# Introduction to Hibernate

SUJATA BATRA

# Object/Relational Mapping

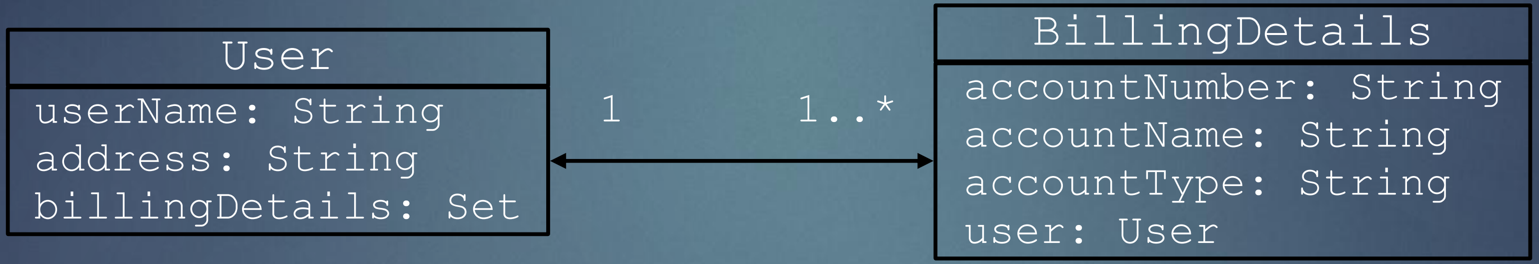
- A major part of any enterprise application development project is the persistence layer
    - Accessing and manipulate persistent data typically with relational database
- ORM handles Object-relational impedance mismatch
    - Data lives in the relational database, which is table driven (with rows and columns)
- Relational database is designed for fast query operation of table-driven data
    - Work with objects, not rows and columns of table

# Object-Relational Mapping

- Automated persistence of object to tables in RDBMS.
- Usually with the help of metadata that describes the mapping.
  - SQL is auto-generated by the metadata description.
- An ORM Solution consists of the following pieces:
  - Persistence Manager with CRUD API.
  - Query API
  - Mapping metadata

## Domain Model and the paradigm mismatch

- Classes implement the business entities of our domain model
  - attributes of the entity are properties of our Java class
  - associations between entities are also implemented with properties



User

userName: String
address: String
billingDetails: Set

1

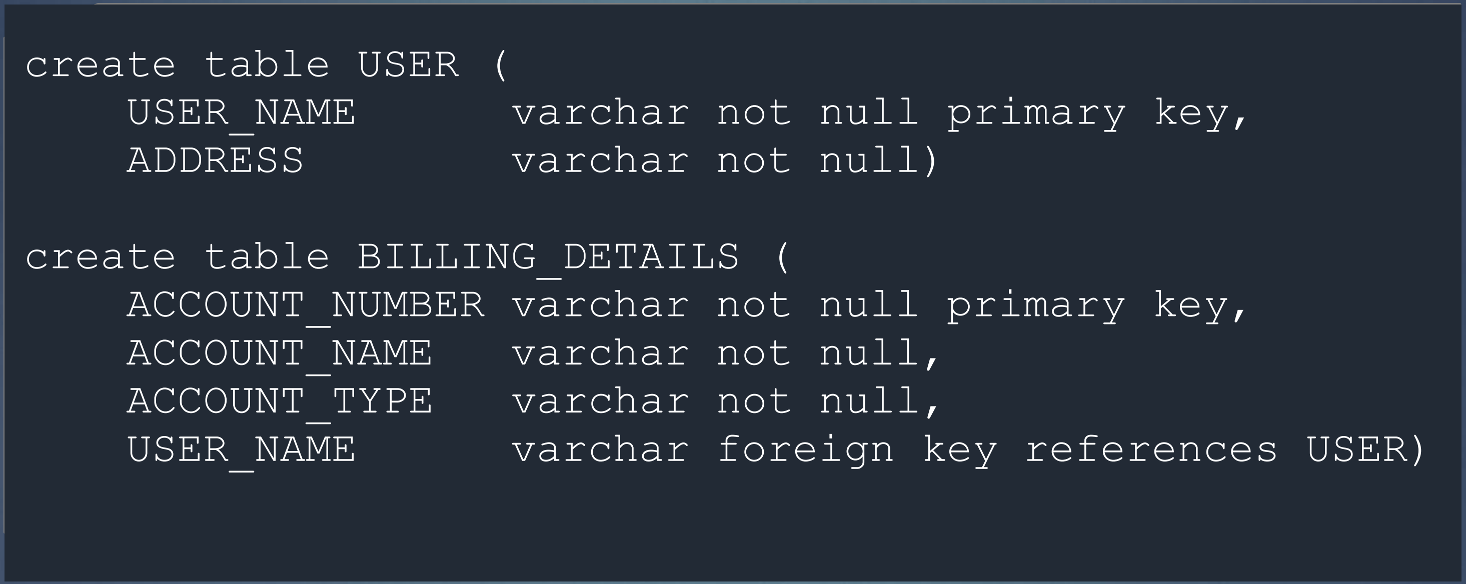
1..*

BillingDetails

accountNumber: String
accountName: String
accountType: String
user: User

- Let’s see if there is a problem mapping this to tables and columns...

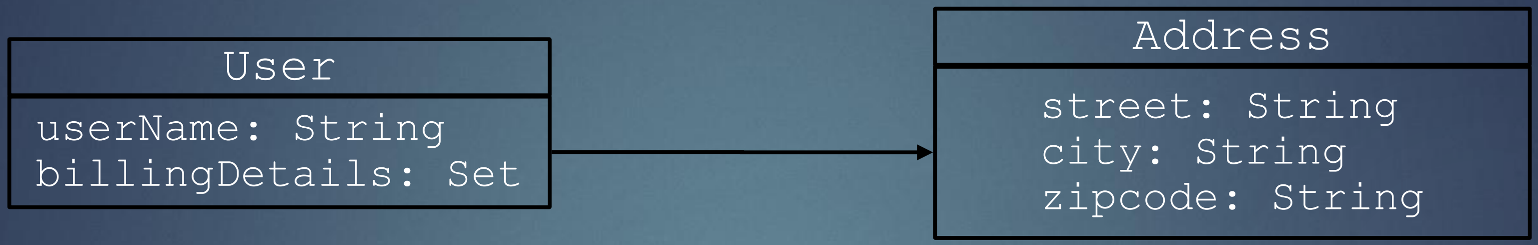
# Creating tables for the Domain Model



```
create table USER (
    USER_NAME      varchar not null primary key,
    ADDRESS        varchar not null)

create table BILLING_DETAILS (
    ACCOUNT_NUMBER varchar not null primary key,
    ACCOUNT_NAME   varchar not null,
    ACCOUNT_TYPE   varchar not null,
    USER_NAME      varchar foreign key references USER)
```

- We’ll see the 5 problems of the O/R paradigm mismatch appear as we gradually make our model more complex…

# The problem of granularity



- should we create a new ADDRESS table?
- should we create a new SQL data type and change the column?
- user-defined data types (UDT) are not portable and the standard is weak

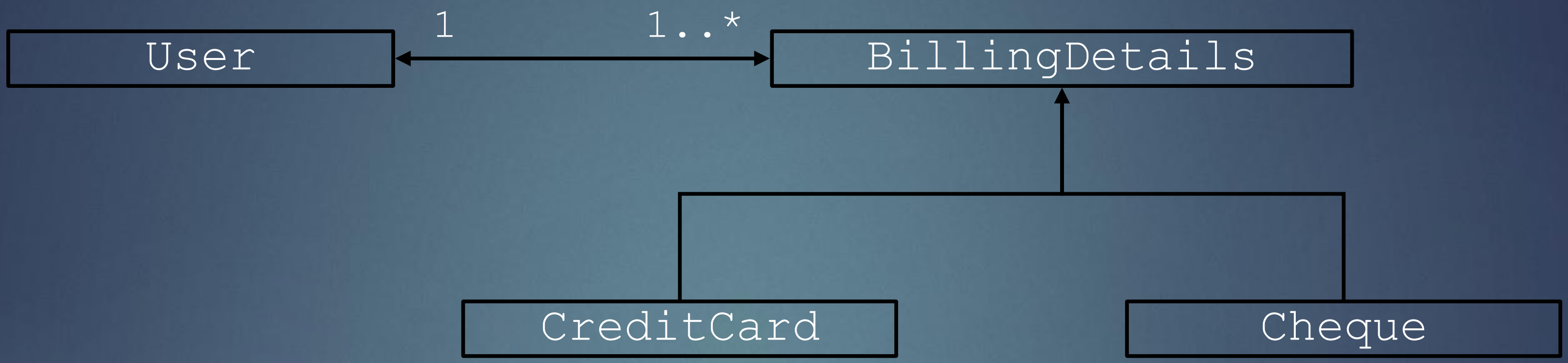
- We usually add new columns to USER with built-in SQL data types:

```
create table USER (
    USER_NAME         varchar not null primary key,
    ADDRESS_STREET    varchar not null,
    ADDRESS_CITY      varchar not null,
    ADDRESS_ZIPCODE   varchar not null)
```

SUJATA BATRA

# The problem of subtypes

We create subclasses of BillingDetails:



User

1

1..*

BillingDetails

CreditCard

Cheque

and use polymorphism in Java to implement our billing strategy.

How do we represent subtypes in our relational model?

## The problem of identity

In Java, we have two notions of "sameness"

- object identity is the memory location of an object, a==b
- object equality (what is this really?), a.equals(b)

In SQL databases, we identify a particular row using the primary key and the table name.

# The problem of associations

- Object-oriented languages represent entity relationships as
  - *object references* (pointers) and *collections* of object references

- Relational databases represent entity relationships as
  - copies of primary key values
  - referential integrity ensured by *foreign key* constraints

- The mismatch:
  - object references are directional, there is no such concept in the relational model
  - many-to-many associations require a *link table* in relational databases

# The problem of object graph navigation

In Java, we "walk" the object graph by following references:

```
david.getBillingDetails().getAccountName()
```

In SQL, we join tables to get the required data:

```
select * from USER u
    left outer join BILLING_DETAILS bd
    on bd.USER_ID = u.USER_ID
    where u.USERNAME = “david"
```

SUJATA BATRA

## The cost of the mismatch

These problems can, at least theoretically, be solved using handwritten SQL/JDBC

- by writing a lot of tedious code
- The “mismatch problem” is real
- better UDT support in SQL will not solve all the issues
- not all applications are suitable for table-oriented approaches

Is the solution design patterns (DAO)

or programming models (EJB entity beans)?

"How should we implement the persistence layer in our application?"

# Implement a Persistence Layer

- **EJB 2.0 entity beans**
  - Bean Managed Persistence
  - Container managed persitence
  - No polymorphic associations and queries
  - Not portable between different application servers
  - Forces an unatural Java Style
- **Object Oriented Database Systems**
  - Not very popular
- **XML persistence**

# Overview of Hibernate

- Object/relational mapping framework for enabling transparent POJO persistence
  - Open Source
- Persistence for JavaBeans, Model is not tied to persistence Implementation
- Powerful and Sophisticated queries (Criteria and HQL)
- Allows developers focus on domain object modeling not the persistence plumbing
- Performance
  - High performance object caching
  - Configurable materialization strategies
- Does not require a container

# Persistent Objects and Collections

- Short-lived, single threaded objects containing persistent state and business function
- These might be ordinary JavaBeans/POJOs,
  - Not associated with (exactly one) Session
- Changes made to persistent objects are reflected to the database tables (when they are committed)
- As soon as the Session is closed, they will be detached and free to use in any application layer

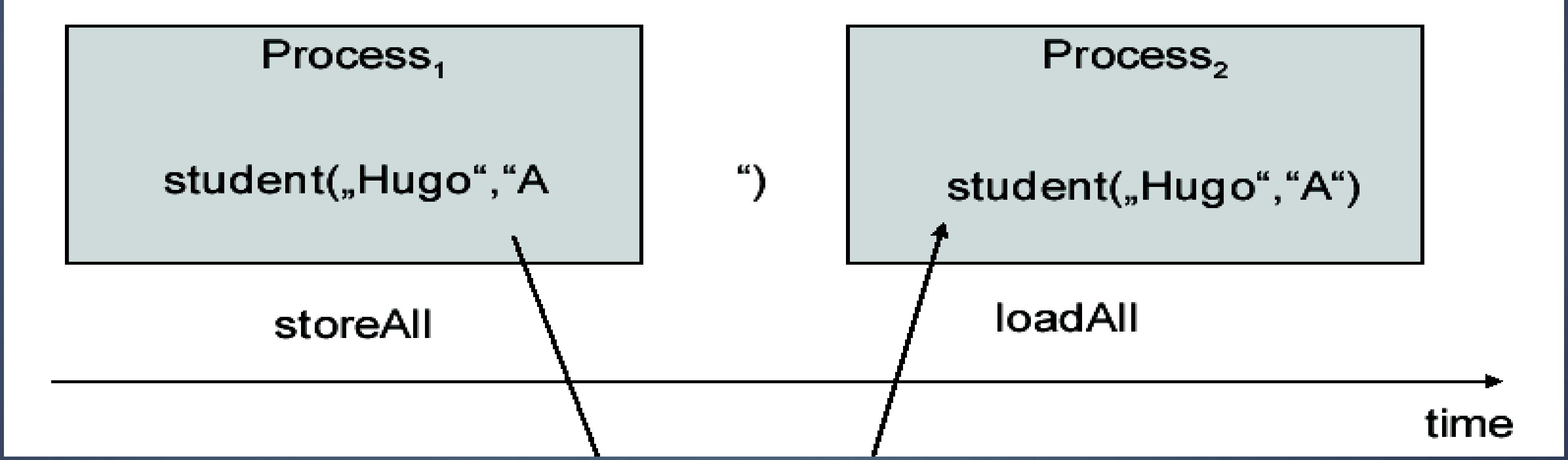
# Entity

- Lightweight persistent domain object – that which is persisted

- Restrictions
  - must have a public or protected no-arg constructor
  - cannot be final
  - cannot have final methods or final instance variables that are to be persisted
  - can be abstract or concrete class
  - must have a primary key

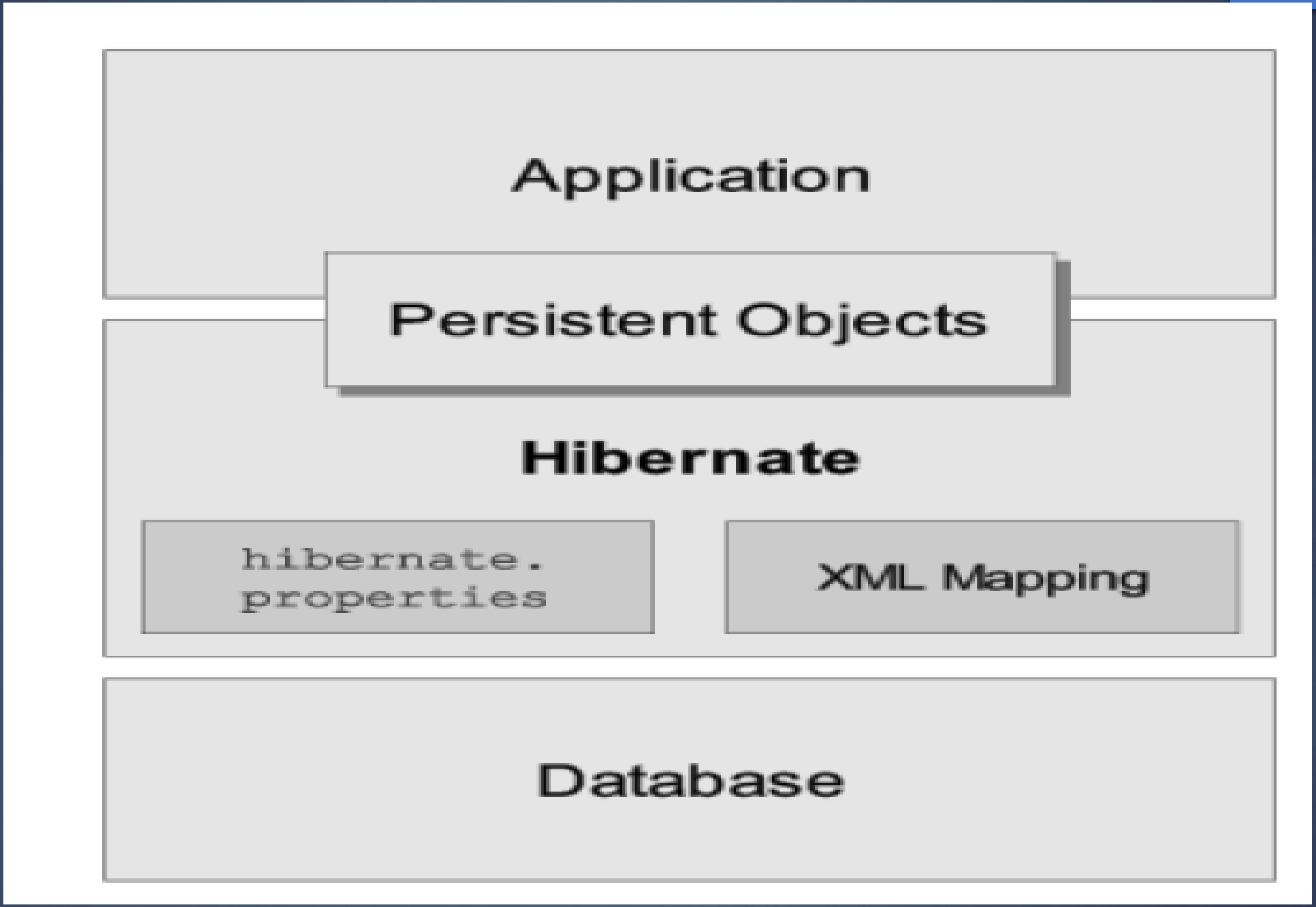
# What is Persistence ?

- Ability of an object to survive even current session or program terminate.
- The ability of an object to remain in existence past the lifetime of the program that creates it.
- Its Achieved by
  - Relational Databases
  - Serialization
  - EJB (entity beans)



SUJATA BATRA

# Hibernate Architecture



# Hibernate Core Interfaces & Classes

- **Session Factory  Interface**
  - Used to create Session objects, created during application initialization, caches generated SQL and other metadata
- **Session  Interface**
  - Primary interface, used to store & retrieve objects
- ***Configuration Class***
  - Used to configure & bootstrap Hibernate
- **Transaction Interface**
    - May not be used by applications
- **Query & Criteria Interface**
    - Query interface is used to run queries in HQL or SQL
    - Criteria interface is also very similar

# SessionFactory

- Represented by *org.hibernate.SessionFactory*
- A factory for *Session* and a client of ***ConnectionProvider***
    - Typically one for each database
    - Maintains a threadsafe (immutable) cache of compiled mappings for a single database
    - Might hold an optional (second-level) cache of data that is reusable between transactions, at a processor cluster-level

# Session

- **Session**
  - Represented by *org.hibernate.Session*
  - The life of a Session is bounded by the beginning and end of a logical transaction.
  - A session represents a persistence context
  - Handles life-cycle operations- create, read and delete operations - of persistent objects
  - A single-threaded, short-lived object representing a conversation between the application and the persistent store
  - Wraps a JDBC connection
  - Factory for *Transaction*

# Configuration

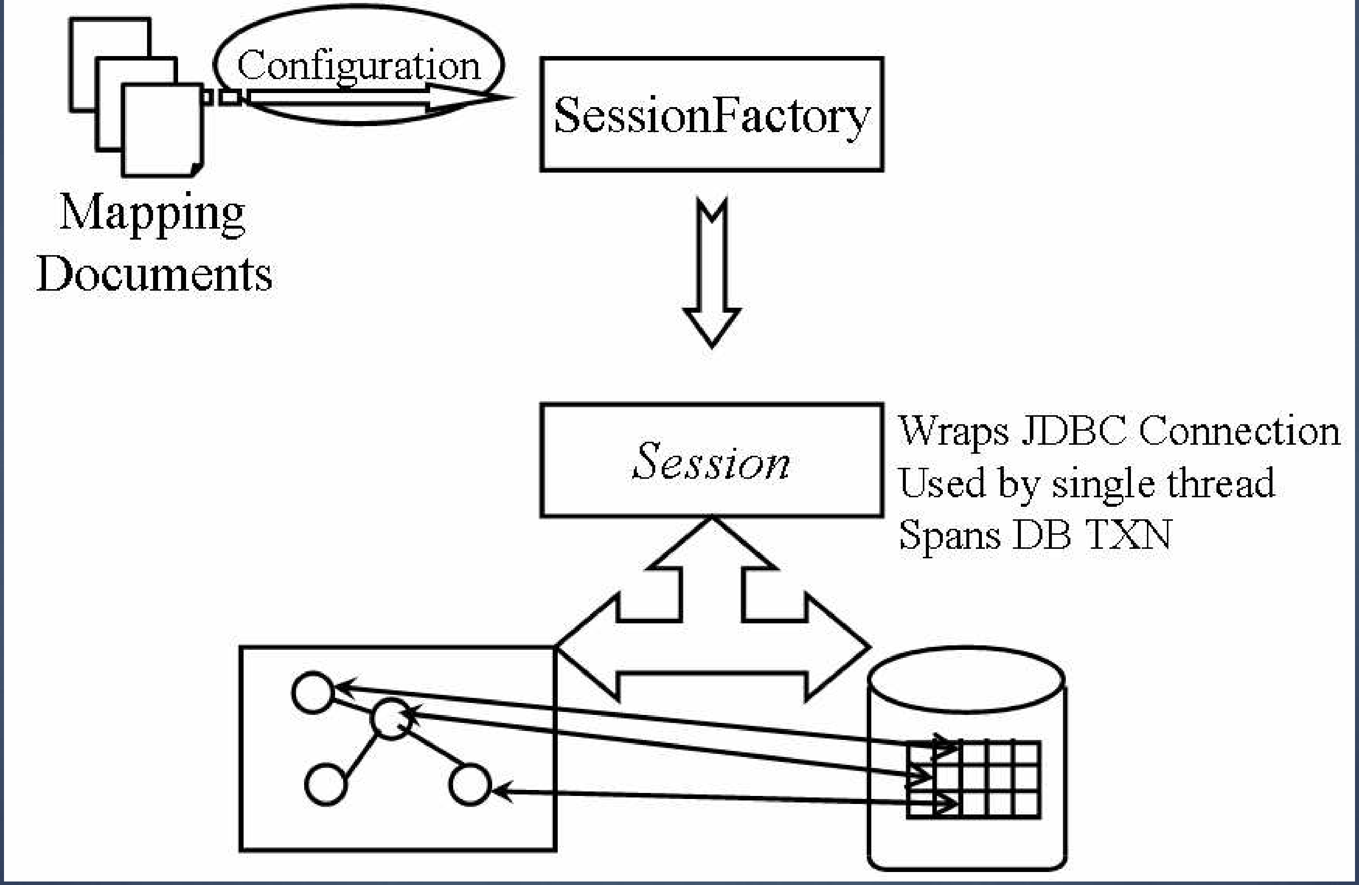
- Represented by org.hibernate.config.Configuration
- Configurations done using **.properties and XML** Files
- Hibernate is part of the app, and so is responsible for getting connections to the database
- Configuration file tells Hibernate how to get the connections
- Hibernate creates and manages a connection pool

# Configuring Hibernate

- Programmatic configuration
- XML configuration file
  - Specify a full configuration in a file named *hibernate.cfg.xml*
  - Configure the class to table mapping in *.hbm.xml
  - By default, is expected to be in the root of your classpath
- Annotation Based Configuration

# Hibernate Mapping

Hibernate uses runtime reflection to determine persistent properties of classes.
A mapping property or configuration file is used to generate database schema and provide persistence



Wraps JDBC Connection
Used by single thread
Spans DB TXN

# Hibernate.cfg.xml

```
<!– DTD →
  <hibernate-configuration>
   <session-factory name="myfact">
     <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
      <property name="hibernate.connection.password">srivatsan</property>
      <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/employee</property>
      <property name="hibernate.connection.username">root</property>
      <property name="show_sql">true</property>
      <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
<property name="hibernate.hbm2ddl.auto">update</property>
      <mapping resource="Invoice.hbm.xml"/>
   </session-factory>
</hibernate-configuration>
```

# Dialects

- MySQL
  - **org.hibernate.dialect.MySQLDialect**
- Oracle
  - **org.hibernate.dialect.OracleDialect**
  - **org.hibernate.dialect.Oracle9Dialect**
- Microsoft SQL Driver
  - **org.hibernate.dialect.SQLServerDialect**
- Postgres
  - **org.hibernate.dialect.PostgreSQLDialect**

# Domain Classes

- Domain classes are classes in an application that implement the entities of the business domain (e.g. Customer and Order in an Ecommerce application)

- Hibernate works best if these classes follow some simple rules, also known as the Plain Old Java Object (POJO) programming model.

- Hibernate assumes very little about the nature of your domain classes
  - You may express a domain model in other ways: using trees of Map instances, for example.

# Steps to write a Domain Class

- **Step 1: Implement a no-argument Constructor**
  - All persistent classes must have a default constructor so that Hibernate can instantiate them using
- **Step 2: Provide an identifier property**
  - This property maps to the primary key column of a database table.
  - The property might have been called anything, and its type might be any primitive type, any primitive "wrapper“ type, *java.lang.String* or *java.util.Date*
  - Composite key is possible
- **Step 3: Declare accessor methods for persistent fields**
  - Hibernate persists JavaBeans style properties

# The POJO Class

```
public class Invoice {

private int invno;
private String customerName;
private double amount;

public Invoice() {
super();
}

public int getInvno() {
return invno;
}
public void setInvno(int invno) {
this.invno = invno;
}
public String getCustomerName() {
return customerName;
//OtherSet/Get Methods
}
```

Identifier property

Constructor –Zero Arg

Accessor/Mutator Methods

# Saving Objects

- To persist the object to the database when save is called with a valid Hibernate session
- An object remains to be in "transient" state until it is saved and moved into "persistent" state
- The class of the object that is being saved must have a mapping file (invoice.*hbm.xml*)

# Getting Session Factory & Saving

```
public static void main(String[] args) {

SessionFactory fact =
        new Configuration().configure().buildSessionFactory();

Session session=fact.openSession();

 Transaction tx = session.beginTransaction();

Customer cust =new Customer(101,”Ramesh”,4040);

            session.save(cust);

              tx.commit();

}
```

# Loading Objects

- Used for loading objects from the database
- Each *load(..)* method requires object's primary key as an identifier
  - The identifier must be *Serializable* – any primitive identifier must be converted to object
- Each *load(..)* method also requires which domain class or entity name to use to find the object with the id
- When Object exist use *load(), - will* throw an exception if the unique id is not found in the database
- *get()* method ,will return null if the unique id is not found in the database
- The load() method may return a proxy instead of a real persistent instance.
- On the other hand, get() never returns a proxy.

# Loading objects

- From Session interface
  - Session ses = sfact.openSession();

```
 Person persObj =
     (Person)sfact.openSession().get(Person.class, 10);

                System.out.println("Name"+persObj.getName());

Person person = (Person) session.get(Person.class, id);
if (person == null){
System.out.println("Person is not found for id " + id);

     ses.close();
```

# Life-cycle Operations and SQL commands

- *save()* result in an *SQL INSERT*
- *delete()* results in an *SQL DELETE*
- *update()*  results in an *SQL UPDATE*
- Changes to persistent instances are detected at flush time and also result in an *SQL UPDATE*

# Updating Objects

- Hibernate automatically manages any changes made to the persistent objects
- The objects should be in "persistent" state not transient state
- If a property changes on a persistent object, Hibernate session will perform the change in the database when a transaction is committed (possibly by queuing the changes first)
- From developer perspective, you do not have to any work to store these changes to the database
- You can force Hibernate to commit all of its changes using *flush()* method

# Updating Objects

- Get a Persitent Object by Calling Load
- Set a new Value for the field that need to be updated
- Call the update Method

```
Invoice inv = (Invoice)sess.load(Invoice.class, new
  Integer(idno));

        inv.setCustomerName("Ganesh Kumar");

        sess.update(inv);

        sess.getTransaction().commit();
```

# Deleting Objects

- Remove a persistent instance from the datastore.

- The argument may be an instance associated with the calling 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".

```
Invoice inv = (Invoice)sess.load(Invoice.class, new Integer(idno));

       sess.delete(inv);

       sess.getTransaction().commit();
       System.out.println("Deleted");
```

# Hibernate Life Cycle

transient state

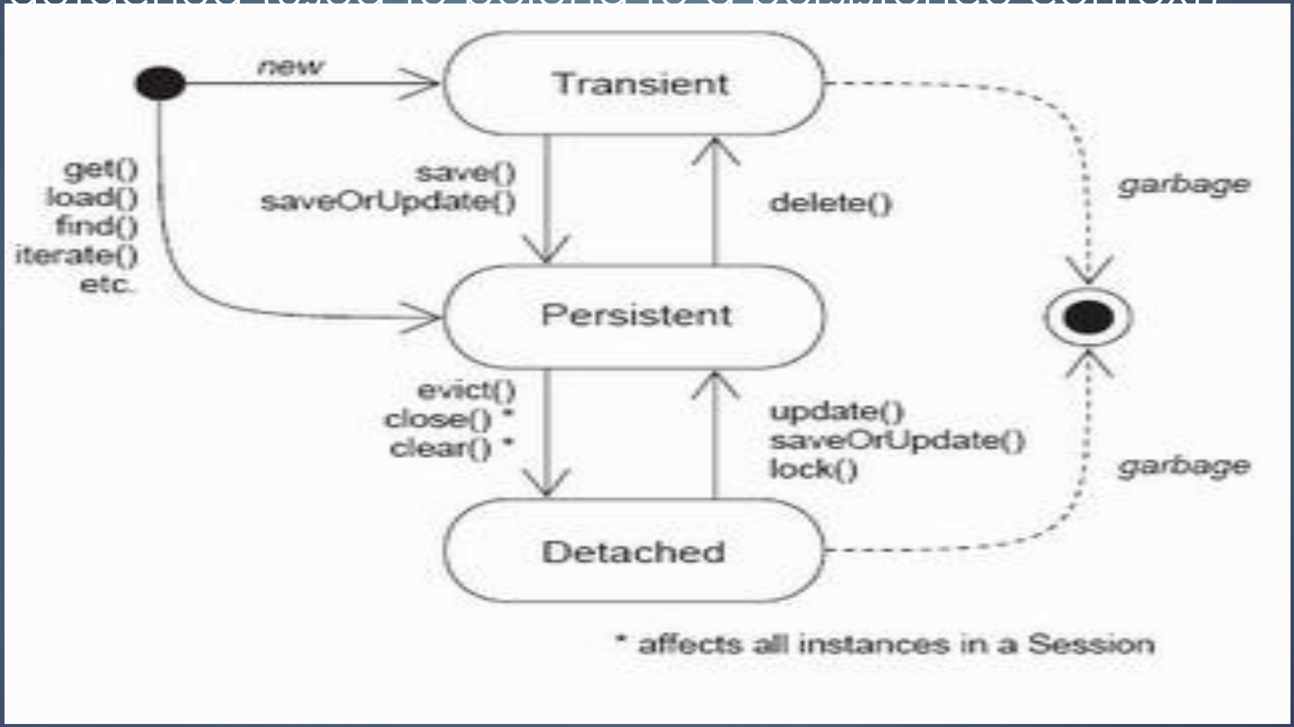
```
Session sess = sf.openSession();
        Order o=new Order();          // search and return
      Query q = session.createQuery("from Order order where”+
                                                 +”order.id=:id");
      q.setString(“id”,name);
       List result = q.list();
        if (list.size() == 0) {
            System.out.println("No Order having id "
                                  + name);
            System.exit(0);
        }
        o = (Order) list.get(0);
        sess.close();

        o.getOrderDate();
```

Persistent state

Detached State

# Instance States

- An instance of a persistent classes may be in one of three different states, which are defined with respect to a persistence context
  - transient (does not belong to a persistence context)
  - persistent (belongs to a persistence context)
  - detached (used to belong to a persistence context)
-



# Instance States

- **“transient” state**
  - The instance is not, and has never been associated with any session (persistence context)
  - It has no persistent identity (primary key value)
  - It has no corresponding row in the database – ex) When POJO instance is created outside of a session
- **“persistent” state**
  - The instance is currently associated with a session (persistence context).
  - It has a persistent identity (primary key value) and likely to have a corresponding row in the database – ex) When an object is created within a session or a transient object gets persisted

# Instance States

- **“detached”**
  - The instance was once associated with a persistence context, but that context was closed, or the instance was serialized to another process
  - It has a persistent identity and, perhaps, a corresponding row in the database
  - Used when POJO object instance needs to be sent over to another program for manipulation without having persistent context

# State Transitions

- Transient instances may be made persistent by calling *save()*, *persist()* or *saveOrUpdate()*
- Persistent instances may be made transient by calling *delete()*
- Any instance returned by a *get()* or *load()* method is persistent
- Detached instances may be made persistent by calling *update()*, *saveOrUpdate()*, *lock()* or *replicate()*
- The state of a transient or detached instance may also be made persistent as a new persistent instance by calling *merge()*.

# saveOrUpdate

- if the object is already persistent in this session, do nothing
- if another object associated with the session has the same identifier, throw an exception
- if the object has no identifier property, save() it
- if the object's identifier has the value assigned to a newly instantiated object, save() it

# Persist

- **Save and persist return type:**

- Both INSERT records into database but **return type of persist is void** while return type of save is Serializable object.

- **transient to instance persistent**.

- persist() method doesn't guarantee that the identifier value will be assigned to the persistent instance immediately, the assignment might happen at flush time.

- **behavior on outside of transaction boundaries.**

- persist() guarantees that it will **not** execute an INSERT statement if it is called outside of transaction boundaries.

- save() method **does not** guarantee the same, if you are inside or outside of a transaction.

# Clear()

- it explicitly is to remove all managed entities from L1 cache, so that it does not grow infinitely when processing large data sets in one transaction.

- It discards all the changes that were made to managed entites not explicitly persisted.

- This means that you can safely modify an entity, update it explicitly and clear the session.

# Hibernate Mapping

SUJATA BATRA

# Association -Relationship

- Association is a relationship of one class to another class
  - Its 'relationships' in database terminology.
- Database design involves the master detail tables to represent the relationship of one entity to another.
- **A many-to-one Association**
  - Each book is related to one publisher
- **A one–to- Many Association**
  - One publisher may publish many books.
- **A One-To-One Association**
  - One Student has one Teacher
- **A Many-To-Many**
  - Many customers have many bank accounts
- **“unidirectional” association.**
  - If Association is navigable from book to publisher only

SUJATA BATRA

# Simple Association (One-to-One)

- Expresses a relationship between two classes where each instance of the first class is related to a single instance of the second or vice versa

- Can be expressed in the database in two ways

1. Giving each of the respective tables the same primary key values

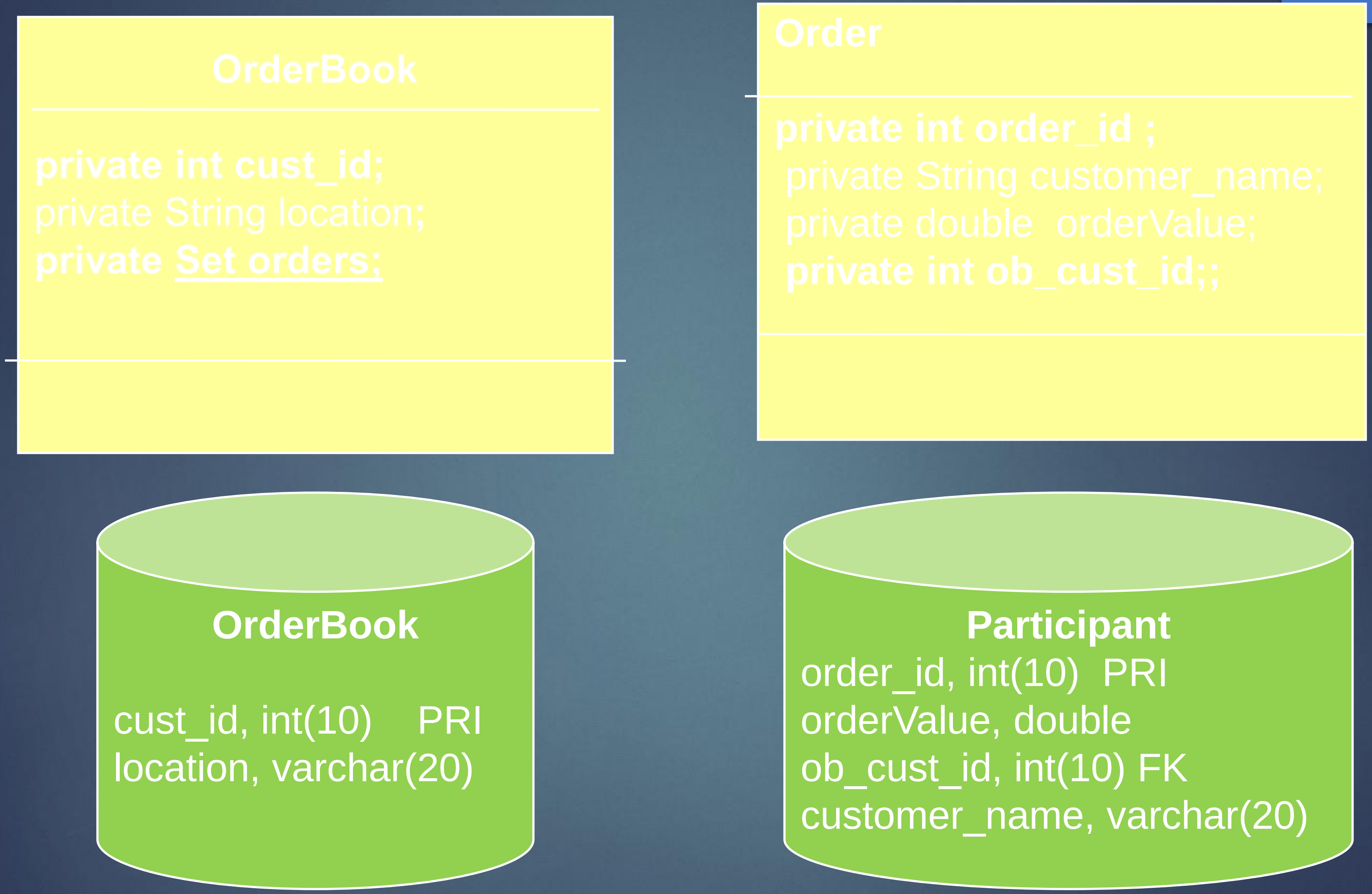
2. Using Foreign Key constraint from one table onto a unique identifier Column of the other.

# One-to-Many-Relationship

- A one-to-many reference is basically a collection. Here class, OrderBook, holds a reference to a collection of another class, Order.
- In general <set> type collections are hibernate supports various types of collections
- We create an extra column in table Order which holds the FK to table OrderBook
- This allows OrderBook to be assigned a collection of Order based on the value of the cust_id column in ob_cust_id

SUJATA BATRA

# 1-N- “UD” Class and Table Design



**OrderBook**

**private int cust_id;**
private String location;
**private Set orders;**

**Order**

**private int order_id ;**
private String customer_name;
private double  orderValue;
**private int ob_cust_id;;**

**OrderBook**

cust_id, int(10)    PRI
location, varchar(20)

**Participant**
order_id, int(10)  PRI
orderValue, double
ob_cust_id, int(10) FK
customer_name, varchar(20)

# Hibernate Query

SUJATA BATRA

# Hibernate Query Language (HQL)

- Very similar to SQL but less verbose
- Understands OO – inheritance, polymorphism, association, aggregation, composition
- Selection: *from, as*
- Associations and joins: *inner join, outer join, right outer join, full join*
  - Projection: *select, elements*
  - Constraints: *where*
  - Other constructs: *aggregate functions, expressions, order by clauses, group by clauses, polymorphic selections, sub-queries*
- **Differences from SQL**
- HQL is fully object-oriented, understanding notions like inheritance, polymorphism and association
- Queries are case-insensitive, except for names of Java classes and properties

# Hibernate Query Capabilities

- Criteria Queries
  - Extensible framework for expressing query criteria as objects
  - Includes “query by example”

- Native SQL Queries

- Enhanced support for queries expressed in the native SQL dialect of the database
  - “from” clause
  - Associations and join
  - “select” clause
  - Polymorphic query
  - “where” clause

SUJATA BATRA

# Querying Objects

- The **list()** Method of the Query retrieves the result list containing objects

```
– Query query = session.createQuery("from Book");
– List books = query.list();
```

- To get One unique object returned as result,
  - **"?" to represent a query parameter and set it by index , which is zero-based not one-based as in JDBC**.

```
Query query = session.createQuery("from Book where isbn = ?");
query.setString(0, "1932394419");
Book book = (Book) query.uniqueResult();
```

```
Query query = session.createQuery("from Book where isbn =
   :isbn");
query.setString("isbn", "1932394419");
Book book = (Book) query.uniqueResult();
```

# where clause

- The where clause allows you to narrow the list of instances returned.
- If no alias exists, you may refer to properties by name

-
  ```
  Query qry =  sess.createQuery("from Invoice where invNo=:ino");
       qry.setInteger("ino", 1001);
       Invoice inv = (Invoice)qry.uniqueResult();
  ```
- If there is an alias, use a qualified property name:

```
Query qry = sess.createQuery("from Book as bk where bk.id=?");

qry.setInteger(0, pubId);
```

# select clause

- Queries may return properties of any value type including properties of component type

  - **Query qry =mySess.createQuery("select invNo,customer from Invoice");**

```
List myList = qry.list();

for(int i =0;i<myList.size();i++)
{
            Object[] obj = (Object[]) myList.get(i);

            System.out.println(obj[0]);
            System.out.println(obj[1]);
}
```

**Query qry =mySess.createQuery("select  new Invoice(invNo,customer) from Invoice“)**

## Aggregate functions

HQL queries may even return the results of aggregate functions on properties:

```
select avg(cat.weight), sum(cat.weight),
    max(cat.weight), count(cat)
from Cat cat
```

The supported aggregate functions are

- avg(...), sum(...), min(...), max(...)
- count(*)
- count(...), count(distinct ...), count(all...)

You may use arithmetic operators, concatenation, and recognized SQL functions in the select clause:

```
select cat.weight + sum(kitten.weight)
from Cat cat
```

# Native SQL Query

- String sql="select * from invoice_table where invNumber=101 and custName='Ramesh'";

Query qry= mySession.createSQLQuery(sql).addEntity(Invoice.**class);**

- To return a list of scalars or values as Object[] for each column.

•Hibernate uses ResultSetMetadata to infer column types can explicitly denote return types using addScalar().

ResultSetMetadata to deduce the actual order and types

```
sess.createSQLQuery("SELECT * FROM Customer").list();

sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM
    Customer").list();
```

SUJATA BATRA

# Native SQL Query

```
public List<Customer> scalarQuery()  {

Session sess=fact.openSession();


Query qry =sess.createSQLQuery
("Select customerNumber,CustomerName from Hiber_CUSTOMER ").
 addScalar("customerNumber",Hibernate.~~INTEGER).~~
addScalar("customerName",Hibernate.~~STRING);~~
    List<Customer> custList =qry.list();
        sess.close();
     return custList;
 }

for(int i =0;i<custList.size();i++)    {
 Object[] val =(Object[]) custList.get(i);
 System.out.println(val[0]);
```

# Named Queries

- HQL statements in the mapping definitions aree called “Named Queries”,The named queries can be put in any mapping definitions.

- For each named query, we need to assign a unique name to it. We should also put the query string in a <![CDATA[...]]> block to avoid conflicts with the special XML characters.

```
<hibernate-mapping package="com.training">
   <class name="Invoice" table="Invoice">
    - - - -
   </class>
 <query name="amountQry">
    <![CDATA[from Invoice where invAmount>? ]]>
</query>
 </hibernate-mapping>
```

```
     Session sess = HibernateUtil.getSession();

         Query qry = sess.getNamedQuery("amountQry");
         qry.setDouble(0, 1000);

    List<Invoice> invList = qry.list();
```

# Using Stored Procedure

```
CREATE PROCEDURE `GetStocks`(int_stockcode VARCHAR(20))
BEGIN
  SELECT * FROM stock WHERE stock_code = int_stockcode;
  END $$

DELIMITER ;

Query query = session.createSQLQuery("CALL GetStocks(:stockCode)")
    .addEntity(Stock.class)     .setParameter("stockCode", "7277");

List result = query.list();
for(int i=0; i<result.size(); i++) {
    Stock stock = (Stock)result.get(i);
    System.out.println(stock.getStockCode());
}
```

# Hibernate Annotations

- @Entity - class level
  - POJO class can be declared as an entity
- @Id - Property Level or at all the getXXX Level
  - declares the identifier property of the entity.
- @Table - class level
  - to define the table, catalog, and schema names for entity mapping.
- @Table(name="tbl_name",

  uniqueConstraints =

# Hibernate Annotations

```
import javax.persistence.*;

@Entity
@Table(name = "employee")

public class Employee {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "id")
    Integer id;
```

# Using Annotations

- In the Hibernate configuration File
- **<mapping class="com.training.CreditCard"/>**
- In the Application Class
- **SessionFactory session = new AnnotationConfiguration().configure().buildSessionFactory();**

SUJATA BATRA