing persistent state. This operation cascades to associated instances if the association is mapped with cascade="delete".

- **void refresh(Object□object) throws HibernateException**

This method is useful to sync the state of the given instance with underlying database in cases where there are chances that database might have been altered outside the application as a result of a trigger. This is also useful in cases where direct SQL is used by the application to update the data.

flush(), close(), clear()

- **void flush() throws HibernateException**
  - This is the method that actually synchronizing the underlying database with persistent object held in session memory.
  - This should be called at the end of a unit of work, before committing the transaction and closing the session (depending on **flush-mode, Transaction.commit()** calls this method).
- **Connection close() throws HibernateException**
  - End the session by releasing the JDBC connection and cleaning up. It is not strictly necessary to close the session but at least it must be disconnected by calling **disconnect()**
- **void clear()**
  - Completely clear the session. Evict all loaded instances and cancel all pending saves, updates and deletions. Do not close open iterators or instances of **ScrollableResults**.

Transaction

- Transaction can be set using the methods of **Session**.

  Transaction beginTransaction() throws HibernateException

  If Transaction object is associated with the session return that else create a new transaction.

  Transaction getTransaction()

  Get the Transaction instance associated with this session

  In both the cases the class of the returned Transaction object is determined by the property **hibernate.transaction_factory.**

- **The most common approach is the session-per-request pattern where a single Hibernate Session has same scope as a single database transaction.**

- The Hibernate session can be used for multiple DB operations (save, query, update) within the same request.

Transaction methods

- **void begin() throws HibernateException**
- **void commit() throws HibernateException**
- **void rollback() throws HibernateException**
- To check if transactions was committed or rolled-back properly
- **boolean wasRolledBack() throws HibernateException**
- **boolean wasCommitted() throws HibernateException**

- Code snippet wrapping statements in a transactions:

```
Session sess = factory.openSession();
Transaction tx;
try
{
tx = sess.beginTransaction();
//do some work ...
tx.commit();
}
catch (Exception e)
 {
if (tx!=null) tx.rollback(); throw e; }
finally { sess.close(); }
```

Workshop topics

- Hibernate-What it is ?
- JPA- What it is?
- **ORM and Issues**
- Hibernate Hello World CRUD
- Primary key generation strategy
- More annotations
- Hibernate Object life cycle
- Hibernate Architecture
- **Component Mapping**
- Relation mapping
  - Many-to-one mapping
  - one-to-one mapping
  - Many-to-Many mapping
- Inheritance in Hibernate
  - Single Table Strategy
  - Table Per Class Strategy
  - Joined Strategy
- **HQL and the Query Object**
  - one class to two tables
  - Select and Pagination in HQL
  - Named Quaries
  - Criteria API
- **Hibernate caching**
  - First Level
  - Second Level Cache
  - **Hibernate optimization**
  - **Hibernate Batch Processing**

One class to two tables

```java
@Entity
public class Customer {
    //must be in one table lets say customer
    @Id
    private int customerId;
    private String customerName;

    /*
     * must be in aonther table lets say customer_details
     *  table and pk and fk should be working?
     */

    private String customerAddess;
    private int customerCreditScore;
    private int customerRewardPoints;


}
```

Let we want to store the object of customer in two different tables?

We have to decide what should be the tables names?

Lets say Customer and other other table CustomerDetails

We have to decide what fields should go in each tables?

One class to two tables: solutions

```java
@Entity
@Table(name="customer")
@SecondaryTable(name="customerDetails")
public class Customer {

    @Id
    private int customerId;
    private String customerName;

    //must be in customerDetails table
    @Column(table="customerDetails")
    private String customerAddess;

    @Column(table="customerDetails")
    private int customerCreditScore;

    @Column(table="customerDetails")
    private int customerRewardPoints;

    public int getCustomerId() {
        return customerId;
    }
```

-> SecondaryTable annotation is the key!!!

```
mysql> desc customer;
+--------------+--------------+------+-----+---------+-------+
| Field        | Type         | Null | Key | Default | Extra |
+--------------+--------------+------+-----+---------+-------+
| customerId   | int(11)      | NO   | PRI | NULL    |       |
| customerName | varchar(255) | YES  |     | NULL    |       |
+--------------+--------------+------+-----+---------+-------+
2 rows in set (0.01 sec)

mysql> desc customerdetails;
+---------------------+--------------+------+-----+---------+-------+
| Field               | Type         | Null | Key | Default | Extra |
+---------------------+--------------+------+-----+---------+-------+
| customerAddess      | varchar(255) | YES  |     | NULL    |       |
| customerCreditScore | int(11)      | YES  |     | NULL    |       |
| customerRewardPoints| int(11)      | YES  |     | NULL    |       |
| customerId          | int(11)      | NO   | PRI | NULL    |       |
+---------------------+--------------+------+-----+---------+-------+
```

One table from two classes

```
public class School {
    private int schoolId;
    private String schoolName;


    //putting getter and setters
```

```
public class SchoolDetails {

    private String schoolAddress;
    private boolean isPublicSchool;


    //putting getter and setters
```

Want to have one table for two classes…..

What to do?

One table from two classes: Solutions

```java
@Entity
public class School {
    @Id
    @GeneratedValue
    private int schoolId;
    private String schoolName;

    @Embedded
    private SchoolDetails schoolDetails;


    //putting getter and setters

```

```java
@Embeddable
public class SchoolDetails {

    private String schoolAddress;
    private boolean isPublicSchool;
```

1. put ref of SchoolDetails in School class (embedded class)
2. declare SchoolDetails as embedded class
3. Declare SchoolDetails as @Embeddable (dont declare it @Entity)

Status | Result1

	SCHOOLID	ISP...	SCHOOLADDRESS	STUDENTCOUNT	SCHOOLNAME
1	1	0	delhi public school, Delhi, India	3000	delhi public school

Workshop topics

- Hibernate-What it is ?
- JPA- What it is?
- ORM and Issues
- Hibernate Hello World CRUD
- Primary key generation strategy
- More annotations
- Hibernate Object life cycle
- Hibernate Architecture
- Component Mapping
- Relation mapping
  - Many-to-one mapping
  - one-to-one mapping
  - Many-to-Many mapping
- Inheritance in Hibernate
  - Single Table Strategy
  - Table Per Class Strategy
  - Joined Strategy
- HQL and the Query Object
  - one class to two tables
  - Select and Pagination in HQL
  - Named Quaries
  - Criteria API
- Hibernate caching
  - First Level
  - Second Level Cache
  - Hibernate optimization
  - Hibernate Batch Processing

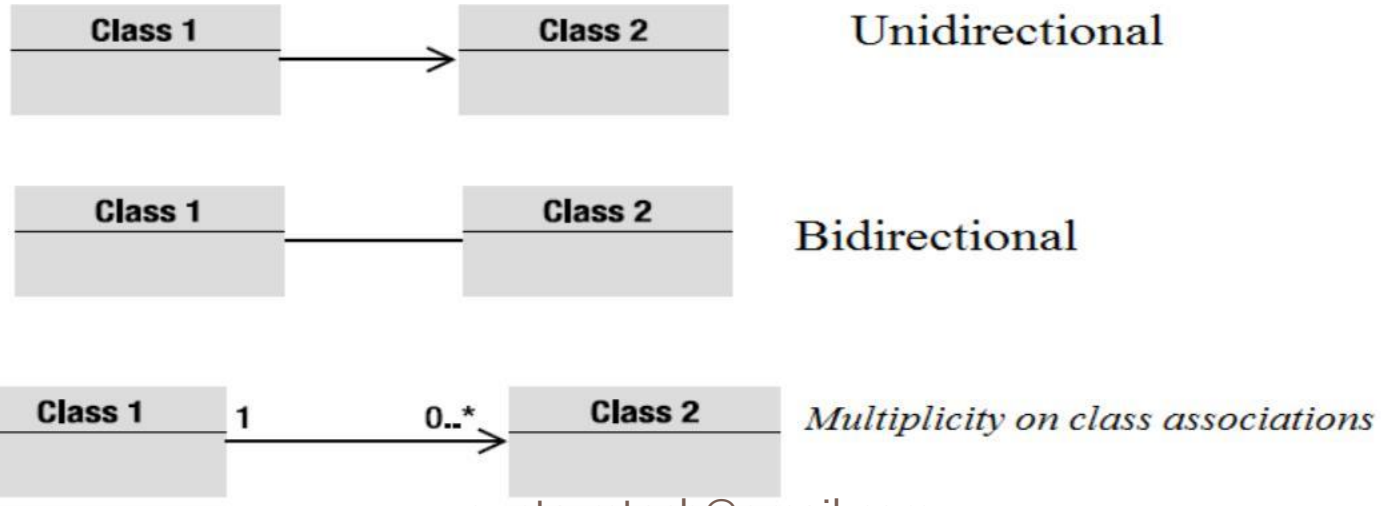
Relationship Mapping

- Most of time we required to map relationship between entities

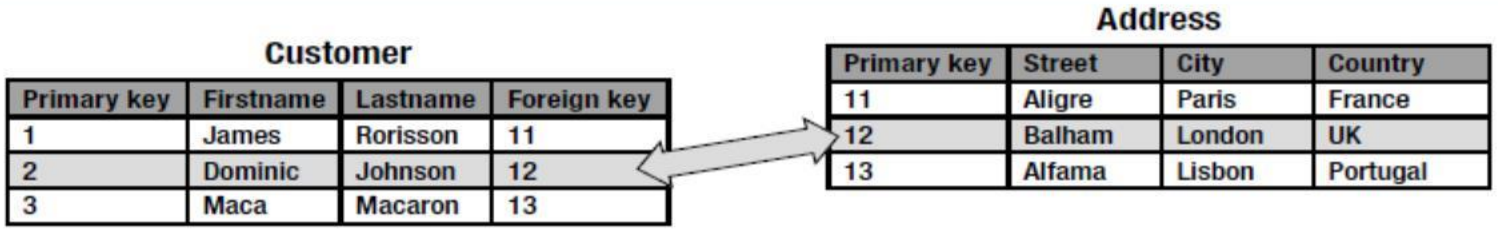
Every relationship has four characteristics:

- Directionality: Unidirectional vs Bidirectional ?
- Role: Each entity in the relationship is said to play a role. Depending on directionality, we can identity the entity playing the role of sources and entity playing the role of target
- Cardinality: The number of entity instances that exists on each side of the relationship
- Ownership: One of the two entities in the relationship is said to own the relationship. Employee is called owner of relationship and Department is called reverse-owner of relationship

Relationship Mapping

- OO relations
  - Association between the Objects
    - IS-A, HAS-A, USE-A
- An association has a direction:
  - Unidirectional
  - 

Relationships in Relational Databases



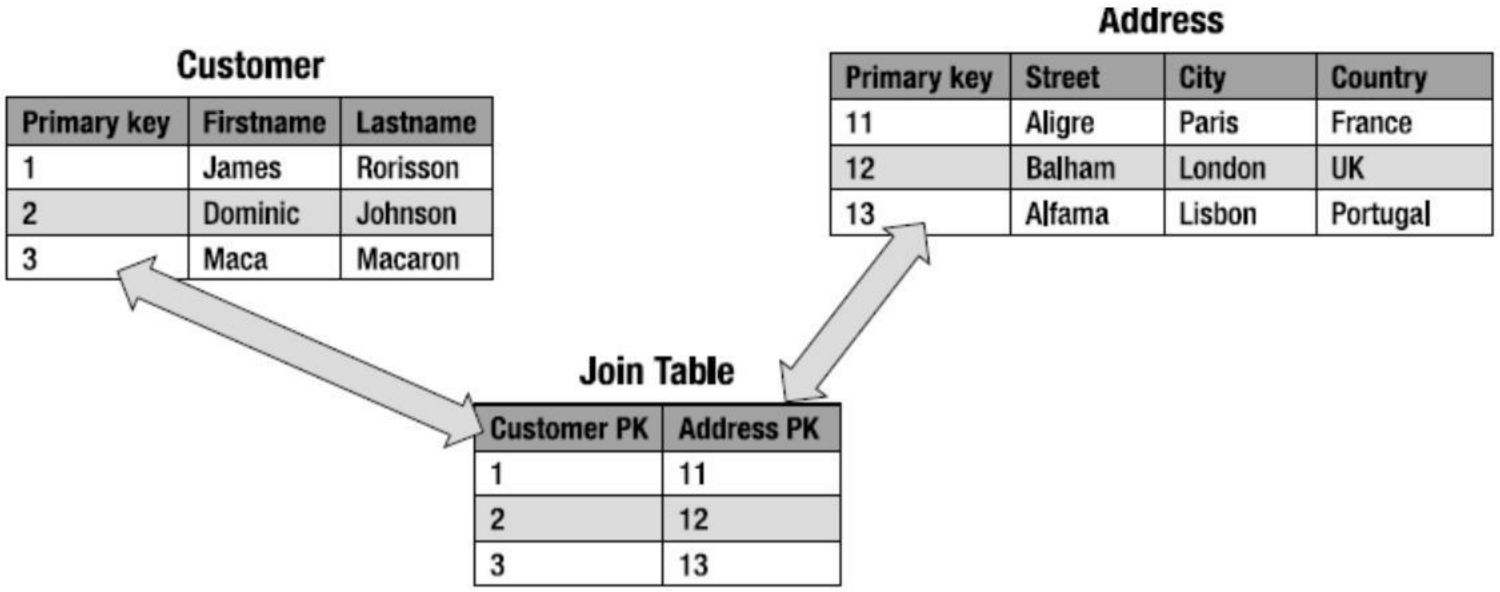
Customer

Primary key	Firstname	Lastname	Foreign key
1	James	Rorisson	11
2	Dominic	Johnson	12
3	Maca	Macaron	13

Address

Primary key	Street	City	Country
11	Aligre	Paris	France
12	Balham	London	UK
13	Alfama	Lisbon	Portugal

Figure 3-9. A relationship using a join column



Customer

Primary key	Firstname	Lastname
1	James	Rorisson
2	Dominic	Johnson
3	Maca	Macaron

Address

Primary key	Street	City	Country
11	Aligre	Paris	France
12	Balham	London	UK
13	Alfama	Lisbon	Portugal

Join Table

Customer PK	Address PK
1	11
2	12
3	13

Figure 3-10. A relationship using a join table

Entity Relationships

Table 3-1. All Possible Cardinality-Direction Combinations

Cardinality	Direction
One-to-one	Unidirectional
One-to-one	Bidirectional
One-to-many	Unidirectional
Many-to-one/one-to-many	Bidirectional
Many-to-one	Unidirectional
Many-to-many	Unidirectional
Many-to-many	Bidirectional

rgupta.mtech@gmail.com

one-to-one mapping

- In a one-to-one mapping the owner can be either the source entity or the target entity.
- one-to-one mapping is defined by annotating the owner entity with the @OneToOne annotations
- If the one-to-one mapping is bidirectional the inverse side of the relationship need to be specified too
- ☐In the non owner entity, the @OneToOne annotations must come with the mappedBy element

@OneToOne annotation in Employee.java

```
@Entity
public class Employee {
    @Id private int id;
    @OneToOne
    private ParkingSpace parkingSpace;
    ...
}
```

@OneToOne annotation (inverse side) in ParkingSpace.java

```
@Entity
public class ParkingSpace {
    @Id private int id;
    @OneToOne(mappedBy="parkingSpace")
    private Employee employee;
    ...
}
```

One to one Bidirectional

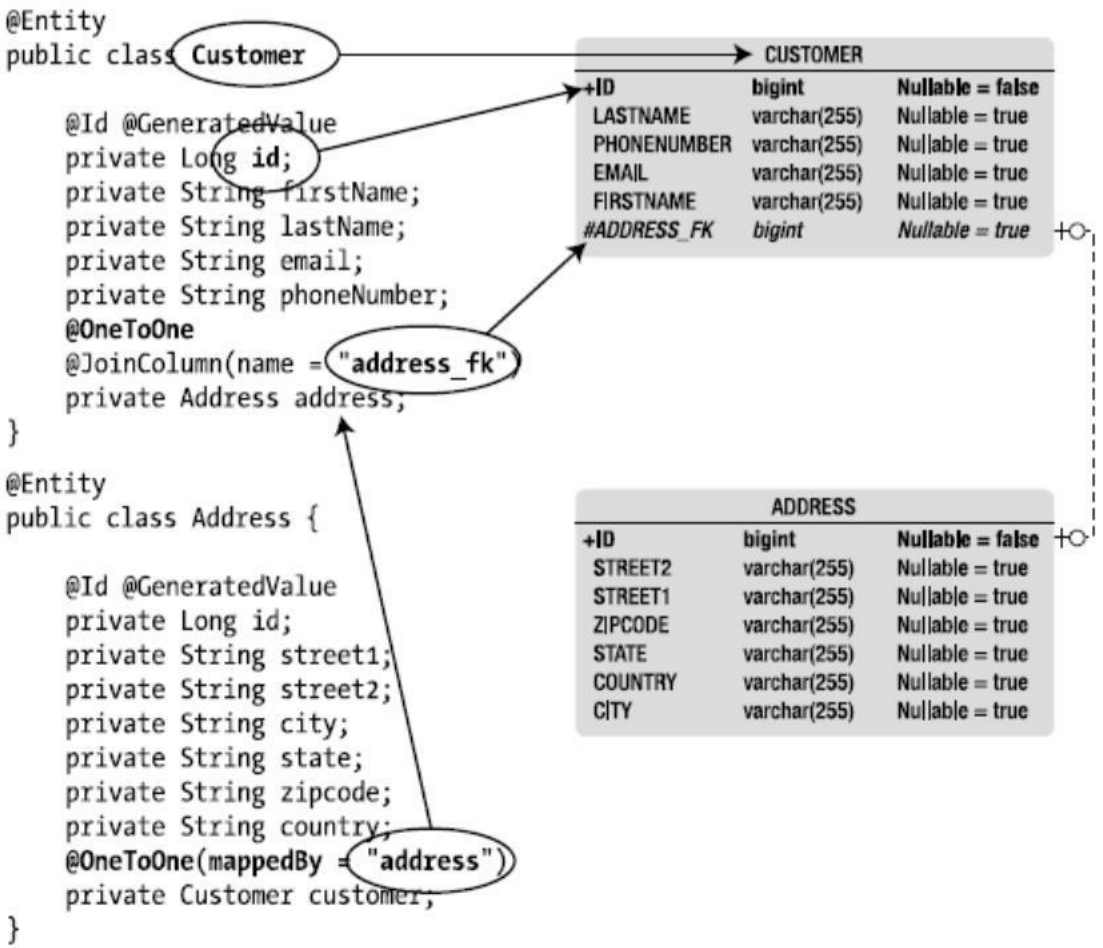


Figure 3-14. Customer and Address code with database mapping

rgupta.mtech@gmail.com

One to many mapping

- In an Many-to-one mapping the owner of the relationship is the source entity.
- ☐A Many-to-one mapping is defined by annotating the source entity with @ManyToOne annotation

@ManyToOne annotation in Employee.java

```
@Entity
public class Employee {
    @Id private int id;
    @ManyToOne
    private Department department;
    ...
}
```

@OneToMany annotation in Department.java

```
@Entity
public class Department {
    @Id private int id;
    @OneToMany(mappedBy="department")
    private Collection<Employee> employees;
    ...
}
```

The attribute on the target entity that owns the relationship

Many-to-Many mapping

- In a Many-to-Many mapping there is no join column . The only way to implement such a mapping is by means of a join table
- ☐Therefore, we can arbitrarily specify as owner either the source entity or the target entity
- If the many-to-many mapping is bi-directional, the inverse side of the relationship need to be specified too.
- ☐In the non owner entity the @ManyToMany annotation must come with mappedBy elements

@ManyToMany annotation in Employee.java

```
@Entity
public class Employee {
    @Id private int id;
    @ManyToMany
    private Collection<Project> projects;
        ...
}
```

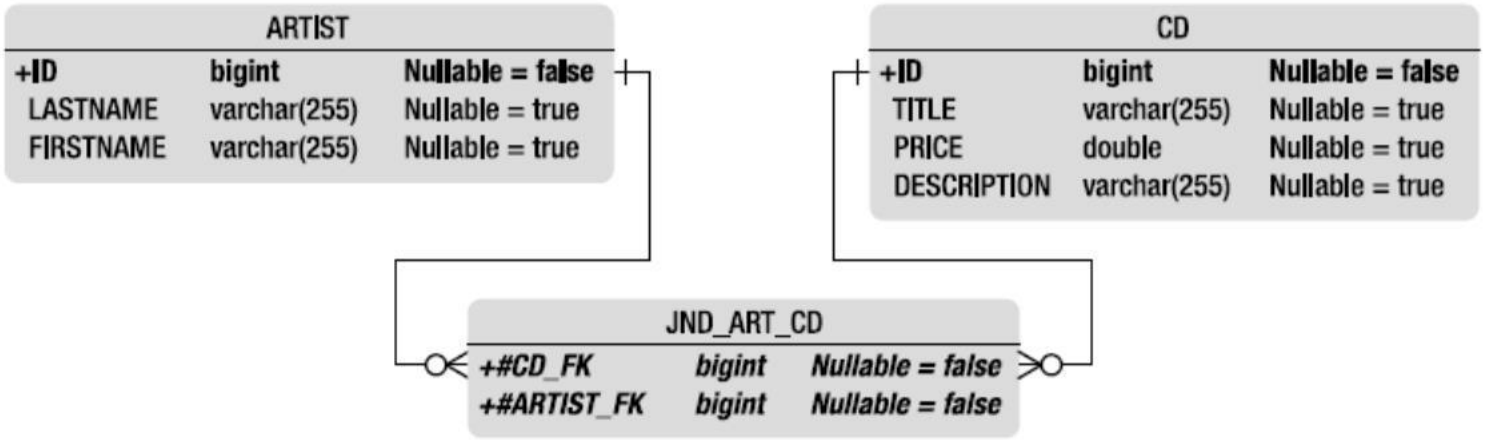
@ManyToMany annotation (inverse side) in Project.java

```
@Entity
public class Project {
    @Id private int id;
    @ManyToMany(mappedBy="projects")
    private Collection<Employee> employees;
    ...
}
```

Many to Many Bi-directional

```java
@Entity
public class CD {

    @Id @GeneratedValue
    private Long id;
    private String title;
    private Float price;
    private String description;
    @ManyToMany(mappedBy = "appearsOnCDs")
    private List<Artist> createdByArtists;

    // Constructors, getters, setters
}
```

```java
@Entity
public class Artist {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    @ManyToMany
    @JoinTable(name = "jnd_art_cd", ↪
        joinColumns = @JoinColumn(name = "artist_fk"), ↪
        inverseJoinColumns = @JoinColumn(name = "cd_fk"))
    private List<CD> appearsOnCDs;

    // Constructors, getters, setters
}
```



@OrderBy

- Dynamic ordering can be done with the @OrderBy annotation. "Dynamically" means that the ordering of the elements of a collection is made when the association is retrieved

```
@Entity
public class Comment {

    @Id @GeneratedValue
    private Long id;
    private String nickname;
    private String content;
    private Integer note;
    @Column(name = "posted_date")
    @Temporal(TemporalType.TIMESTAMP)
    private Date postedDate;

    // Constructors, getters, setters
}
```

```
@Entity
public class News {

    @Id @GeneratedValue
    private Long id;
    @Column(nullable = false)
    private String content;
    @OneToMany(fetch = FetchType.EAGER)
    @OrderBy("postedDate DESC")
    private List<Comment> comments;

    // Constructors, getters, setters
}
```

Lazy Loading

- The performance can be optimized by deferring the fetching of such data until they are actually needed. ☐This design pattern is called lazy loading
- At relationship level, lazy loading can be of great help in enhancing performance because it can reduce the amount of SQL get executed.
- the fetch mode can be specified on any of the four relationship mapping types
- ☐The parkingSpace attributers may not be loaded each time employee is

Lazy loading of the parkingSpace attribute

```
@Entity
public class Employee {
    @Id private int id;
    @OneToOne(fetch=FetchType.LAZY)
    private ParkingSpace parkingSpace;
    ...
}
```

When the fetch mode is not specified

- On a single valued relationship the related object is guaranteed to be loaded eagerly and Collection-valued relationship default to be lazily loaded
- ☐In case of bi-directional relationship the mode might be lazy on one side but eager on the other
- Quite common situation, relationship are often accessed in different way depending on the direction from which navigation occurs.
  - ☐The directive to lazily fetch an attribute is meant only to be hint to the persistence provider
  - The provider is not required to respect the request as the behaviour of the entity will not be compromised if the provider decides to eagerly load data.
  - ☐The converse is not true because specifying that an attributes be eagerly fetched might be critical to access the entity once detached.

Cascading operations

- Hibernate/JPA provides a mechanism to define when operations such as save()/persist() should be automatically cascaded across relationships
- You need to be sure that Address instance bas been set on Employee instance before invoking persist() on it.
- ☐The cascade attribute accepts several possible values coming from the cascade Type enumerations
  - **PERSIST, REFRESH, REMOVE, MERGE and DETACH**
- The constant ALL is a shorthand for declaring that all five operations should be cascaded.
- ☐As for relationship, cascade settings are unidirectional
- They must be explicitly set on both sides of a relationship if the same behaviours is inte

Enabling cascade persist

```
@Entity
public class Employee {
    @ManyToOne(cascade=CascadeType.PERSIST)
    Address address;
}
```

Collection of Basic Types

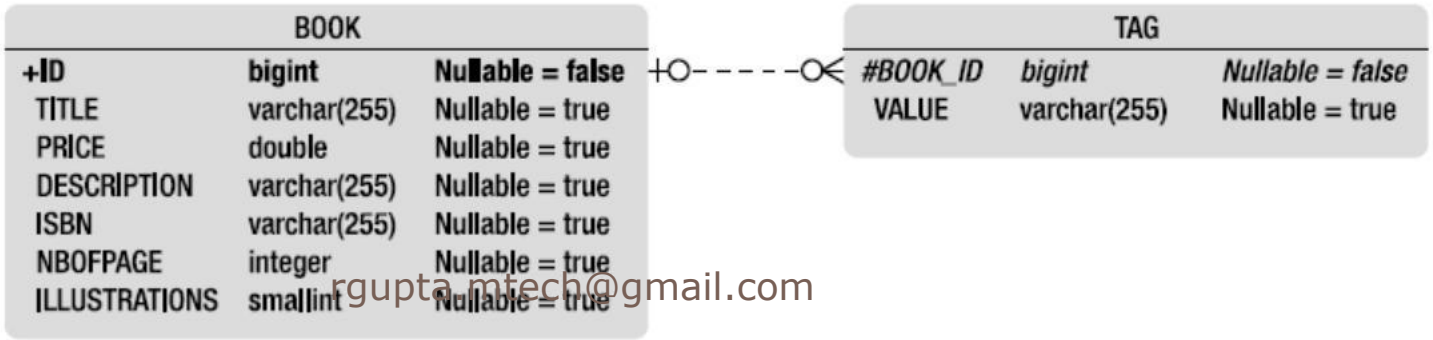
- @ElementCollection annotation is used to indicate that an attribute of type java.util.Collection contains a collection of instances of basic types (i.e., nonentities)

- Attribute can be of the following types:
  - **• java.util.Collection: Generic root interface in the collection hierarchy.**
  - **• java.util.Set: Collection that prevents the insertion of duplicate elements.**
  - **• java.util.List: Collection used when the elements need to be retrieved in some**
  - user-defined order.

Collection of Basic Types

Book Entity with collection of Strings

```
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;
    @ElementCollection(fetch = FetchType.LAZY)
    @CollectionTable(name = "Tag")
    @Column(name = "Value")
    private List<String> tags = new ArrayList<String>();

    // Constructors, getters, setters
}
```



Embeddables

```java
@Embeddable
public class Address {

    private String street1;
    private String street2;
    private String city;
    private String state;
    private String zipcode;
    private String country;

    // Constructors, getters, setters
}
```

```java
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Embedded
    private Address address;

    // Constructors, getters, setters
}
```

Listing 3-35. Structure of the CUSTOMER Table with All the Address Attributes

```
create table CUSTOMER (
   ID BIGINT not null,
   LASTNAME VARCHAR(255),
   PHONENUMBER VARCHAR(255),
   EMAIL VARCHAR(255),
   FIRSTNAME VARCHAR(255),
   STREET2 VARCHAR(255),
   STREET1 VARCHAR(255),
   ZIPCODE VARCHAR(255),
   STATE VARCHAR(255),
   COUNTRY VARCHAR(255),
   CITY VARCHAR(255),
   primary key (ID)
);
```

rgupta.mtech@gmail.com

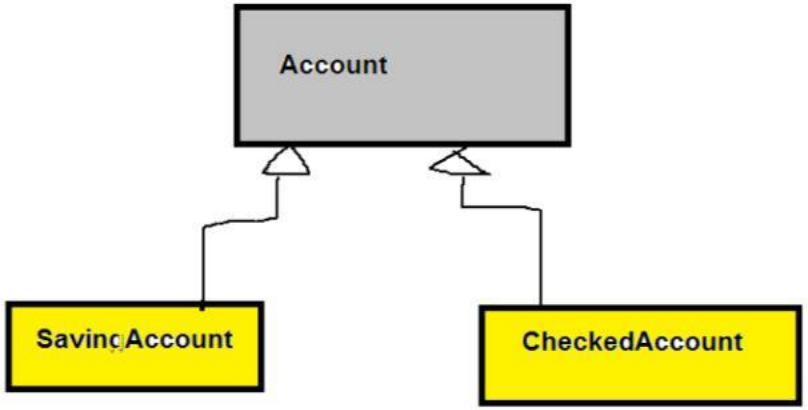
Workshop topics

- **Hibernate-What it is ?**
- **JPA- What it is?**
- **ORM and Issues**
- **Hibernate Hello World CRUD**
- **Primary key generation strategy**
- **More annotations**
- **Hibernate Object life cycle**
- **Hibernate Architecture**
- **Component Mapping**
- **Relation mapping**
  - **Many-to-one mapping**
  - **one-to-one mapping**
  - **Many-to-Many mapping**
- **Inheritance in Hibernate**
  - **Single Table Strategy**
  - **Table Per Class Strategy**
  - **Joined Strategy**
- **HQL and the Query Object**
  - one class to two tables
  - Select and Pagination in HQL
  - Named Quaries
  - Criteria API
- **Hibernate caching**
  - First Level
  - Second Level Cache
- **Hibernate optimization**
- **Hibernate Batch Processing**

Mapping Inheritance

- Inheritance can be used also for persistent objects, for factoring out data members inherited by multiple subclasses

- The mapping of a hierarchy to the database can follow different strategies: -
  - –Single table per hierarchy
  - –Table per class(table per concrete class
  - –Joined



Inheritance

- Single table (single table per hierarchy)
  - Here only one table is going to be created, all fields mapped to single table.
  - Not very memory efficient, Faster, support polymorphic quaries
- Joined (as per normalization)
  - Separate table mapped to all classes in the hierarchy
  - Three table is created Account, SavingAccount, CurrentAccount, as per normalization, table are need to join to get all data (sql outer join)
- Table per class(Table per concrete class)
  - Two table is going to create SavingAccount and CurrentAccount
    Sql Union operation is required to get data, Identity auto gen key is not supported
  - A U B, Not Supported by all vendors

Single table (single table per hierarchy)

```java
@Entity
@Table(name = "Account")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="accountType",discriminatorType=DiscriminatorType.STRING)
public class Account{
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private int accountId;
    private String accountHolderName;
    private double balance;
```

```java
@Entity
@Table(name="Account")
@DiscriminatorValue("S")
public class SavingAccount extends Account{
    private double intrestRate;
```

```java
@Entity
@Table(name="Account")
@DiscriminatorValue("C")
public class CurrentAccount extends Account{
    private double overdraft;
```

```
mysql> desc account;
+-------------------+--------------+------+-----+---------+
| Field             | Type         | Null | Key | Default |
+-------------------+--------------+------+-----+---------+
| accountType       | varchar(31)  | NO   |     | NULL    |
| accountId         | int(11)      | NO   | PRI | NULL    |
| accountHolderName | varchar(255) | YES  |     | NULL    |
| balance           | double       | NO   |     | NULL    |
| overdraft         | double       | YES  |     | NULL    |
| intrestRate       | double       | YES  |     | NULL    |
+-------------------+--------------+------+-----+---------+
```

Table per class

```java
@Entity
@Table(name = "Account")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)

public class Account{
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private int accountId;
    private String accountHolderName;
    private double balance;
```

```java
@Entity
@Table(name="SavingAccount")
public class SavingAccount extends Account{
    private double intrestRate;
```

```java
@Entity
@Table(name="CurrentAccount")
public class CurrentAccount extends Account{
    private double overdraft;
```

Workshop topics

- Hibernate-What it is ?
- JPA- What it is?
- **ORM and Issues**
- Hibernate Hello World CRUD
- Primary key generation strategy
- More annotations
- Hibernate Object life cycle
- Hibernate Architecture
- Component Mapping
- Relation mapping
  - Many-to-one mapping
  - one-to-one mapping
  - Many-to-Many mapping
- Inheritance in Hibernate
  - Single Table Strategy
  - Table Per Class Strategy
  - Joined Strategy
- **HQL and the Query Object**
  - Select and Pagination in HQL
  - Named Quarries
  - Criteria API
- **Hibernate caching**
  - First Level
  - Second Level Cache
- **Hibernate optimization**
- **Hibernate Batch Processing**

HQL

- HQL is OO version of SQL
- HQL uses class name instead of table name, and property names instead of column name
- HQL fully supports polymorphic queries
- **Way to pulling data from the database in the Hibernate.**
  1. **Using session methods(get() and load() methods)**
     - **limited control to accessing data**
  2. **Using HQL & Native HQL**
     - **Slightly more control using where clause**
  3. **Using Criteria API**
     - **The criteria API is an alternative of HQL queries.**
     - **It is more powerful and flexible for writing tricky criteria functions and dynamic queries**

HQL Syntax

- **HQL Queries elements:**
  - Clauses, Aggregate functions, Subqueries
- **Clauses in the HQL are:**
  - from, select, where, order by, group by
- **Aggregate functions are:**
  - avg(...), sum(...), min(...), max(...) , count(*)
  - count(...), count(distinct ...), count(all...)

Hello World HQL

Select All

```
//List<Customer>clist=session.createQuery("from Customer").list();
List<Customer>clist=session.createQuery("select c from Customer c").list();
for(Customer c:clist)
    System.out.println(c);
```

Select on condition

```
Query query=session.createQuery("select c from Customer c where name=:name1");
query.setParameter("name1", "rajeev");
List<Customer>clist=query.list();
for(Customer c:clist)
    System.out.println(c);
```

```
Query query=session.createQuery("select c from Customer c where name=:name1 and address=:address1");
query.setParameter("name1", "rajeev");
query.setParameter("address1", "noida");
```

```
Query query=session.createQuery("delete Customer where name=:name ");
query.setParameter("name", "rajeev");
```

HQL Insert Query Example

- In HQL, only the INSERT INTO … SELECT … is supported; there is no INSERT INTO … VALUES.
- HQL only support insert from another table. For example

```
"insert into Object (id, name) select oo.id, oo.name from OtherObject oo";
```

Insert a stock record from another backup_stock table. This can also called bulk-insert statement.

```
Query query = session.createQuery("insert into Stock(stock_code, stock_name)" +
                        "select stock_code, stock_name from backup_stock");
int result = query.executeUpdate();
```

The **query.executeUpdate()** will return how many number of record has been inserted, updated or deleted.

Native SQL queries

- In Hibernate, HQL or criteria queries should be able to let you to execute almost any SQL query you want.
- However, many developers are complaint about the Hibernates generated SQL statement is slow and more prefer to generated their own SQL (native SQL) statement
- Hibernate provide a **createSQLQuery** method to let you call your native SQL statement directly.
  - In this example, you tell Hibernate to return you a Stock.class, all the select data (*) will match to your Stock.class properties automatically.

```
Query query = session.createSQLQuery(
"select * from stock s where s.stock_code = :stockCode")
.addEntity(Stock.class)
.setParameter("stockCode", "7277");
List result = query.list();
```

```
Query query = session.createSQLQuery(
"select s.stock_code from stock s where s.stock_code = :stockCode")
.setParameter("stockCode", "7277");
List result = query.list();
```

Named Quarries

- Give unique name to the Queries that work for entire application.
- The application can use the query by using the name of the

```
@Entity
@NamedQueries({ @NamedQuery(name = "findCustomer.byId",
        query = "from Customer c where c.id = :id") })
@NamedNativeQueries({ @NamedNativeQuery(name = "findCustomer.byAddress",
        query = "select * from Customer c where c.address = :address", resultClass = Customer.class) })
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
```

```
Query query = session.getNamedQuery("findCustomer.byId").setInteger("id", 3);

List<Customer> cList = query.list();
for(Customer temp : cList)
{
 System.out.println(temp);
}
```

Criteria API in Hibernate

- The criteria API is an alternative of HQL queries.
- It is more powerful and flexible for writing tricky criteria functions and dynamic queries

```
Criteria criteria=session.createCriteria(Customer.class);
criteria.add(Restrictions.eq("name", "rajeev"));
criteria.add(Restrictions.eq("address", "delhi"));
criteria.add(Restrictions.le("id", 5));
List<Customer>cList=criteria.list();

for(Customer customer:cList)
    System.out.println(customer);
```

There is a difference in terms of performance between HQL and criteriaQuery, everytime you fire a query using criteriaQuery, it creates a new alias for the table name which does not reflect in the last queried cache for any DB. This leads to an overhead of compiling the generated SQL, taking more time to execute.

Regarding fetching strategies [http://www.hibernate.org/315.html]

- http://stackoverflow.com/questions/197474/hibernate-criteria-vs-hql

Criteria API in Hibernate

```
Criteria criteria=session.createCriteria(Customer.class);

//============================================================
//Restiction Restrictions.in
//============================================================
criteria.add(Restrictions.in("id", new Integer[]{3,5,7}));

//============================================================
//Restiction Restrictions.like
//============================================================

 Criteria criteria = session.createCriteria(Customer.class)
 criteria.add(Restrictions.like("name", "%raja%"));


//============================================================
//Restiction Restrictions.isNull, Restrictions.isNotNull
//============================================================

Criteria criteria = session.createCriteria(Customer.class);

criteria.add(Restrictions.isNull("address"));


//============================================================
//Restiction Restriction.between
//============================================================

Criteria criteria = session.createCriteria(Customer.class);

criteria.add(Restrictions.between("id", 3, 7));
```

Criteria API in Hibernate

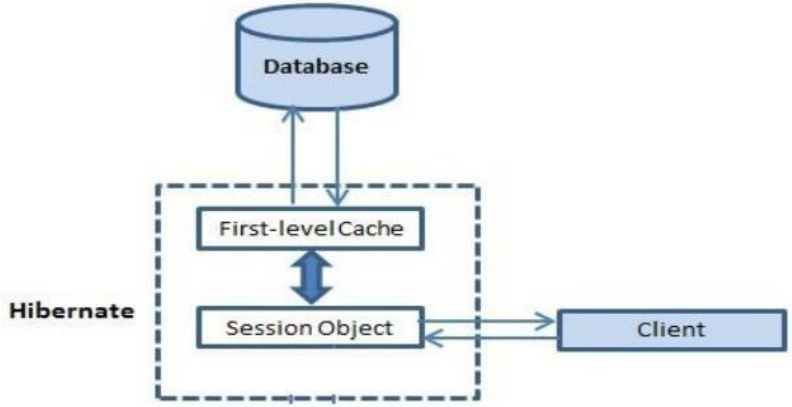
```
// Restriction.allEq
 //SELECT * FROM user WHERE userName = +userName AND userPassword = +userPassword;

 Map map = new HashMap();
 map.put("username", "username");
 map.put("userPassword", "userPassword");
 Criteria criteria = session.createCriteria(User.class);
 criteria.add(Restrictions.allEq(map));
 List list = (List) criteria.uniqueResult();
```

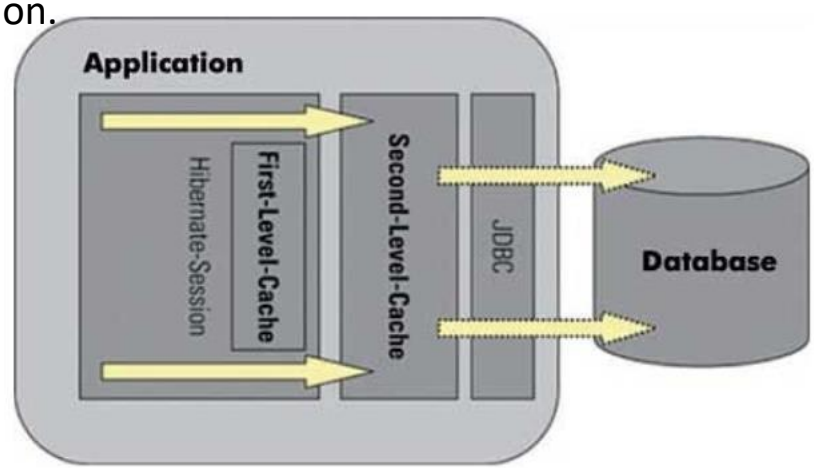
Workshop topics

- Hibernate-What it is ?
- JPA- What it is?
- ORM and Issues
- Hibernate Hello World CRUD
- Primary key generation strategy
- More annotations
- Hibernate Object life cycle
- Hibernate Architecture
- Component Mapping
- Relation mapping
  - Many-to-one mapping
  - one-to-one mapping
  - Many-to-Many mapping
- Inheritance in Hibernate
  - Single Table Strategy
  - Table Per Class Strategy
  - Joined Strategy
- HQL and the Query Object
  - Select and Pagination in HQL
  - Named Quarries
  - Criteria API
- Hibernate caching
  - First Level
  - Second Level Cache
- Hibernate optimization
- Hibernate Batch Processing

Hibernate caching

- Caching is a facility provided by ORM frameworks which help users to get fast running web application, while help framework itself to reduce number of queries made to database in a single transaction.
- **Fist level cache** in hibernate is enabled by default and you do not need to do anything to get this functionality working. In fact, you can not disable it even forcefully.
- First level cache is provided by Session Object. First level cache associated with session object is available only till session object is live. It is available to session object only and is **not accessible to any other session** tion.





Important facts about primary caching

- First level cache is associated with “session” object and other session objects in application can not see it.
- The scope of cache objects is of session. Once session is closed, cached objects are gone forever.
- First level cache is enabled by default and you can not 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 be 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.

Primary **caching**

- Primary caching is by default and you can not disable it.
- What if you try to load same object in one session , hibernate don't hit database twice, in fact it will hit database first time and put result in cache so that if next time it is request it can be provided from cache.

```
Session session=factory.openSession();

session.beginTransaction();

Customer customer1=(Customer) session.load(Customer.class, 1);
System.out.println(customer1.getName());

Customer customer2=(Customer) session.load(Customer.class, 1);
System.out.println(customer1.getName());
```

```
Hibernate:
    select
        customer0_.id as id0_0_,
        customer0_.address as address0_0_,
        customer0_.name as name0_0_
    from
        Customer customer0_
    where
        customer0_.id=?
rajeev
rajeev
```

- As you can see that **second "session.load()" statement does not execute select query again and load the Customer entity directly**.

First level cache case II

- **First level cache retrieval example with new session**
- With new session, entity is fetched from database again irrespective of it is already present in any other session in

```
//Loading Same Customer record in
// two different sessions

Session session1=factory.openSession();
Session session2=factory.openSession();

session1.beginTransaction();
session2.beginTransaction();

Customer customer1=(Customer) session1.load(Customer.class, 1);
System.out.println(customer1.getName());

Customer customer2=(Customer) session2.load(Customer.class, 1);
System.out.println(customer2.getName());

session1.getTransaction().commit();
session2.getTransaction().commit();
```

```
Hibernate:
    select
        customer0_.id as id0_0_,
        customer0_.address as address0_0_,
        customer0_.name as name0_0_
    from
        Customer customer0_
    where
        customer0_.id=?
rajeev
Hibernate:
    select
        customer0_.id as id0_0_,
        customer0_.address as address0_0_,
        customer0_.name as name0_0_
    from
        Customer customer0_
    where
        customer0_.id=?
rajeev
```

Removing cache objects

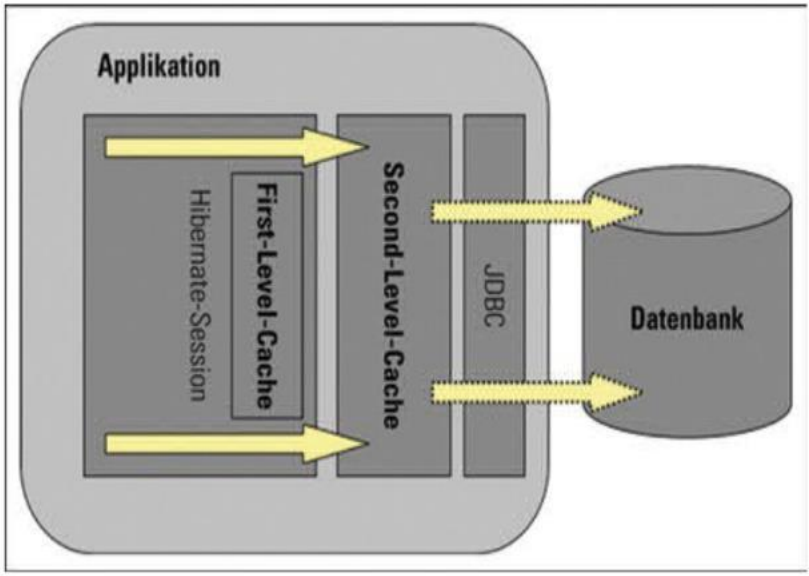
- **How to remove from first level cache?**
- Though we can not disable the first level cache in hibernate, but we can certainly remove some of objects from it when needed. This is done using two methods :

```
1  //Loading Same Customer record in
   // same  sessions
2
   Session session1=factory.openSession();

   session1.beginTransaction();


   Customer customer1=(Customer) session1.load(Customer.class, 1);
   System.out.println(customer1.getName());

   session1.evict(customer1);

   Customer customer2=(Customer) session1.load(Customer.class, 1);
   System.out.println(customer2.getName());
```

Clearly, evict() method removed the department object from cache so that it was fetched again from database.

Hibernate second level

- **second level cache is created in session factory scope** and is **available to be used in all sessions** which are created using that particular session factory.
- It means that **once session factory is closed, all cache associated with it die** and cache manager also closed down.
- Further, It also means that if you have two instances of session factory (normally no application does that), you will have two cache managers in your application and while accessing cache stored in physical store, you might get unpredictable results like cache-miss.



How second level cache works

1. Whenever hibernate session try to load an entity, the very first place it look for cached copy of entity in first level cache (associated with particular hibernate session).
2. If cached copy of entity is present in first level cache, it is returned as result of load method.
3. If there is no cached entity in first level cache, then second level cache is looked up for cached entity.
4. If second level cache has cached entity, it is returned as result of load method. But, before returning the entity, it is stored in first level cache also so that next invocation to load method for entity will return the entity from first level cache itself, and there will not be need to go to second level cache again.
5. If entity is not found in first level cache and second level cache also, then database query is executed and entity is stored in both cache levels, before returning as response of load() method.
6. Second level cache validate itself for modified entities, if modification has been done through hibernate session APIs.
7. If some user or process make changes directly in database, the there is no way that second level cache update itself until “timeToLiveSeconds” duration has passed for that cache region. In this case, it is good idea to invalidate whole cache and let hibernate build its cache once again. You can use below code snippet to invalidate whole hibernate second level cache.

Workshop topics

- Hibernate-What it is ?
- JPA- What it is?
- **ORM and Issues**
- Hibernate Hello World CRUD
- Primary key generation strategy
- More annotations
- Hibernate Object life cycle
- Hibernate Architecture
- Component Mapping
- Relation mapping
  - Many-to-one mapping
  - one-to-one mapping
  - Many-to-Many mapping
- Inheritance in Hibernate
  - Single Table Strategy
  - Table Per Class Strategy
  - Joined Strategy
- **HQL and the Query Object**
  - Select and Pagination in HQL
  - Named Quarries
  - Criteria API
- **Hibernate caching**
  - First Level
  - Second Level Cache
- **Hibernate optimization**
- **Hibernate Batch Processing**

Workshop topics

- Hibernate-What it is ?
- JPA- What it is?
- **ORM and Issues**
- Hibernate Hello World CRUD
- Primary key generation strategy
- More annotations
- Hibernate Object life cycle
- Hibernate Architecture
- Component Mapping
- Relation mapping
  - Many-to-one mapping
  - one-to-one mapping
  - Many-to-Many mapping
- Inheritance in Hibernate
  - Single Table Strategy
  - Table Per Class Strategy
  - Joined Strategy
- **HQL and the Query Object**
  - Select and Pagination in HQL
  - Named Quarries
  - Criteria API
- **Hibernate caching**
  - First Level
  - Second Level Cache
- **Hibernate optimization**
- **Hibernate Batch Processing**

JPA

JPA?

- What is JPA?
  - JSR 220, JSR-317 ,Java Persistence 2.0
  - Specification implemented by : Hibernate , eclipseLink, topLink etc
  - JPA abstraction above JDBC
  - **javax.persistence package**

- Component of JPA?
  - ORM
  - Entity manager API CRUD operations
  - JPQL
  - Transactions and locking mechanisms when accessing data concurrently provided by:
    - Java Transaction API (JTA)
    - Resource-local (non-JTA)

  - Callbacks and listeners to hook business logic into the life cycle of a persistent

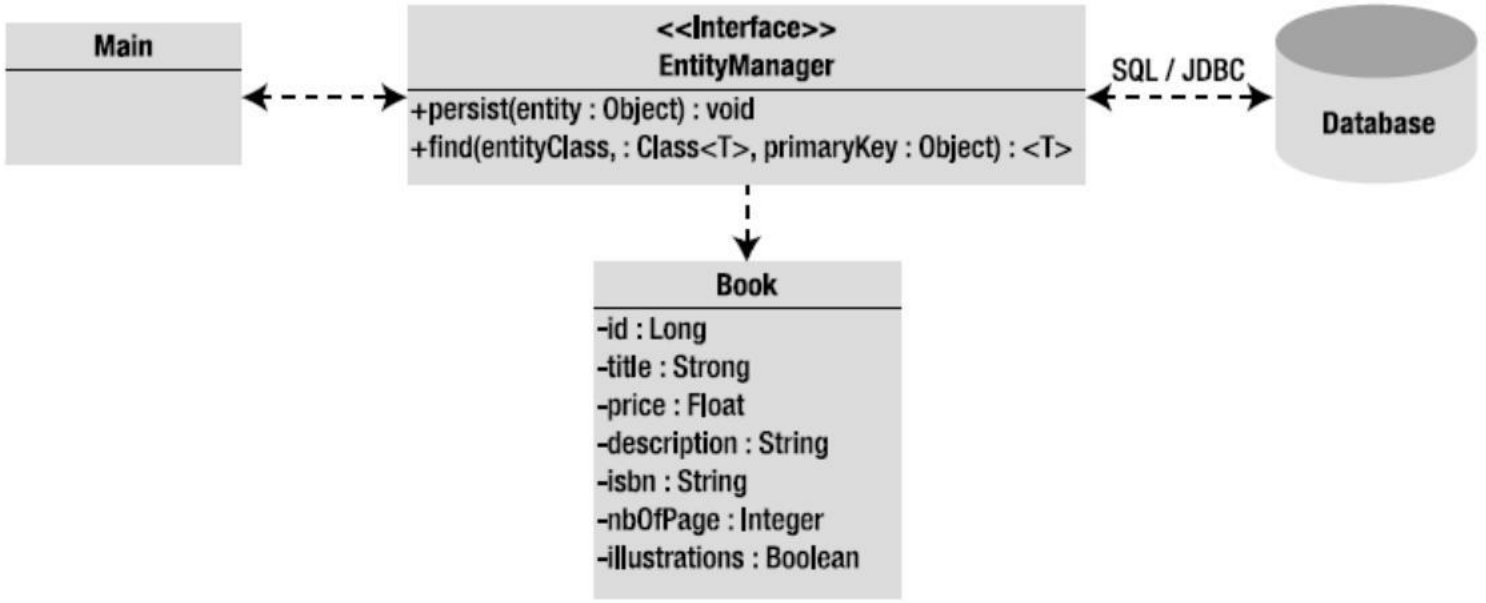
JPA vs Hibernate

JPA	Hibernate
Entity Classes	Persistent Classes
EntityManagerFactory	SessionFactory
EntityManager	Session
Persistence	Configuration
EntityTransaction	Transaction
Query	Query
Persistence Unit	Hibernate Config

rgupta.mtech@gmail.com

EntityManager interacts with Entity

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("chapter02PU");
EntityManager em = emf.createEntityManager();
em.persist(book);
```



EntityManager Methods

Table 4-1. EntityManager Interface Methods to Manipulate Entities

Method	Description
`void persist(Object entity)`	Makes an instance managed and persistent
`<T> T find(Class<T> entityClass, Object primaryKey)`	Searches for an entity of the specified class and primary key
`<T> T getReference(Class<T> entityClass, Object primaryKey)`	Gets an instance, whose state may be lazily fetched
`void remove(Object entity)`	Removes the entity instance from the persistence context and from the underlying database
`<T> T merge(T entity)`	Merges the state of the given entity into the current persistence context
`void refresh(Object entity)`	Refreshes the state of the instance from the database, overwriting changes made to the entity, if any
`void flush()`	Synchronizes the persistence context to the underlying database
`void clear()`	Clears the persistence context, causing all managed entities to become detached
`void detach(Object entity)`	Removes the given entity from the persistence context, causing a managed entity to become detached
`boolean contains(Object entity)`	Checks whether the instance is a managed entity instance belonging to the current persistence context

EntityManager Methods

Listing 4-9. *Persisting a Customer with an Address*

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE", "UK");
customer.setAddress(address);

tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
```

Listing 4-10. *Finding a Customer by ID*

```
Customer customer = em.find(Customer.class, 1234L)
if (customer!= null) {
    // Process the object
}
```

Listing 4-11. *Finding a Customer by Reference*

```
try {
    Customer customer = em.getReference(Customer.class, 1234L)
    // Process the object
} catch(EntityNotFoundException ex) {
    // Entity not found
```

em.getReference()

=>Takes the same parameters, but it retrieves a reference to an entity (via its primary key) and not its data.
=>It is intended for situations where a managed entity instance is needed, but no data, other than potentially the entity's primary key, being accessed.
=>With getReference(), the state data is fetched lazily,which means that, if you don't access state before the entity is detached, the data might not be there.
=> If the entity is not found, an EntityNotFoundException is thrown

EntityManager Methods remove()

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE", "UK");
customer.setAddress(address);

tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();

tx.begin();
em.remove(customer);
tx.commit();
```

Listing 4-13. The Customer Entity Dealing with Orphan Address Removal

```
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    @OneToOne (fetch = FetchType.LAZY, orphanRemoval=true)
    private Address address;

    // Constructors, getters, setters
}
```

EntityManager Methods persist(), flush(), refresh()

```
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
```

```
tx.begin();
em.persist(customer);
em.flush();
em.persist(address);
tx.commit();
```

Forcing persistence to flush the data, to synch with DB

```
Customer customer = em.find(Customer.class, 1234L)
assertEquals(customer.getFirstName(), "Antony");

customer.setFirstName("William");

em.refresh(customer);
assertEquals(customer.getFirstName(), "Antony");
```

The **refresh() method is used for data synchronization in the opposite direction of the flush, meaning it** overwrites the current state of a managed entity with data as it is present in the database. A typical case is where the **EntityManager.refresh() method is used to undo changes that have been done to the entity** in memory only. The test class snippet in Listing 4-14 finds a **Customer by ID, changes its first name, and** undoes this change using the **refresh() method.**

EntityManager Methods :Clear and Detach

- The clear() method is straightforward: it empties the persistence context, causing all managed entities to become detached.
- The detach(Object entity) method removes the given entity from the persistence context.

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");

tx.begin();
em.persist(customer);
tx.commit();

em.detach(customer);
```

hronized to the

Merging an Entity

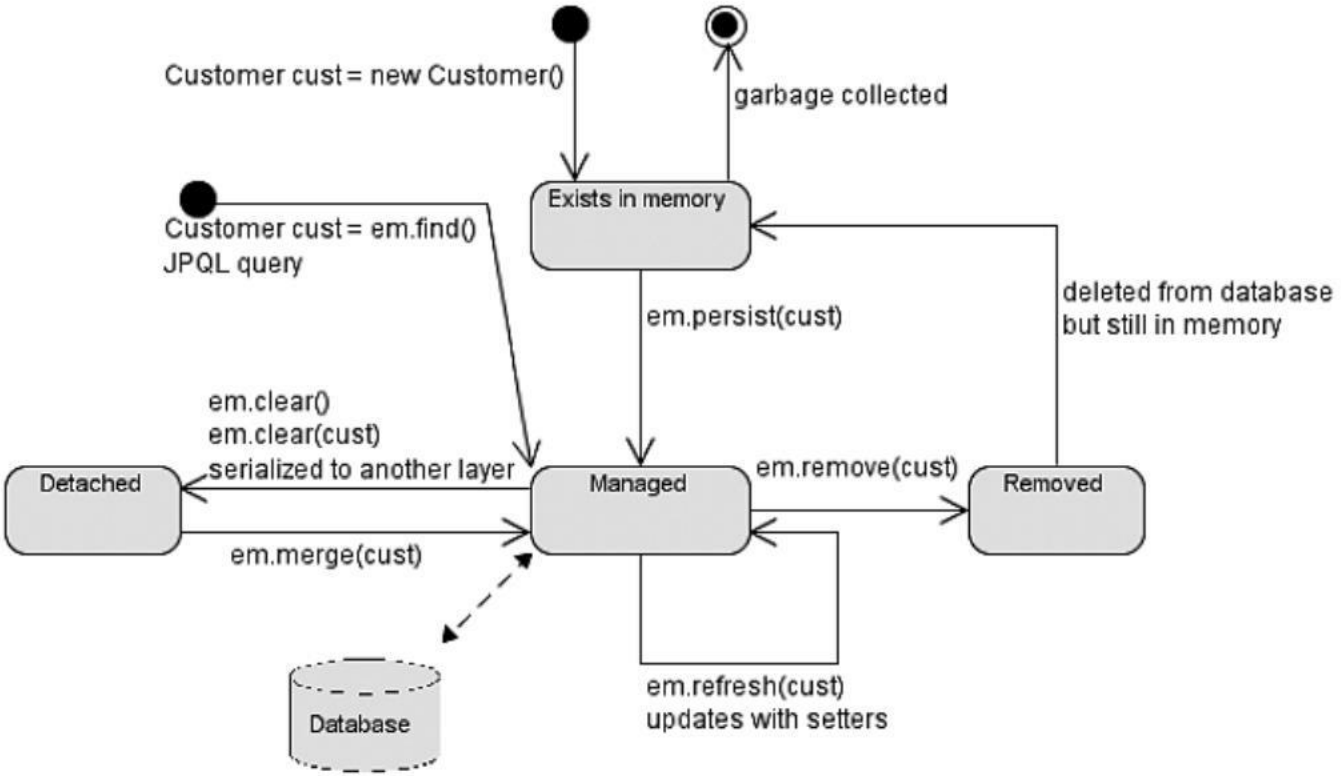
- A detached entity is no longer associated with a persistence context. If you want to manage it, you need to merge it.
- Let's take the example of an entity that needs to be displayed in a JSF page. The entity is first loaded from the database in the persistent layer (it is managed), it is returned from an invocation of a local EJB (it is detached because the transaction context ends), the presentation layer displays it (it is still detached), and then it returns to be updated to the database.
- However, at that moment, the entity is detached and needs to be attached again, or merged, to synchronize its state with the database.

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");

tx.begin();
em.persist(customer);
tx.commit();

em.clear();

// Sets a new value to a detached entity
customer.setFirstName("William");

tx.begin();
em.merge(customer);
tx.commit();
```

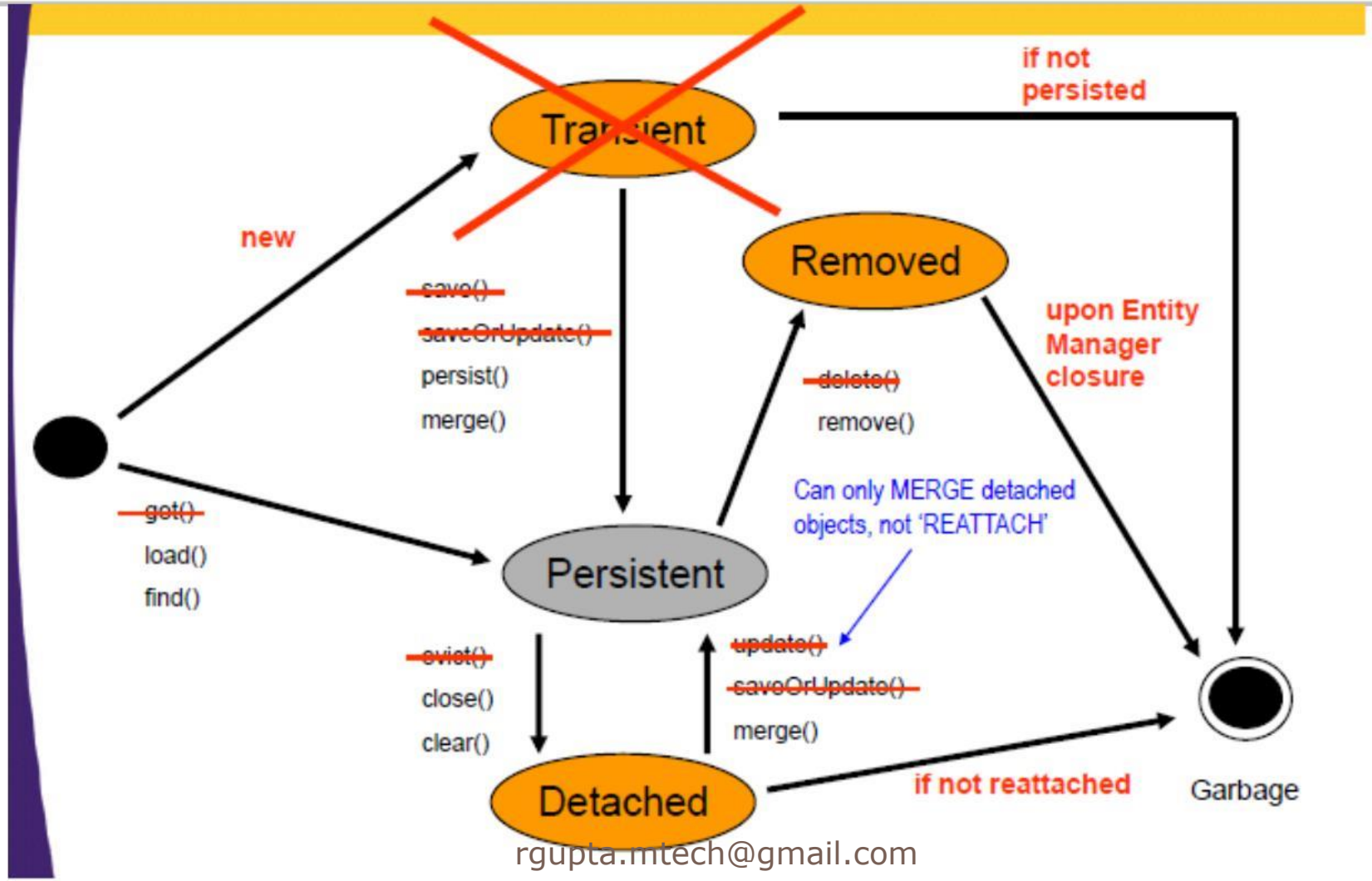
Updating an Entity

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");

tx.begin();
em.persist(customer);

customer.setFirstName("Williman");

tx.commit();
```

Entity Life Cycle



Entity Life Cycle



persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
             http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">

  <persistence-unit name="chapter02PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>com.apress.javaee6.chapter02.Book</class>
    <properties>
      <property name="eclipselink.target-database" value="DERBY"/>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
      <property name="eclipselink.logging.level" value="INFO"/>
      <property name="javax.persistence.jdbc.driver" ➥
              value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="javax.persistence.jdbc.url" ➥
         value="jdbc:derby://localhost:1527/chapter02DB;create=true"/>
      <property name="javax.persistence.jdbc.user" value="APP"/>
      <property name="javax.persistence.jdbc.password" value="APP"/>
    </properties>
  </persistence-unit>
</persistence>
```

Inserting Record

```
// Creates an instance of book
Book book = new Book();
book.setTitle("The Hitchhiker's Guide to the Galaxy");
book.setPrice(12.5F);
book.setDescription("Science fiction comedy book");
book.setIsbn("1-84023-742-2");
book.setNbOfPage(354);
book.setIllustrations(false);

// Gets an entity manager and a transaction
EntityManagerFactory emf = ➥
        Persistence.createEntityManagerFactory("chapter02PU");
EntityManager em = emf.createEntityManager();

// Persists the book to the database
EntityTransaction tx = em.getTransaction();
tx.begin();
em.persist(book);
tx.commit();

em.close();
emf.close();
}

// Retrieves all the books from the database
List<Book> books = ➥
        em.createNamedQuery("findAllBooks").getResultList();
```