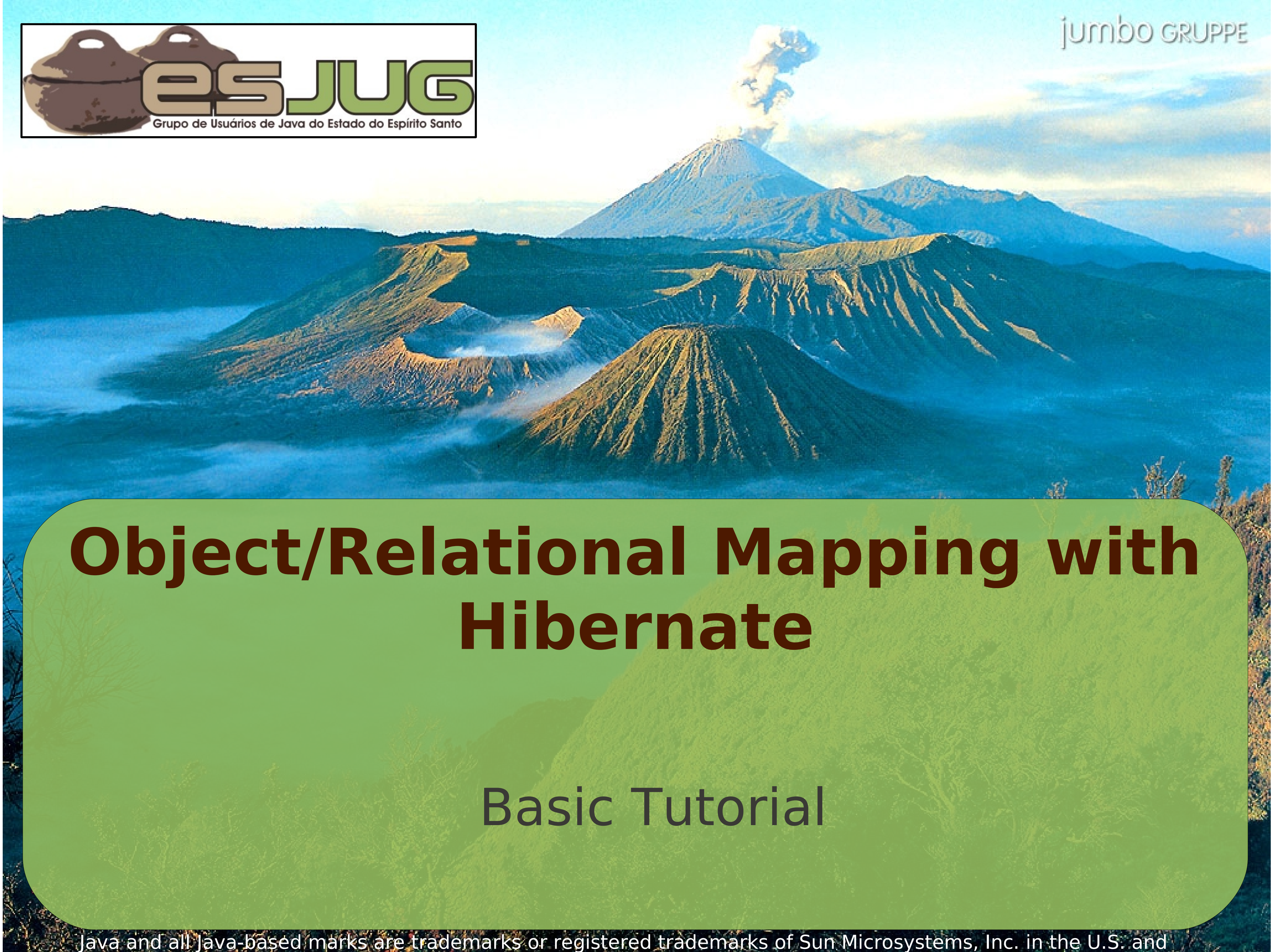


Grupo de Usuários de Java do Estado do Espírito Santo

jumbo GRUPPE

Object/Relational Mapping with Hibernate

Basic Tutorial

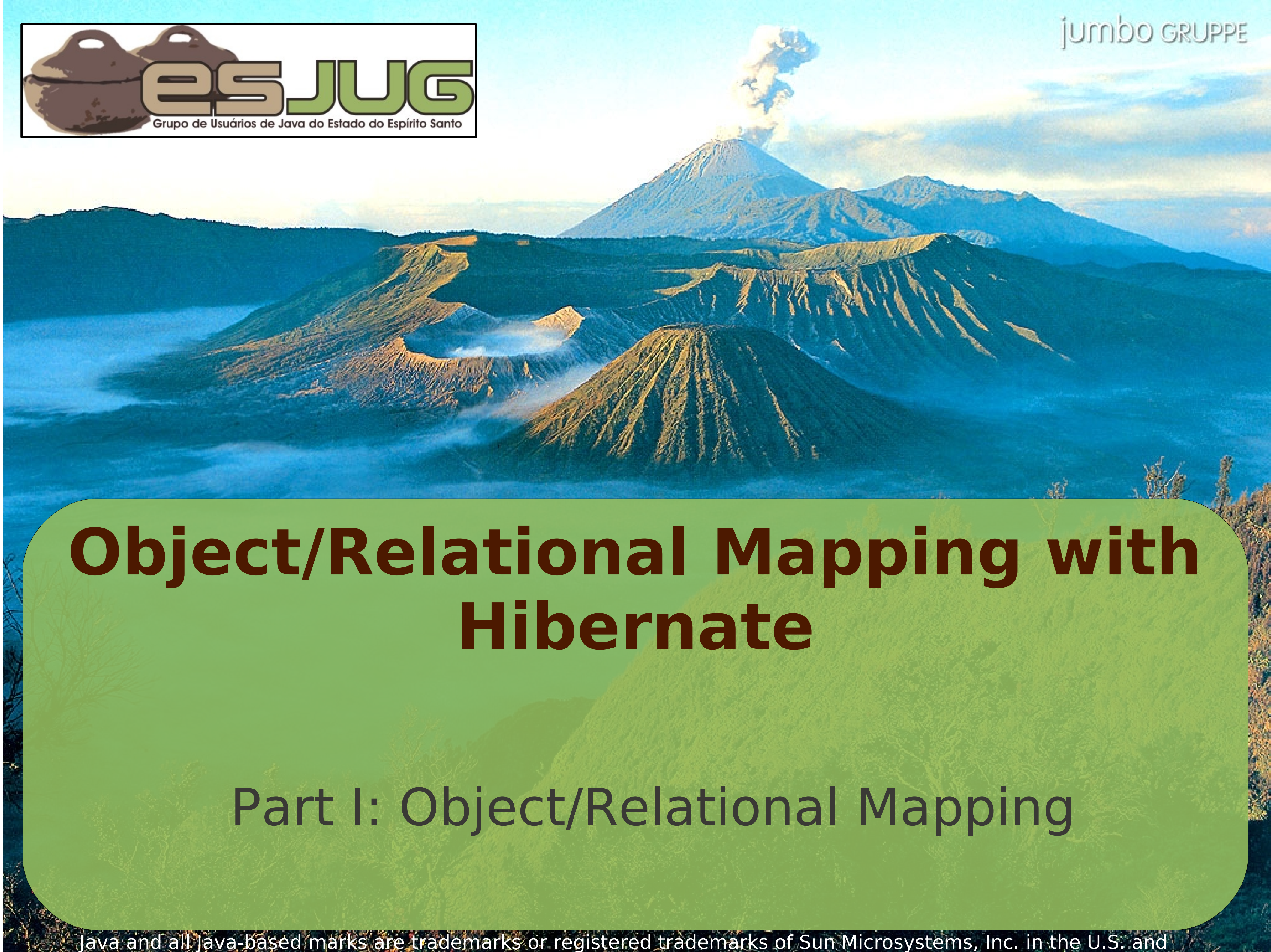
License for use and distribution

About the author – Vítor Souza

- Education:
  - Computer Science graduate, masters in Software Engineering – Federal University of Espírito Santo (Brazil).
- Java:
  - Developer since 1999;
  - Focus on Web Development;
  - Co-founder and coordinator of ESJUG (Brazil).
- Professional:
  - Substitute teacher at Federal University of ES;
  - Engenho de Software Consulting & Development.
- Contact: vitorsouza@gmail.com

Goals

- Learn about object/relational mapping, a “new” way of persisting data;
- Know the basic concepts of Hibernate;
- Be able to build informations systems using Hibernate.



esJUG
Grupo de Usuários de Java do Estado do Espírito Santo

Object/Relational Mapping with Hibernate

Part I: Object/Relational Mapping

What is persistence?

- Ability to preserve user data after the software has already been closed;
- In Java, many ways to do it:
  - Writing directly to files (text or binary);
  - Serialization (with or without a framework);
  - Relational databases (RDBMS) with JDBC;
  - OO databases (OODBMS);
  - Etc.
- For information systems, RDBMS is the most common case.

Use of RDBMS in Java

- Communication through SQL queries:
  - Create and alter tables;
  - Insert, update and delete data;
  - Constraints, projections and junctions;
  - Grouping, ordering, agregation functions;
  - Etc.
- Connection to the database via driver;
- Standardization of the JDBC API;
- Tedious and error-prone task.

Object oriented software

- Simple software can be built on top of the JDBC access logic;
  - Business logic would work with lines and columns.
- More complex software have a domain model:
  - Classes that represent objects from the problem domain;
  - Utilization of OO concepts such as polymorphism;
  - Business logic works with objects.

The paradigm mismatch

- Being discussed for 15 years;
- Representing data in tables is very different than a graph of interconnected objects;
- The problems:
  - Granularity: limited to table and column;
  - Inheritance: storage and polymorphism;
  - Identity: == vs. equals() vs. primary-key;
  - Associations: transposition of primary-keys;
  - Navigation on the object graph: the n+1 SELECTs problem.

The cost of incompatibility

- Approximately 30% of the code is to manipulate data using SQL/JDBC;
- Similar structures are repeated in INSERT, UPDATE and SELECT commands;
- The domain model is twisted to adequate itself to the data model;
- Software becomes hard to maintain.

Alternatives to persistence

- To divide in layers is common sense;
- Alternatives to the persistence layer:

Hand-coded SQL/JDBC	Wasted effort, low productivity and high maintenance, possibly lower performance compared to existing solutions.
Serialization	Access to entire graph, no searching, concurrency issues.
Entity EJBs	Twists the object model, no support for polymorphism and inheritance, not portable in practice, not serializable, intrusive model that makes unit testing very hard.
OO Databases	Low market acceptance, immature standard.

Object/relational mapping

- An ideal solution to the problem;
- Also known as:
  - ORM, O/RM, O/R mapping, etc.;
  - *Gateway-based Object Persistence* (GOP).

In a nutshell, object/relational mapping is the automated (and transparent) persistence of objects in a Java application to the tables in a relational database, using metadata that describes the mapping between the objects and the database. ORM, in essence, works by (reversibly) transforming data from one representation to another.

Hibernate in Action

Components of an ORM solution

- API for the execution of CRUD operations;
- Language or API for the construction of queries that refer to the classes and their properties;
- Specification of the mapping metadata;
- RDBMS interaction techniques, including:
  - Dirty checking;
  - Lazy association fetching;
  - Optimization functions.

The ORM solution should specify

- How persistent classes and metadata should be written;
- How to map hierarchies of classes;
- How do object identity and table line identity relate;
- What is the lifecycle of a persistent object;
- How to retrieve data from associations in an efficient way;
- How to manage transactions, cache and concurrency.

Why use ORM?

- Productivity:
  - Eliminates plumbing code.
- Maintainability:
  - Less lines of code, less maintenance;
  - Changes in data structure do not impact as much.
- Performance:
  - More time to implement optimizations;
  - More knowledge of each RDBMS detail.
- Vendor independence:
  - Use of SQL dialects.

The truth about ORM frameworks

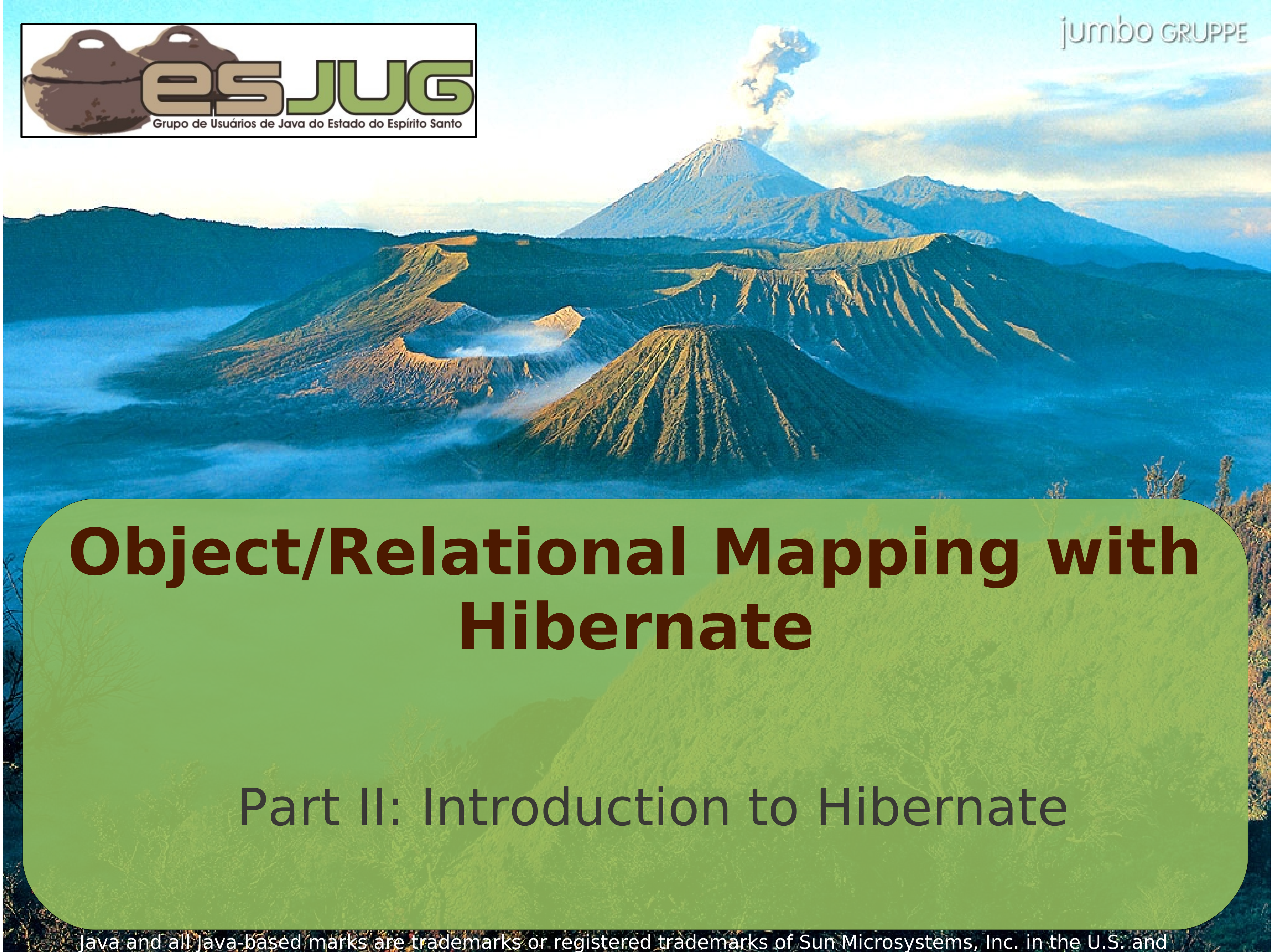
- They are not easy to learn;
- To use them well, you should also know well SQL and relational database technology;
- Problems that come from their use are complex and hard to solve;
- They are not the “silver bullet” of persistence!

Conclusions of part I

- Persistence is a common requirement in information systems and there are many possible solutions;
- The relational/OO paradigm mismatch make the issue more complex;
- Object/Relational Mapping (ORM) is one of the possible solutions for the issue:
  - Has a lot of advantages, such as productivity, maintainability, performance, etc.;
  - Has costs of complexity;
  - Is not the “silver bullet” of persistence.

Grupo de Usuários de Java do Estado do Espírito Santo

jumbo GRUPPE



Object/Relational Mapping with Hibernate

Part II: Introduction to Hibernate

Overview

- Hibernate is the most well-known ORM framework
- Implements everything that is expected from a complete ORM solution;
- Steps for its utilization:
  - Download and install;
  - Write persistent classes;
  - Create tables on the RDBMS;
  - Define the O/R mapping;
  - Configure the framework;
  - Use its API to query and manipulate.

Download and install

- Required files:
  - Hibernate distribution (www.hibernate.org);
  - Hibernate Annotations (idem);
  - HSQLDB database (www.hsqldb.org).
- Using the Eclipse IDE:
  - The Hibernate Tools plug-in can help;
  - Not used in this tutorial, though.
- Add the required libraries to the project's Build Path in Eclipse (`lib` folder).

Required libraries

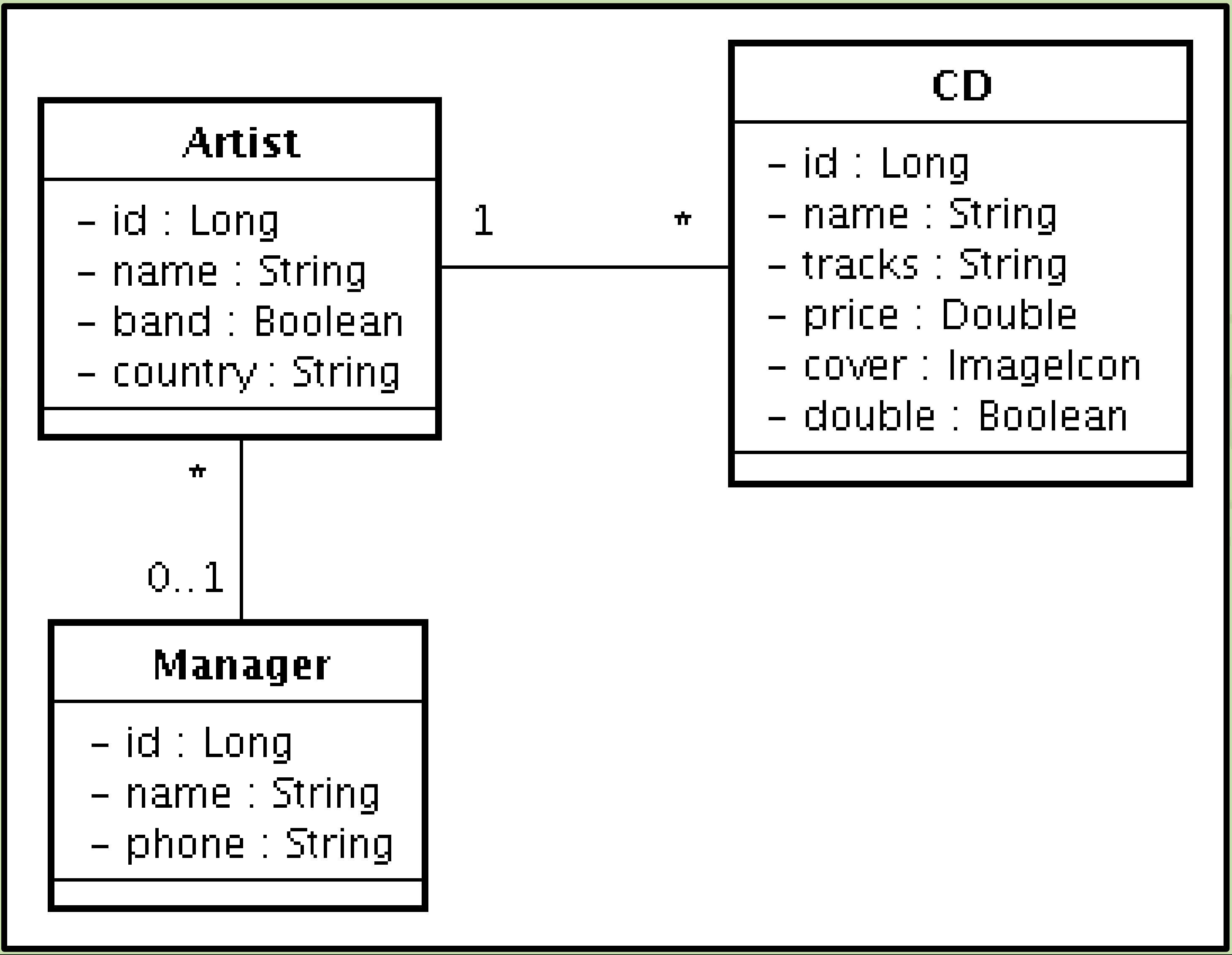
- antlr: ANother Tool for Language Recognition;
- asm-attrs: ASM bytecode library;
- asm: ASM bytecode library;
- c3p0: JDBC connection pool;
- cglib: bytecode generation;
- commons-collection: Commons Collection;
- commons-logging: Commons Logging;
- dom4j: parser for the mapping and configuration;
- ehcache: cache provider;

Required libraries

- hibernate3: Hibernate 3;
- hsqldb: HSQLDB database;
- jaxen: optional, use for the desserialization of the configuration (performance increase);
- jdbc2_0-stdext: Standard Extension JDBC APIs (mandatory outside an *Application Server*);
- jta: Standard JTA API (ditto);
- log4j: logging tool;
- ejb3-persistence e hibernate-annotations: Hibernate Annotations.

Example system: Java Discs

- We will use a CD store as example in this tutorial:



The persistent class

```
package hibernatetutorial.domain;

public class Artist {
   private Long id;

   private String name;

   private Boolean band;

   private String country;

   /* Implicit construtor. */

   /* Properties getters and setters. */
}
```

The persistent class

- A common class (POJO);
- Hibernate is not intrusive:
  - Only exception: the class must have a no-args constructor (but it can be `private`);
- There are some recommendations:
  - Each property should have a getter and a setter;
  - The class should have an identity property.
- Not following the recommendations can complicate Hibernate's use.

Running HSQLDB

- Configuration (server.properties):

```
server.database.0=file:javadiscs
server.dbname.0=javadiscs
```

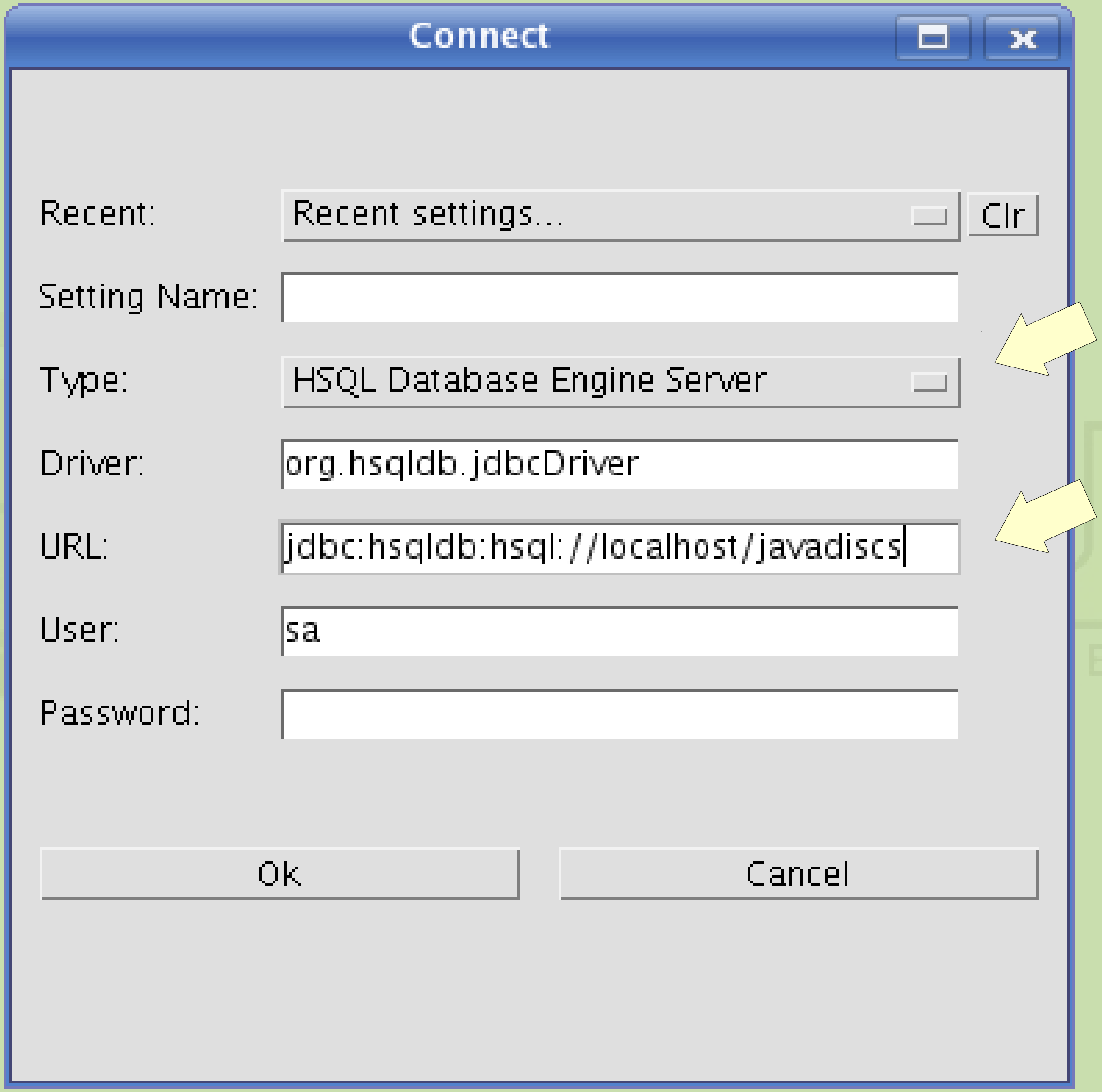
- Running the server:

```
java -cp hsqldb.jar org.hsqldb.Server
```

- Running the manager:

```
java -cp hsqldb.jar org.hsqldb.util.DatabaseManager
```

Connecting with HSQLDB Manager



Recent:	Recent settings...
Setting Name:	
Type:	HSQL Database Engine Server
Driver:	org.hsqldb.jdbcDriver
URL:	jdbc:hsqldb:hsql://localhost/javadiscs
User:	sa
Password:	

The table on the RDBMS

```
CREATE TABLE Artist (
  id BIGINT NOT NULL IDENTITY,
  name VARCHAR(100) NOT NULL,
  band BIT NULL,
  country VARCHAR(50) NOT NULL,
  PRIMARY KEY(id)
);
```

- Created manually;
- Hibernate has tools for automatic generating tables from classes (and vice-versa).

The class' O/R mapping

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping ... >
<hibernate-mapping>
  <class name="hibernatetutorial.domain.Artist"
                                  table="Artist">
    <id name="id" column="id">
              <generator class="native" /></id>
    <property name="name" column="name"
                  type="string" length="100" />
    <property name="band" column="band"
                             type="boolean" />
    <property name="country" column="country"
                  type="string" length="100" />
  </class>
</hibernate-mapping>
```

The class' O/R mapping

- XML file, preferably one per class;
  - Specifies class, table and each property with respective column, type and constraints;
- Avoiding the metadata hell:
  - Hibernate has sensible defaults;
  - We can replace by annotations (later).
- By default, mapping files should be in the same directory as the mapped class.

Use of sensible defaults

```
<class name="hibernatetutorial.domain.Artist">
   <id name="id">
      <generator class="native" />
   </id>
   <property name="name" length="100" />
   <property name="band" />
   <property name="country" length="100" />
</class>
```

- Table name = class name;
- Column name = property name;
- Column type inferred through reflection.

Saving data with Hibernate

```java
// Imports from org.hibernate.*

// Creates an object.
Artist artist = new Artist();
artist.setName("Dave Matthews Band");
artist.setBand(true);
artist.setCountry("USA");

// Obtains a session (implemented later).
Session session = HibernateUtil.openSession();

// Stores on database using Hibernate.
Transaction tx = session.beginTransaction();
session.save(artist);
tx.commit();
session.close();
```

Retrieving an object given its id

```
// Obtains a session (implemented later).
Session session = HibernateUtil.openSession();

// Retrieves the artist with id = 1.
Transaction tx = session.beginTransaction();
Artist artist = (Artist)
        session.load(Artist.class, 0l);

// Prints and closes the connection.
System.out.println(artist.getName());
tx.commit();
session.close();
```

Retrieving objects with queries

```java
// Obtains a session (implemented later).
Session session = HibernateUtil.openSession();

// Retrieves all artists.
Transaction tx = session.beginTransaction();
Query query = session.createQuery(
               "from Artist a order by a.name");
List result = query.list();

// Prints and closes the connection.
for (Object o : result) System.out.println(o);
tx.commit();
session.close();

// Artist must implement toString() for printing.
```

Configuring the framework

- The missing piece for our example:

```
// Obtains a session (implemented later).
Session session = HibernateUtil.openSession();
```

```
public final class HibernateUtil {
   private static SessionFactory sessionFactory;
   private static SessionFactory getSessionFactory() {
      if (sessionFactory == null) sessionFactory = new
Configuration().configure().buildSessionFactory();
      return sessionFactory;
   }
   public static Session openSession() {
      return getSessionFactory().openSession();
   }
}
```

Alternatives for configuration

- There are four configuration options:
  - Programatic (assembling a Properties object and calling configuration methods);
  - System properties: `java -Dkey=value`;
  - The file `hibernate.properties`;
  - The file `hibernate.cfg.xml`.
- Files (properties or XML):
  - Most common case;
  - Should be in the root of the classpath;
  - Automatically found by `configure()`.

Configuration parameters

- What is the database driver, URL, user and password for the creation of JDBC connections?
- Which connection pool is going to be used?
- What is the SQL Dialect of the database?
- Should Hibernate print the SQL commands it generates?
- Where are the classes' mapping files?
- Etc.

hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration ... >

<hibernate-configuration>
 <session-factory>

   <!-- RDBMS configuration. -->
   <property name="connection.driver_class">
     org.hsqldb.jdbcDriver
   </property>
   <property name="connection.url">
     jdbc:hsqldb:hsql://localhost/javadiscs
   </property>
   <property name="connection.username">sa</property>
   <property name="connection.password"></property>
```

hibernate.cfg.xml

```
<!-- Connection pool (using built-in). -->
<property name="connection.pool_size">1</property>

<!-- SQL Dialect. -->
<property name="dialect">
   org.hibernate.dialect.HSQLDialect
</property>

<!-- Automatic session management. -->
<property name="current_session_context_class">
   thread
</property>

<!-- Second-level cache is disabled. -->
<property name="cache.provider_class">
   org.hibernate.cache.NoCacheProvider
</property>
```

hibernate.cfg.xml

```
   <!-- Print SQL commands. -->
   <property name="show_sql">true</property>

   <!-- Mappings: -->
   <mapping
 resource="hibernatetutorial/domain/Artist.hbm.xml"
   />

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

```
Configuration cfg = new Configuration();

// Looks for hibernate.properties & hibernate.cfg.xml.
cfg.configure();

sessionFactory = cfg.buildSessionFactory();
```

About the connection pool

- Managed set of JDBC connections;
- Motivation:
  - Creating a new connection is costly;
  - Having an idle connection is waste of resources.
- Provided by application servers;
- Hibernate comes with C3P0, DBCP and Proxool;
- There is a default connection pool, although not recommended for production environments.

c3p0 configuration example

```
<property name="c3p0.min_size">5</property>
<property name="c3p0.max_size">20</property>
<property name="c3p0.timeout">1800</property>
<property name="c3p0.max_statements">50</property>
```

Logging

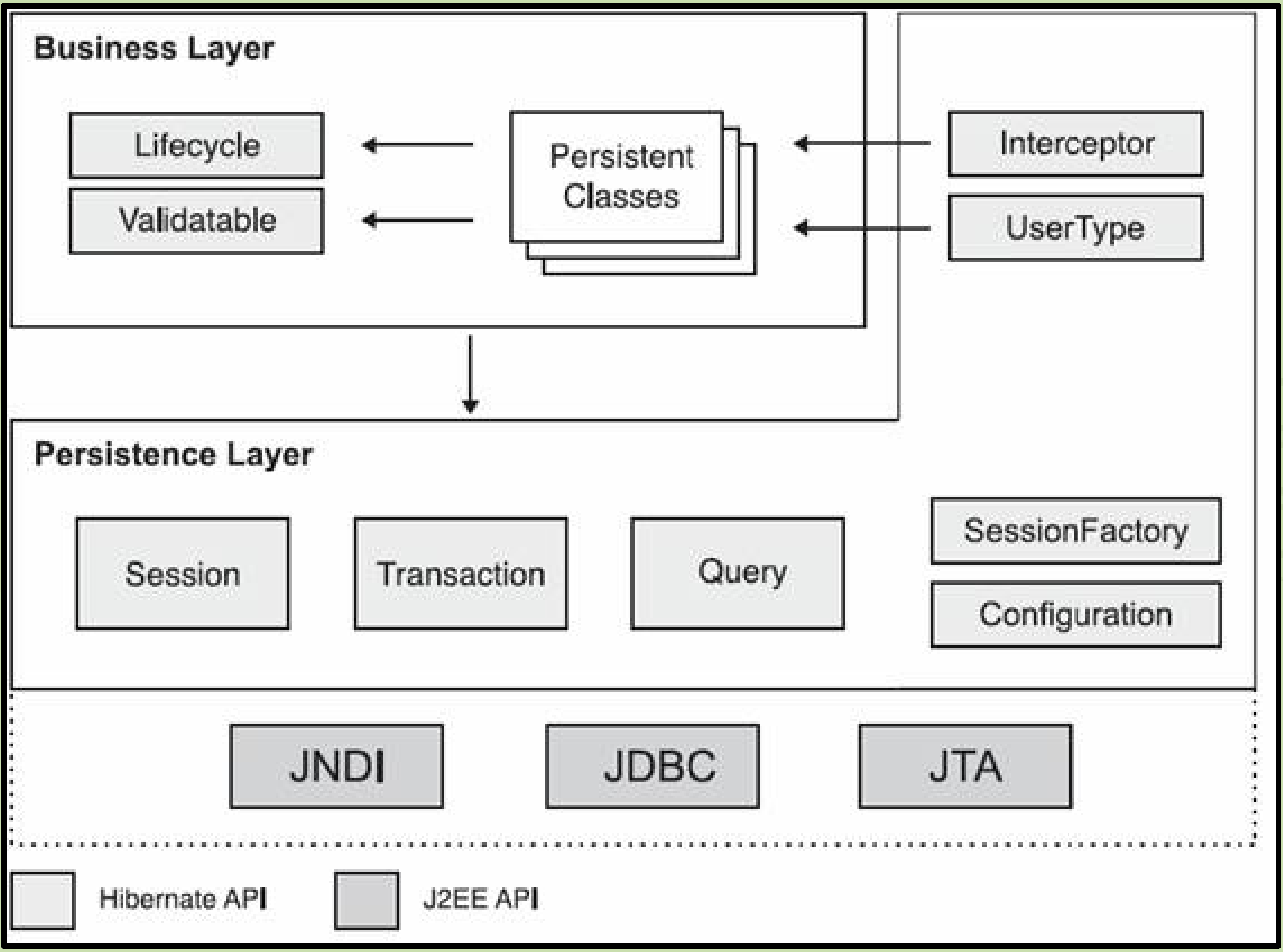
- log4j.properties file, at the root of the classpath:

```
# File Appender:
log4j.appender.tmpFile = org.apache.log4j.FileAppender
log4j.appender.tmpFile.File = /tmp/info.log
log4j.appender.tmpFile.layout =
                        org.apache.log4j.PatternLayout
log4j.appender.tmpFile.layout.ConversionPattern =
                                     [%d] %c %5p: %m%n

# Loggers:
log4j.rootLogger = warn, tmpFile
log4j.org.hibernate = info, tmpFile
```

```
$ tail -f /tmp/info.log
```

Let's try the examples!

Architecture



org.hibernate.Session

- Data access session;
- Main interface with the persistence system;
- Light object: low creation cost;
- Not threadsafe: should not be shared among threads;
- Obtained from the session factory;
- Contains a collection of objects associated to that unit of work.

org.hibernate.SessionFactory

- Used to obtain sessions;
- Heavy object: high cost of construction;
- Threadsafe: can be shared by all threads;
- There should be a factory for each database used by the application;
- Created from the configuration.

org.hibernate.cfg.Configuration

- Allows for the configuration of the framework;
- Creates the session factory.

org.hibernate.Transaction

- Abstracts the transaction mechanism used by JDBC in the background;
  - JDBC transactions, JTA transaction, CORBA, etc.;
- Helps to make the code more portable;
- Its use is optional:
  - Hibernate will assume begin and end of transactions for each operation as the default behaviour.

`org.hibernate.Query`

- Execution of queries:
  - In HQL (Hibernate Query Language);
  - In SQL;
  - Programatically.
- Used together with `org.hibernate.Criteria`.

Callback interfaces

- Receive notifications when some important event occurs (like listeners);
- Intrusive interfaces (implemented by the domain class):
  - `org.hibernate.classic.Lifecycle`;
  - `org.hibernate.classic.Validatable`.
- Non-intrusive interface:
  - `org.hibernate.Interceptor`.

Types

- Classes that map from OO types to database columns;
- Hibernate comes with several types ready to use:
  - `org.hibernate.type.Type.`
- The developer can create his/her own type:
  - `org.hibernate.usertype.UserType;`
  - `org.hibernate.usertype.CompositeUserType.`

Extension interfaces

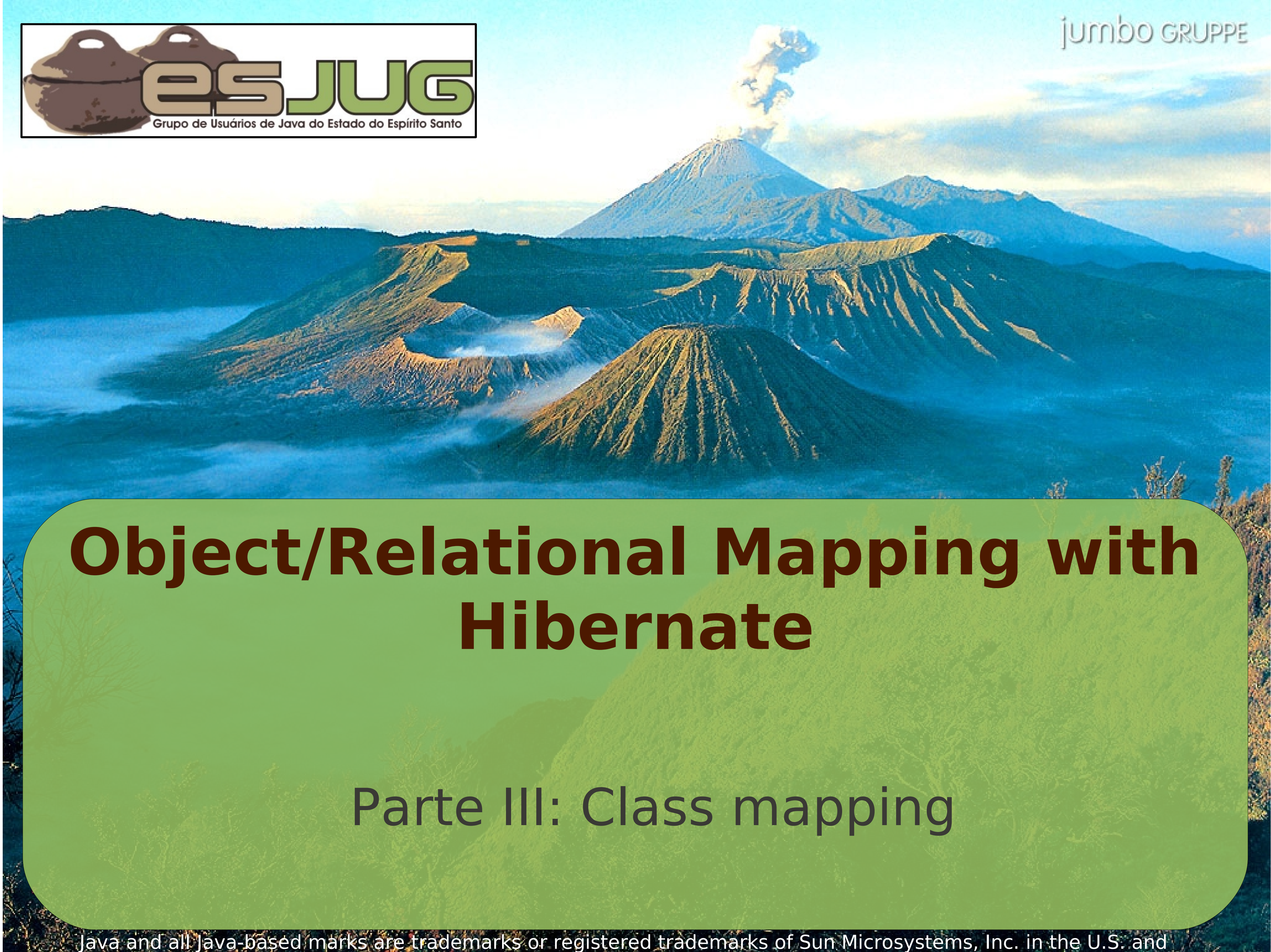
- Features that can be customized:
  - Primary-key generation;
  - SQL dialect support;
  - Caching strategies;
  - JDBC connection management;
  - Transaction management;
  - ORM strategy;
  - Property access strategy;
  - Proxy creation.

Hibernate features

- Persistent object management;
- Dirty checking: checks if persistent objects have changed and updates their data on the database;
- Transaction write-behind: only sends the SQL command when the transaction is commited;
- Flexible mapping, polymorphic queries;
- Two-level cache;
- Lazy initialization: associations and properties;
- Outer-join fetching;
- Etc., etc., etc.

Conclusions of part II

- Hibernate can be a good solution for persistence;
- We learned how to install and provide its required dependencies;
- We passed quickly through configuration options and basic usage;
- We also summarized its architecture and general features.



Object/Relational Mapping with Hibernate

Parte III: Class mapping

Which classes should I map?

- All classes that need to store their state in persistent media (database);
- Usually, these are domain classes;
- Domain model:
  - Constructed from the analysis of the domain of the problem to be solved;
  - Contains classes that represent concepts of the real world that are part of the business problem;
  - Let's analyze a little further some of its details...

Rules for domain classes

- They're API-independent:
  - We shouldn't have JDBC, Swing, Web or any external library code inside domain classes;
- They don't worry about cross-cutting concerns:
  - Persistence, transaction management, loggin, etc. are cross-cutting concerns;
  - It's not their goal to worry about these tasks;
  - We need transparent persistence.

Transparent persistence

- ≠ automated persistence (EJB);
- Total separation between domain classes and persistence logic;
- Doesn't require interface implementation or abstract class inheritance;
- The class can be reused in other contexts.

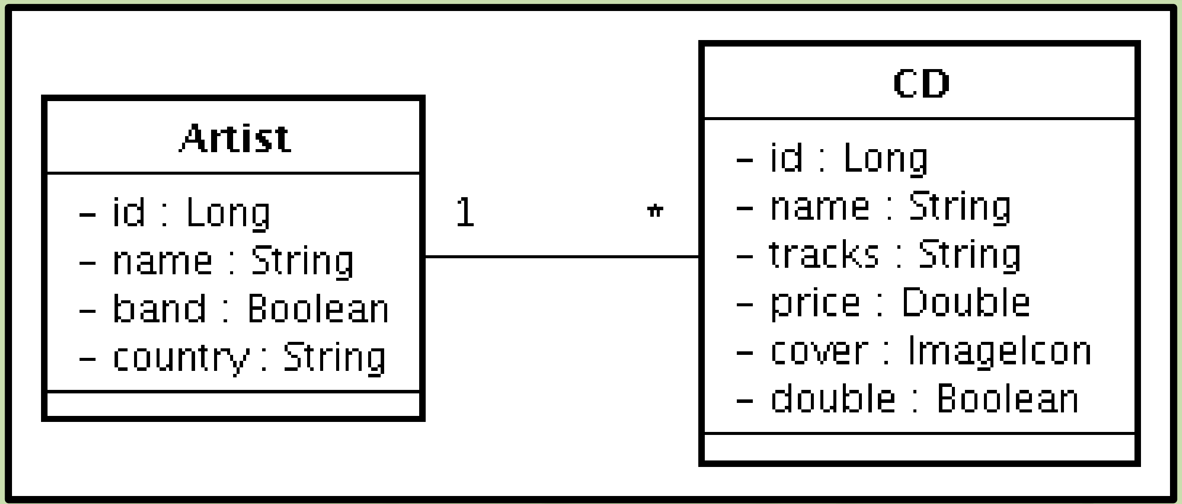
Hibernate works with POJOs

- Mandatory:
  - The class must have a default constructor (it can be private);
  - For collection attributes (e.g.: lists), use the interface (e.g.: List) and not a class (e.g.: ArrayList);
- Optional, but recommended:
  - All attributes should have get/set methods;
  - The class should have a specific id attribute.

A POJO

```java
package hibernatetutorial.domain;

public class Artist {
   private Long id;
   private String name;
   private Boolean band;
   private String country;

   private Set cds;

   /* Implicit construtor. */

   /* Properties getters and setters. */
}
```

Implementing associations



Using the interface
Hibernate provides
its own implementat

```
public class Artist {
   /* ... */

   private Set<CD> cds;

   public Set<CD> getCds() { return cds; }
   private void setCds(Set<CD> cds) {
      this.cds = cds;
   }
}
```

Setter is private.

Implementing associations

- Hibernate doesn't manage associations for you;
- Do your own convenience method:

```
public class Artist {
   /* ... */

   public void addCd(CD cd) {
      if (cd == null) throw new
                IllegalArgumentException("null CD");
      if (cd.getArtist() != null)
                cd.getArtist().getCds().remove(cd);
      cd.setArtist(this);
      cds.add(cd);
   }
}
```

Implementing associations

- Your convenience method should also guarantee the cardinalities of the association;
  - It's good practice to implement behaviour and check domain constraints.
- The `getCds()` method should not return a copy of the set:
  - It's a common encapsulation practice, so to avoid `artist.getCds().add(anyCd);`
  - However, this causes a confusion on Hibernate's dirty checking mechanism...

Mapping options

- XML;
  - Markup files associated with a DTD/Schema;
  - Criticized by many (metadata hell).
- XDoclet:
  - Tool that generates XML from JavaDoc annotations.
  - Good if Java < 5.
- Hibernate Annotations:
  - Uses Java annotations to do the mapping on the classes themselves;
  - Only if Java >= 5.

XML Mapping

- Let's review our prior example:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-
3.0.dtd">

<class name="hibernatetutorial.domain.Artist">
   <id name="id">
      <generator class="native" />
   </id>
   <property name="name" length="100" />
   <property name="band" />
   <property name="country" length="100" />
</class>
```

XML Mapping

- Header:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-
mapping-3.0.dtd">
```

  - Mandatory use;
  - Indicates the DTD to be used.

XML Mapping

- Class e identifier:

```
<hibernate-mapping>
   <class name="hibernatetutorial.domain.Artist"
                                  table="Artist">
      <id name="id" column="id">
         <generator class="native" />
      </id>

   </class>
</hibernate-mapping>
```

  - One class per file is recommended;
  - Indicates which property is the object identifier and how its value is generated (more on this later).

XML Mapping

- Class attributes:

```
<property name="country" column="country"
type="string" length="100" />
```

  - The use of sensible defaults are recommended;
  - We can indicate if an attribute can be null or not:

```
<property name="country" not-null="true" />
```

  - We can use the `<column />` tag to detail the configuration of the column:

```
<property name="country" length="100">
  <column name="country" />
</property>
```

XML Mapping

- Derived properties:

```
<property name="discountPrice" formula="price –
(0.1 * price)" />
```

  - Calculated by the DBMS (using SQL) during *runtime*;
  - Used only on SELECT operations.
- Property access strategy:

```
<property name="pais" access="field" />
```

  - Indicates direct access (without get/set);
  - Default value is `property` (using get/set);
  - It's possible to define your own `PropertyAccessor`.

XML Mapping

- Controlling INSERTs and UPDATEs:

```
<property name="readOnlyData" insert="false"
update="false" />
```

  - Indicates if a property should participate in sentences INSERT and UPDATE (default is `true`).

```
<class name="hibernatetutorial.domain.Artist"
 dynamic-insert="true" dynamic-update="true">
```

  - Dynamic insert: include only non-nulls on INSERT statements (default is `false`);
  - Dynamic update: include only attributes that have changed in UPDATE sentences (default is `false`);

XML Mapping

- Quoted SQL identifiers:

```
<property name="description"
          column="'Item Description'" />
```

  - Adds quotes around an SQL identifier (table name, column name, etc.);
  - In some databases, forces case-sensitivity;
  - Used commonly with legacy databases.
- Namespace:

```
<hibernate-mapping
            package="hibernatetutorial.domain">
   <class name="Artist" table="Artist">
```

XML Mapping

- Other possibilities:
  - NamingStrategy & SQL Schemas: determine naming standards for tables and columns;
  - Meta-data manipulation during runtime:

```
// Use before cfg.buildSessionFactory()
PersistentClass metaData;
metaData = cfg.getClassMapping(Artist.class);
```

Understanding object identity

- To understand Hibernate ids, first we need to understand object id;
- Two objects, A and B, can be:
  - Identical: (`A == B`) is `true`;
  - Equal: (`A.equals(B)`) is `true`;
  - Database-identical: they represent the same line, i.e., they are on the same table and have the same primary key value.

Hibernate IDs

```
package hibernatetutorial.domain;

public class Artist {
   private Long id;

   public Long getId() { return id; }

   public void setId(Long id) {
      this.id = id;
   }
}
```

- Should the get/set methods be public?
- If they're private, only Hibernate has access.

ID mapping

```
<id name="id" column="id">
   <generator class="native" />
</id>
```

- DB identity can now be verified with `A.getId().equals(B.getId());`
- A class can have its ID managed by Hibernate:

```
<id column="id">
   <generator class="native" />
</id>
```

- Obtain it using `session.getIdentifier(o);`
- Not recommended (performance loss).

Choosing an ID

- It's the same as choosing a primary key:
  - (a) Set of attributes that uniquely identifies an object (a person's tax code, a book's ISBN, etc.);
  - (b) Creation of a specific property as PK.
- Option A:
  - Called “natural key”;
  - Can cause maintenance problems.
- Option B:
  - Called “synthetic keys” or “surrogate keys”;
  - Recommended by Hibernate's authors.

Surrogate key generation

- Hibernate provides some generators, which are configured with `<generator class="" />`:

`increment`	Automatic increment for non-concurrent use.
`identity`	IDENTITY column for databases that support it.
`sequence`	SEQUENCE column for databases that support it.
`hilo`	Use of the high/low algorithm by Scott Ambler.
`native`	Chooses among `identity`, `sequence` e `hilo`, depending on SGDB support.
`uuid`	Generates an unique (inside a network) 32-char string.
`assigned`	Manually set by the developer before saving.

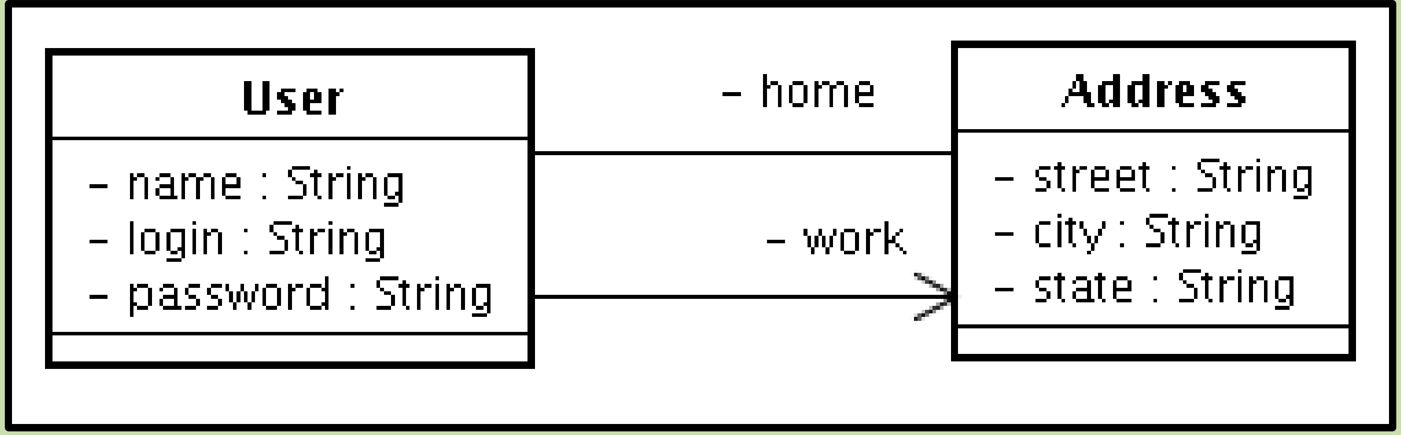
- You can make your own `IdentifierGenerator`.

Composite IDs

- Two or more properties compose the object's ID;
- Used more often with natural keys in legacy databases;
- Advanced use, not recommended for new projects.

Entities x value types

- Hibernate allows us to have more classes than database tables (granularity);
- A class that doesn't have a table is a value type;
  - It exists only associated to an entity;
  - Its properties are stores in the associated entity's table, it doesn't have an id and follows the life cycle of its owner.



Other examples:
String, Integer, Date, etc.

Components

- On HBMs, value types are called components;
  - Not to be confused with software components!

```
<class name="User">
  ...
  <component name="home" class="Address">
    <parent name="user" />
    <property name="street" />
    ...
  </component>

  <component name="work" class="Address">
    ...
  </component>
</class>
```

Components

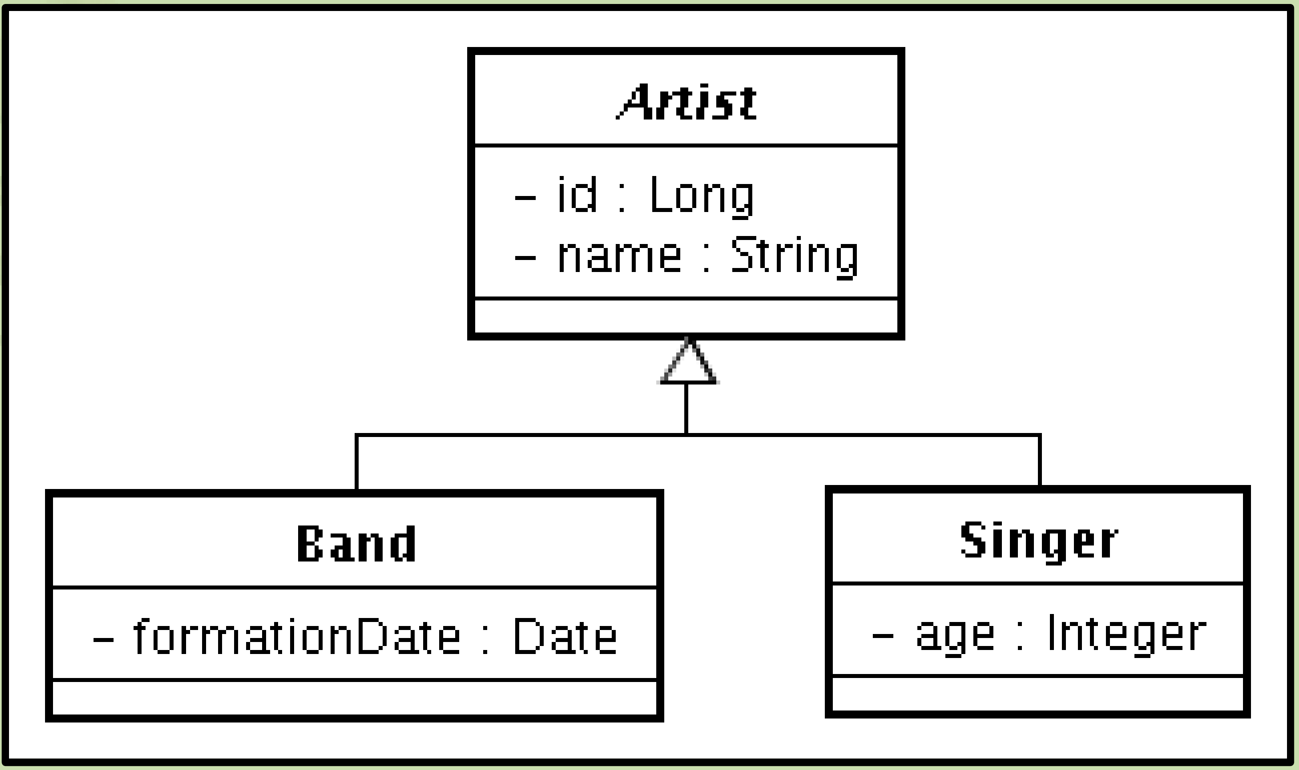
- A component can:
  - Have as many properties as you wish;
  - Have components of his own;
  - Be associated with other entities.
- Limitations:
  - Can't be shared (more than one owner);
  - There is no difference between a null component and a component with all properties null.

Inheritance mapping

- Inheritance distinguishes OO from Relational;
- A conversion is necessary;
- Scott Ambler proposed three possibilities:
  - A table for each class of the hierarchy;
  - A table for each concrete class of the hierarchy;
  - A single table for all the hierarchy.

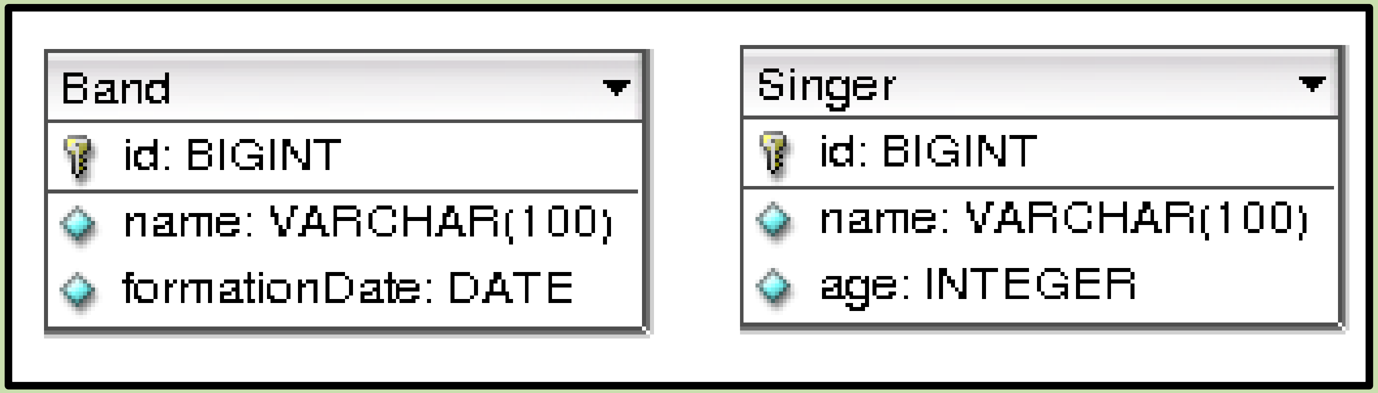
Inheritance mapping

- Example:
  - Artist is an abstract class;
  - Two concrete subclasses: Band and Singer.



A table for each concrete class

- Bad support for polymorphism (e.g.: associations in the superclass);
- Inefficient polymorphic queries (several SELECTs);
- Efficient concrete class queries;
- Column duplication is bad for maintenance;
- Use when polymorphism is not a requirement.

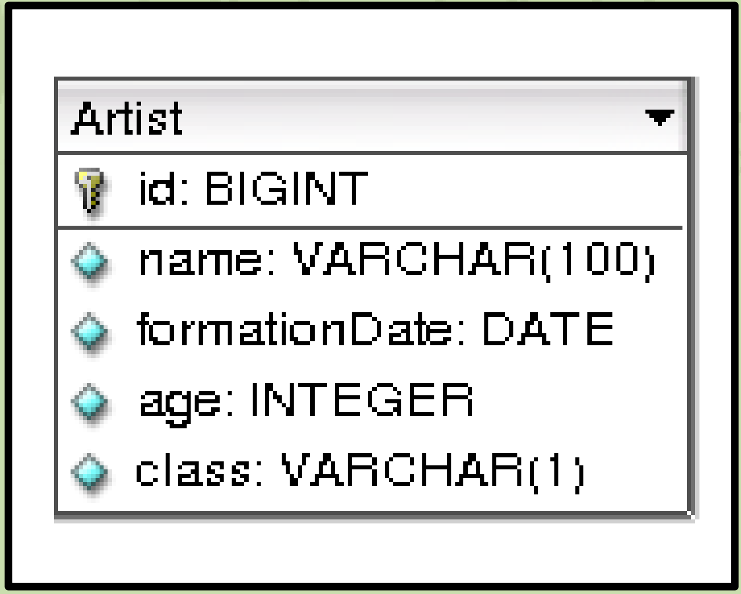


A table for each concrete class

```
<class name="Band">
   <!-- Property mappings. -->
</class>

<class name="Singer">
   <!-- Property mappings. -->
</class>
```

Single table

- Efficient polymorphic queries;
- No redundant columns;
- Columns that belong only to subclasses must be nullable (integrity constraint problem);
- Waste of space;
- Recommended for most cases.

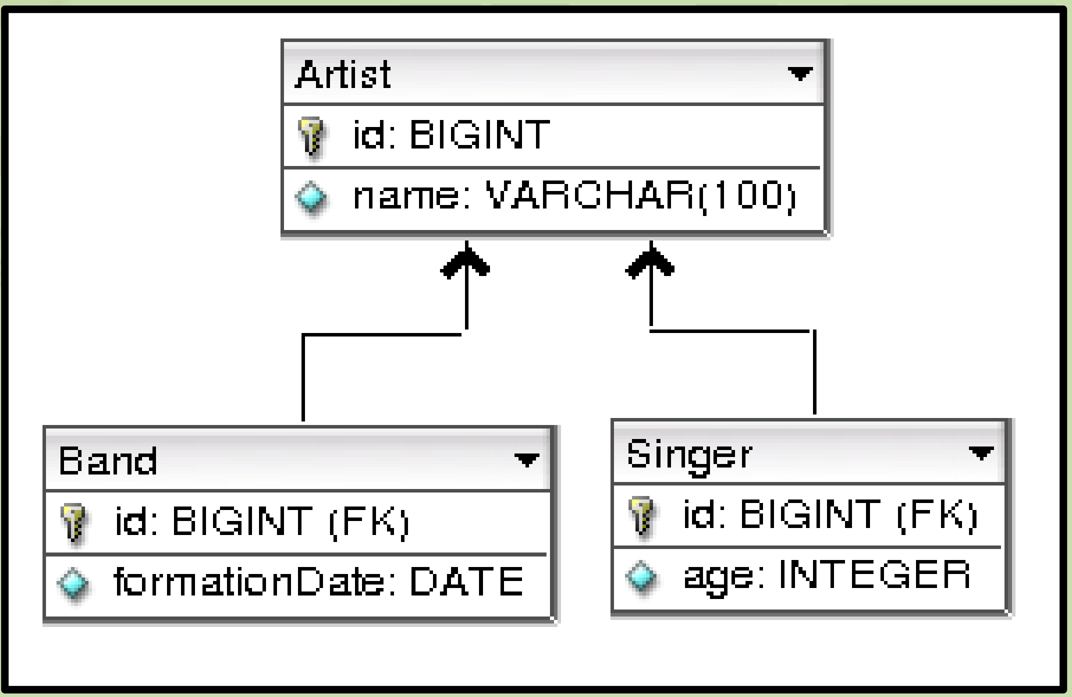


Single table

```
<class name="Artist" discriminator-value="a">
   <id name="id"> ... </id>

   <discriminator column="class" type="char" />

   <property name="name" />

   <subclass name="Band" discriminator-value="b">
      ...
   </subclass>

   <subclass name="Singer" discriminator-
value="s">
      ...
   </subclass>
</class>
```

A table for each class

- No problems with integrity constrains or ambiguity;
- Bad performance because of JOIN operations;
- Recommended when integrity is a strong requirement.



A table for each class

```
<class name="Artist">
  <id name="id"> ... </id>

  <property name="name" />

  <joined-subclass name="Band">
    <key column="id" />
    ...
  </subclass>

  <joined-subclass name="Singer">
    <key column="id" />
    ...
  </subclass>
</class>
```

Inheritance mapping

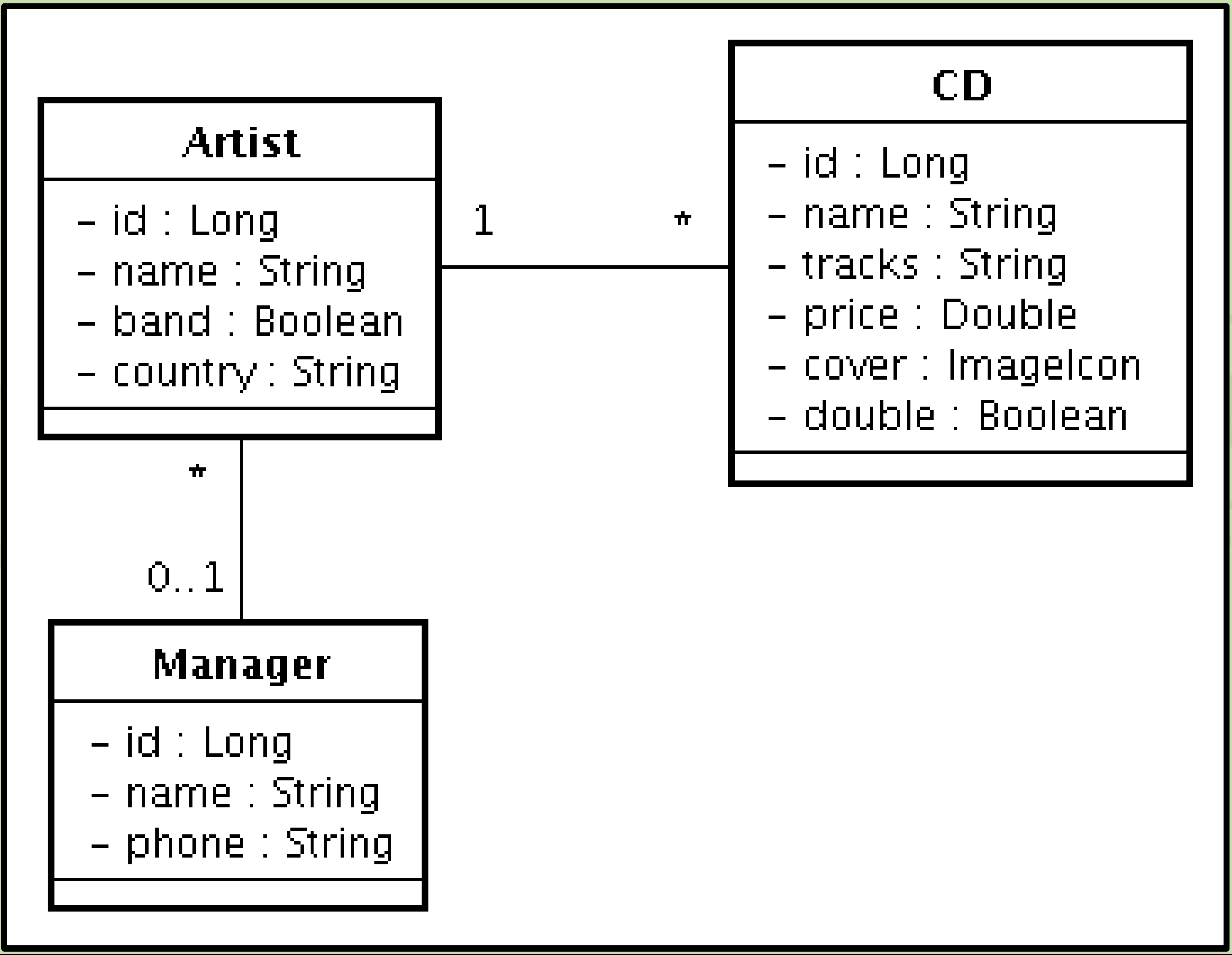
- Inheritance mapping strategies can't be combined;
- `<subclass />` e <joined-subclass /> can be declared:
  - Inside the super `<class />` (as the examples);
  - In a separate mapping file (must specify `<subclass name="..." ` **`extends="..."`** `/>`).

Association mapping

- The most complex part of the mapping;
- We will see only the simplest cases;
- Hibernate associations aren't managed:
  - EJBs with CMP have container-managed associations. Hibernate works with POJOs!
- By default, associations are unidirectional.

Collections

- Association cardinality:



Many-to-one mappings

```
<class name="CD">
   ...
   <many-to-one name="artist"
     class="hibernatetutorial.domain.Artist"
     not-null="true"
     cascade="none" />
</class>

<class name="Artist">
   ...
   <many-to-one name="manager"
     class="hibernatetutorial.domain.Manager"
     not-null="false"
     cascade="delete-orphan" />
</class>
```

Transitive persistence

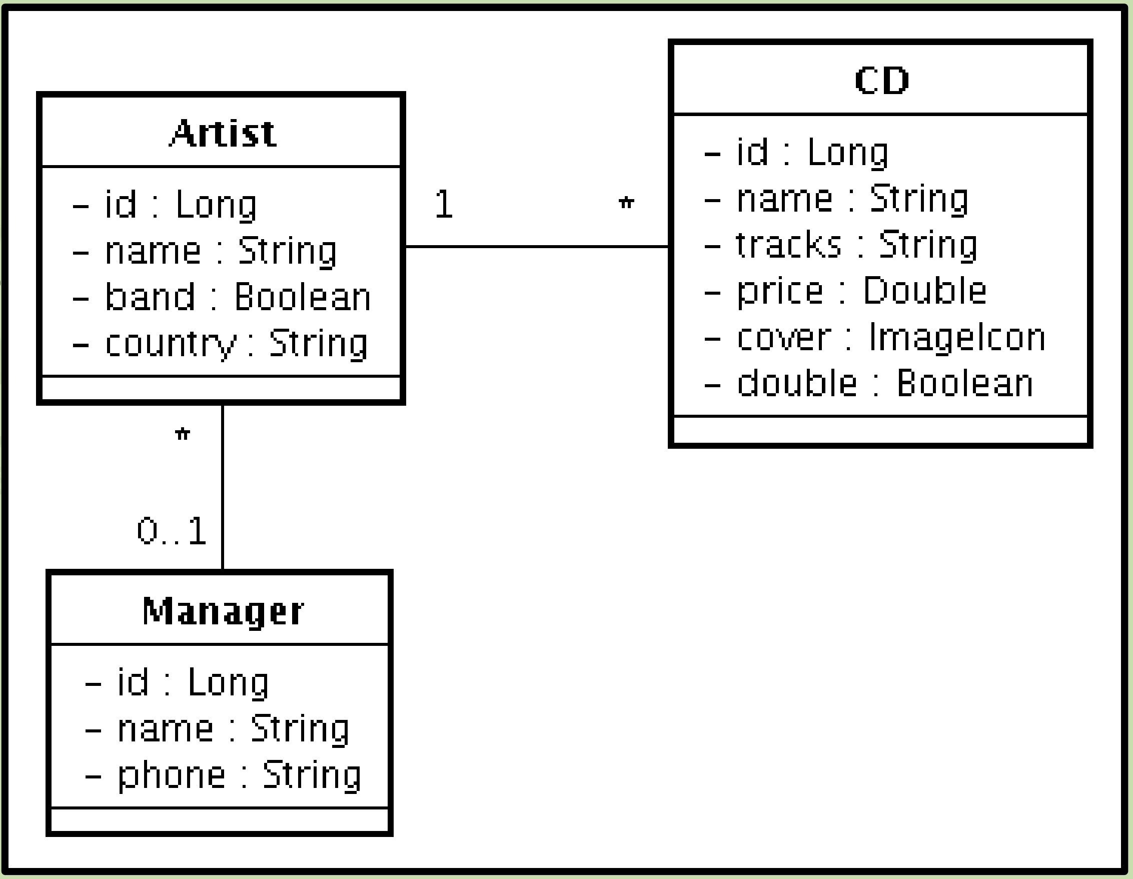
- Hibernate applies persistence by transitivity;
  - If an object X is persistent and an object Y is associated to it, Y must become persistent also.
- Configurable by the parameter `cascade="..."`:
  - `save-update`: if X is saved, Y will also be;
  - `delete`: if X is deleted, Y will also be;
  - `refresh`: if X is refreshed (data retrieved from database and refreshed in memory), Y will also be;
  - `delete-orphan`: if an Y has no associated X anymore, it will be deleted;
  - `all`: all cascades combined.

One-to-many mapping

```
<class name="Artist">
   ...
   <set name="cds" inverse="true" cascade="all">
      <key column="artistId" />
      <one-to-many class="[...].CD" />
   </set>
</class>

<class name="Manager">
   ...
   <list name="artists" lazy="false" inverse="true">
      <key column="managerId" />
      <list-index column="order" />
      <one-to-many class="[...].Artist" />
   </set>
</class>
```

Exercise

- Write a single program that registers CDs, based on our example domain model;
- Choose the simplest GUI available;
- Check the database for the modifications made by Hibernate.



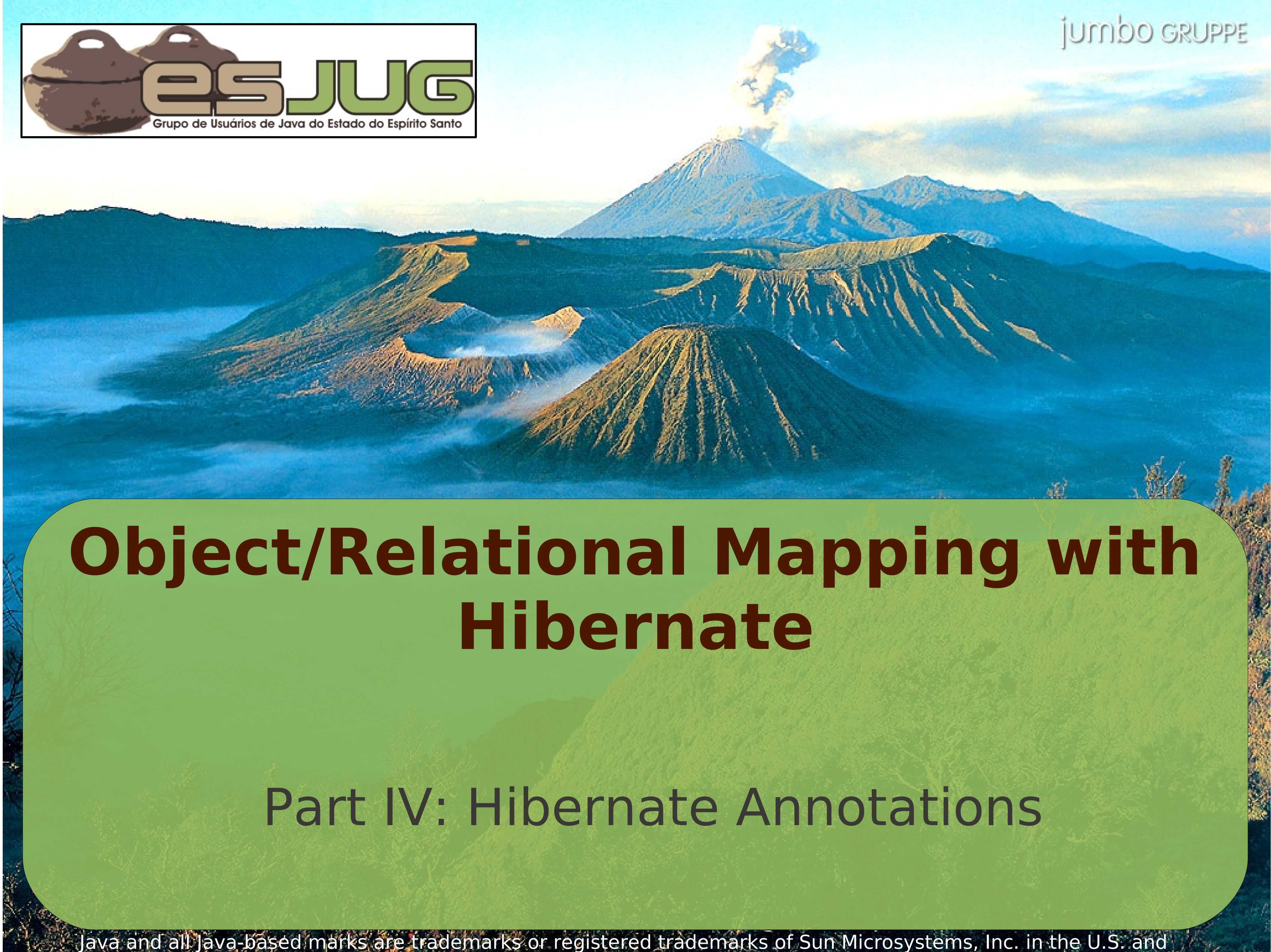
Conclusions of part III

- Complex systems have a domain layer, representing concepts of the business problem;
- An interesting approach to data management is the automatic persistence of POJOs;
- We learned how to map simple classes, hierarchies and associations using XML files;
- We discussed object identity and saw how to generate surrogate ids and how to map value types.

Grupo de Usuários de Java do Estado do Espírito Santo

Object/Relational Mapping with Hibernate

Part IV: Hibernate Annotations



Annotations

- Meta-data added to source code to describe it;
- This meta-data is then available during runtime;
- It is seen as the main alternative to XML configuration;
- In Java, it's available since version 5.

Hibernate Annotations

- Use of annotations to do O/R mapping;
- Replaces the HBM XML files;
- Follows the standard for EJB 3 persistence;
- Libraries required:
  - `hibernate-annotations;`
  - `ejb3-persistence.`

Replacing XML with annotations

1) Annotate your domain class (you can delete the `.hbm.xml` file after this step);

2) Change `hibernate.cfg.xml` to load the classes themselves instead of the XML files;

3) Obtain a `SessionFactory` by means of an `AnnotationConfiguration`.

Changes on hibernate.cfg.xml

- Using XML:

```
<mapping
resource="hibernatetutorial/domain/Artist.hbm.xml" /
>
```

- Using Hibernate Annotations:

```
<mapping class="hibernatetutorial.domain.Artist" />
```

Obtaining the SessionFactory

- Using XML:

```
Configuration cfg = new Configuration();
cfg.configure();
sessionFactory = cfg.buildSessionFactory();
```

- Using Hibernate Annotations:

```
Configuration cfg = new AnnotationConfiguration();
cfg.configure();
sessionFactory = cfg.buildSessionFactory();
```

Annotating the classes

- Annotations are defined in `javax.persistence`;
- We place them in specific points of the code:
  - Before a class definition;
  - Before a property definition;
  - Before a getter method definition.

```
@Entity
public class Artist {
   @Id @GeneratedValue(strategy = GenerationType.AUTO)
   public Long getId() { return id; }

   @Column(length = 100)
   public String getName() { return name; }

   /* ... */
}
```

Entity Beans

- A persistent class is an Entity Bean;
- The name was inherited from EJB technology;
- Use the `@Entity` annotation before the definition of the class:

```
@Entity
public class Artist {

    /* ... */

}
```

The entity's id

- After defining an entity, we should define its id;
- The annotations @Id and @GeneratedValue can be used before the id property or its getter method:
  - @GeneratedValue allows us to define a generation srategy (AUTO, IDENTITY, SEQUENCE, TABLE) or class:

```
@Id @GeneratedValue(strategy = GenerationType.AUTO)
public Long getId() { return id; }
```

Annotating property x getter method

- If you annotate the property, Hibernate will use the `field` access strategy;
- If you annotate the getter method, Hibernate will use the `property` access strategy;
- It's recommended not to mix the strategies.

Simple properties

- There are many annotations for simple properties:
  - @Transient: not stored in the database;
  - @Basic: numbers, booleans, Strings;
  - @Temporal: date and time;
  - @Lob: large textual or binary object.
- Attention: @Basic is the default!

```
@Transient
public String getValue() { return value; }

@Basic
public Double getSalary() { return salary; }

@Temporal(TemporalType.DATE)
public Date getBirthdate() { return birthdate; }
```

Column attributes

- We can specify the characteristics of the column where the property will be stored;
- We use the @Column annotation:

```
@Basic
@Column(length = 50, nullable = false)
public String getName() { return name; }
```

Changing the default values

- If you don't want to use the default values, you can specify parameters on annotations:
  - Specifying the table:

```
@Entity
@Table(name = "ARTISTS")
public class Artist { }
```

  - Specifying the column:

```
@Basic
@Column(name = "ARTIST_NAME")
public String getName { return name; }
```

Inheritance mapping

- A table for each concrete class:

```
@Entity
public abstract class Artist { }

@Entity
@Inheritance(
   strategy = InheritanceType.TABLE_PER_CLASS
)
public class Band extends Artist { }
```

Inheritance mapping

- Single table for all the hierarchy:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
   name = "class",
   discriminatorType = DiscriminatorType.CHAR
)
@DiscriminatorValue("A")
public class Artist { }

@Entity
@DiscriminatorValue("B")
public class Band extends Artist { }
```

Inheritance mapping

- A table for each class (joined-subclasses):

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Artist { }

@Entity
public class Band extends Artist { }
```

Mapped superclasses

- They are not entities:
  - Can't be stored, retrieved or used in any queries;
- However, they define persistent properties that can be inherited by subclasses;
- It's a good way to build utility classes.

```
@MappedSuperclass
public class Artist { }

@Entity
public class Band extends Artist { }
```

Associations

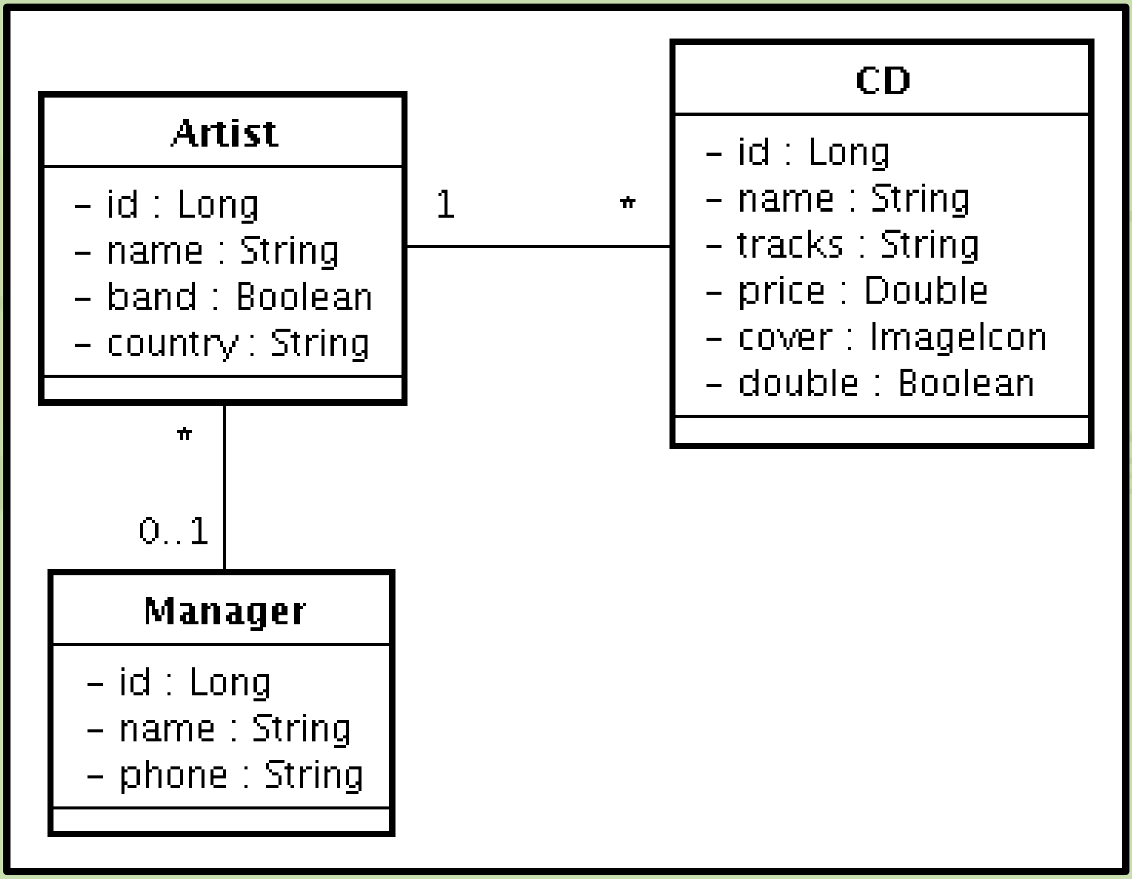
- There are four kinds:
  - `@OneToOne;`
  - `@OneToMany;`
  - `@ManyToOne;`
  - `@ManyToMany.`
- Most commonly used properties:
  - `cascade = CascadeType.___;`
  - `mappedBy = "___";`
  - `fetch = FetchType.___;`
  - `@JoinColumn(nullable="true|false").`

Associations – examples:

```
public class CD {
   @ManyToOne
   public Artist getArtist() { return artist; }

   @ManyToOne
   public Label getLabel() { return label; }
}

public class Artist {
   @OneToMany(
      cascade = CascadeType.ALL,
      mappedBy="artist"
   )
   public Set<CD> getCds() { return cds; }
}
```

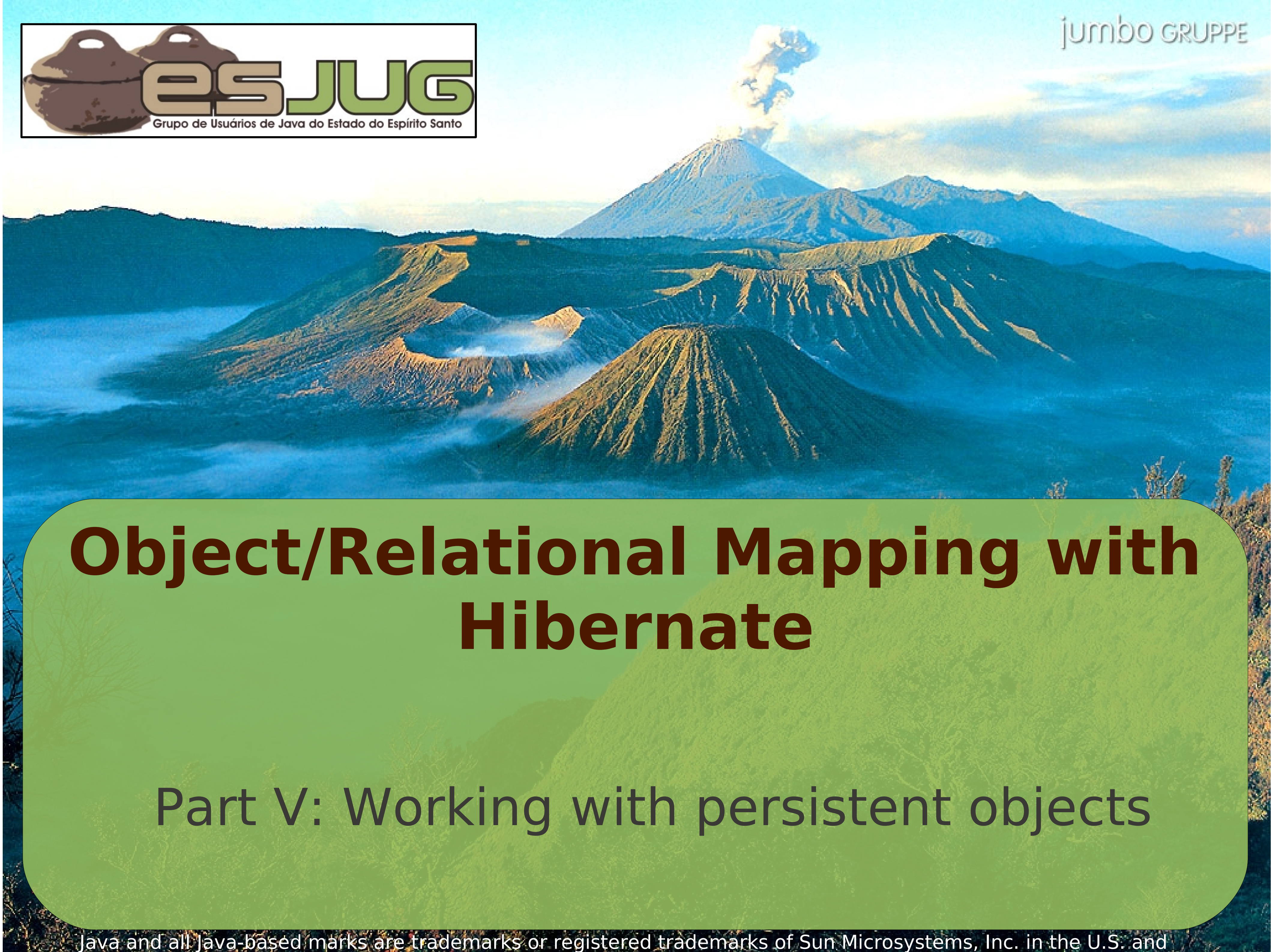
Exercise

- Change the XML mapping from the last exercise to annotations.



Conclusions of part IV

- Annotations are meta-data that are added to Java classes, available only since version 5;
- Hibernate Annotations allows us to replace the HBM XML files with annotations using the EJB3 standard;
- We saw how to do the simple mappings that we learned on part III using EJB3 standard annotations;
- We also saw the changes we have to make in Hibernate's configuration to use annotations.



jumbo GRUPPE

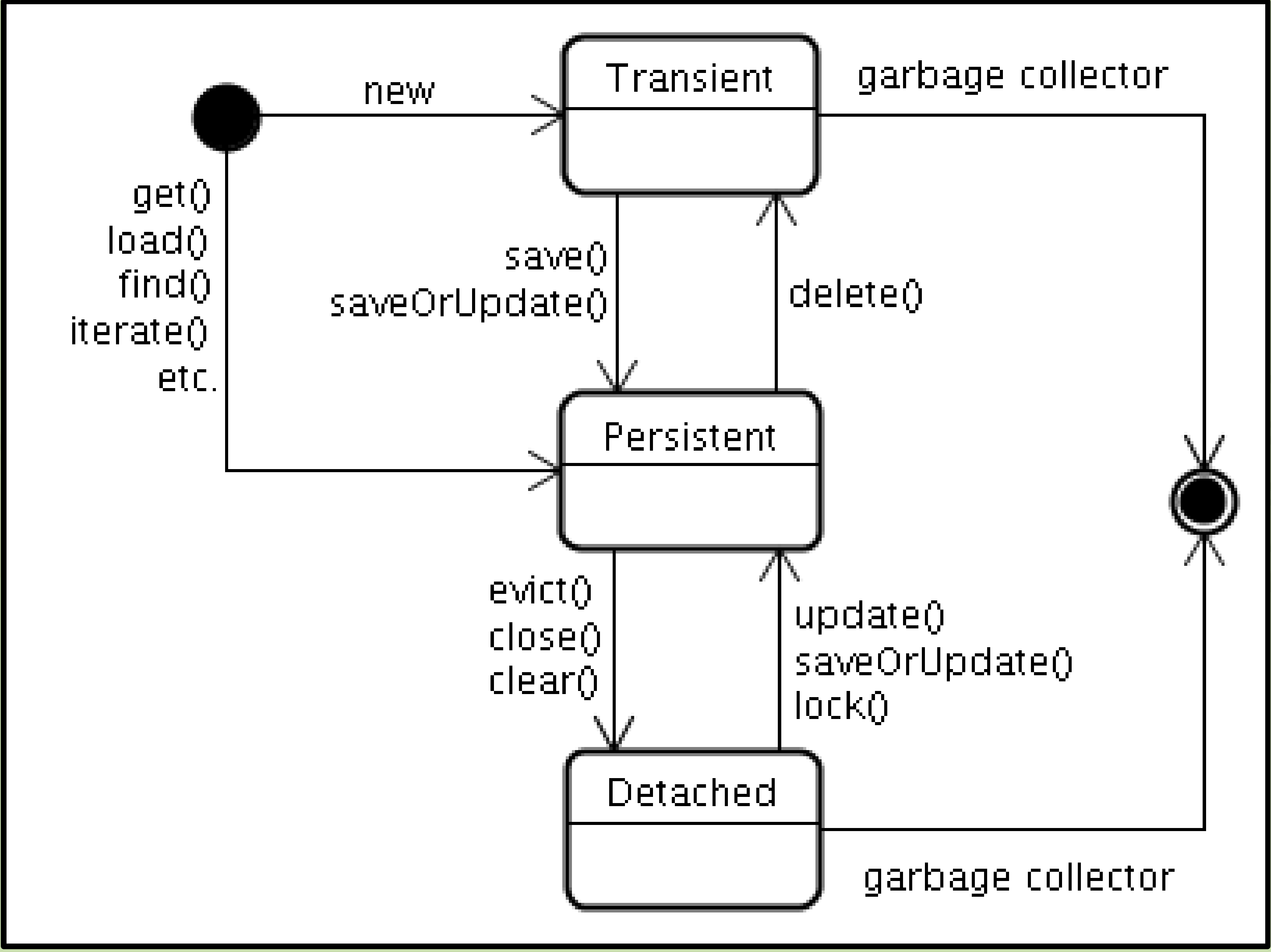
Object/Relational Mapping with Hibernate

Part V: Working with persistent objects

The interface with the persistence

- Hibernate is not a container (it uses POJOs);
  - Your application can work with objects independently of knowing if they are persistent or not.
- To carry persistence tasks, we use:
  - The persistence manager (retrieving by id, saving and excluding objects);
  - Query interfaces (retrieving with queries).
- Before we go deeper into them, we need to know the objects' life cycle and scope.

Object life cycle with Hibernate



Transient objects

- Objects that haven't been persisted by Hibernate: normal Java objects, unloaded with GC;
- Can't have association with persistent objects (transitive persistence);
- Don't have any relationship with Hibernate.

Persistent objects

- Objects that have database identity;
- They were saved to or retrieved from the database by Hibernate (directly or by transitivity);
- They are always associated with a `Session` object.

Persistent objects are transactional

- To be associated with a *session* means participating in a transaction;
- It means its state is updated at the end of the transaction (or during syncs);
- Only new objects or objects that have been changed are updated (dirty checking);
- If `dynamic-update = true`, Hibernate updates only the properties that have been changed;
- The application may not know about any of it (transparent transaction-level write-behind).

Detached objects

- When a session closes, its objects become detached;
- They are not longer managed by Hibernate:
  - There is no further guarantee that their state is in sync with the database;
- They can still be used if reattached with another session;
  - Careful, because its data could be stale!
- We can detach an object from a session with the `evict()` method.

The scope of object identity

- An ORM solution can have as scope:
  - None: there are no guarantees that the same object is returned if the same query is made twice;
  - Transaction: the guarantee exists within a transaction;
  - Process: it exists within the entire JVM (high cost).
- Hibernate has session scope:
  - If A and B are objects from the same class, are retrieved by the same `Session` object and `A.getId().equals(B.getId())`, then `A == B`;
  - This is called “first level cache”.

Selective reassociation

- In distributed applications (e.g.: Web), a session can't be open all the time:
  - It's usual to obtain an object in a session and then save it in another session;
  - The object (and all its associated graph of objects) must be reassociated with a second session.
- Hibernate performs selective reassociation:
  - Only objects that interest are reassociated;
  - This is made in an automatic, efficient way.

The impact of object equivalence

- `equals()` and `hashCode()` are used in many situations (e.g.: the Collections API);
- The default implementation (`Object` class) uses memory equality (`a == b`);
- Hibernate does not guarantee this equality in different sessions.

We have a problem...

The impact of object equivalence

- Possible solutions:
  - Using database identity (PK): problem with transient objects;
  - Compare property values: it's difficult to find immutable values that uniquely define an object (ex.: tax code, ISBN, etc.);
  - Using an UUID: generate a synthetic identifier (there aren't two objects with the same UUID in the same JVM).

The impact of object equivalence

- UUID implementation:

```
@MappedSuperclass
public abstract class PersistentObject {
   private String uuid;
   private Long id;

   public PersistentObject() {
      uuid = java.util.UUID.randomUUID().toString();
   }

   @Column(nullable = false, length = 40)
   public String getUuid() { return uuid; }
   public void setUuid(String u) { this.uuid = u; }

   @Id @GeneratedValue(strategy = GenerationType.AUTO)
   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }
```

The impact of object equivalence

- Implementing equals() and hashCode():

```
    public boolean equals(Object obj) {
        // Checks if they belong to the same class.
        if (! getClass().equals(obj.getClass()))
            return false;
        PersistentObject o = (PersistentObject)obj;

        // Compare UUIDs.
        return uuid.equals(o.uuid);
    }

    public int hashCode() {
        return uuid.hashCode();
    }
}
```

The impact of object equivalence

- Can the UUID be used as the primary key?
  - Yes, it is a String id with assigned generation strategy.
- What's the impact of this on DB performance?
  - Ask a DBA!
  - Joins and indexes are made using the primary key;
  - Something tells me numeric PKs perform better...

The persistence manager

- An object that provides:
  - Basic CRUD operations;
  - Query execution;
  - Transaction management;
  - Cache management.
- In Hibernate, it's composed by the objects: `Session`, `Query`, `Criteria` and `Transaction`;
- A session (light object) is obtained in the session factory (heavy object). Each thread should obtain its own session.

Making an object persistent

- Four steps:
  - Create the object;
  - Obtain a Session object;
  - Call the method `.save(obj)` (the SQL command will be sent to the database in the proper time);
  - Close the session.

```
Artist artist = new Artist(); /* ... */
Session session = HibernateUtil.openSession();
Transaction tx = session.beginTransaction();
session.save(artist);
tx.commit();
session.close();
```

Updating an object

- Persistent objects are updated automatically until the end of the sentence;
- Detached objects need to be reassociated:
  - `update(obj)`: forces the update (SQL command);
  - `lock(obj)`: update is done only if necessary.

```
// artist was obtained in another session.
Session session = HibernateUtil.openSession();
Transaction tx = session.beginTransaction();
session.update(artist); // session.lock(artist)
tx.commit();
session.close();
```

Saving an object

- Hibernate can distinguish between transient and detached objects;
- The method `saveOrUpdate(obj)` inserts or updates and object in the database;
- The decision is made checking the id (if it's null);
- For this reason, the null value must be specified for ids that are of primitive types:

```
<id name="id" type="long" unsaved-value="0">
   <generator class="identity"/>
</id>
```

More on transitive persistence

- Operation cascading:

`none`	Does not cascade anything (default).
`save-update`	Applies transitivity on saving (insert/ update).
`delete`	Applies transitivity on deletion.
`all`	Equivalent to `save-update` + `delete`.
`delete-orphan`	Deletes objects that are no longer participating in the association (orphans).
`all-delete-orphan`	Equivalent to `all` + `delete-orphan`.

Deleting an object

- To delete an object is to make it transient (it stays in memory until being garbage collected);
- Use the method `delete(obj)`;
- Can be used on persistent or detached objects (detached objects are automatically reassociated).

```
// artist was obtained in this or another session
Session session = HibernateUtil.openSession();
Transaction tx = session.beginTransaction();
session.delete(artist);
tx.commit();
session.close();
```

Exception handling

- Many of these methods can throw a `HibernateException`;
    - The exception handling code is not shown for brevity, but it's supposed to be carried out.
- Usually, DBMS errors are not recoverable (i.e.: get another connection);
- The recommendation is to discard the session and obtain a new one when a DB error occurs;
- Hibernate does the rollback of the transaction in the database, but not in memory!

Retrieving objects

- Several ways to do it:
  - Navigating by the object graph starting from an object that has been retrieved before (the session has to be open);
  - Obtaining the object using its ID (primary key);
  - Using Hibernate Query Language (HQL);
  - Using the Criteria API;
  - Query by example;
  - Using SQL.
- Hibernate caches retrieved objects, avoiding unnecessary hits to the DB.

Retrieving objects

- Retrieving using the object's ID:
  - `get(class, id)`: returns null if the object doesn't exist on the DB and doesn't use proxies;
  - `load(class, id)`: throws an exception if the object doesn't exist on the DB and can use proxy.

```
Session session = HibernateUtil.openSession();
Transaction tx = session.beginTransaction();
Artist artist = (Artist)session.get(
                      Artist.class, new Long(1));
tx.commit();
session.close();
```

Retrieving objects

- Hibernate Query Language:
  - SQL-like, object oriented query language;
  - Reminds OQL and EJB-QL 2.1;
  - Used as basis to EJB-QL 3.0;
  - Unlike SQL, it is used only for object retrieval.

```
// Session open/close not shown.
Query query = session.createQuery("from Artist a
where a.country = :country");

query.setString("country", "Italy");
List result = query.list(); // List of Artists
```

Retrieving objects

- HQL examples:

```
-- Retrieves all artists:
from Artist;

-- Using criteria:
from Artist a where a.name like 'C%'

-- Using joins:
from Artist a inner join a.cds as cds

-- Navigating the object graph on criteria:
from Artist a where a.manager.nome = 'John Doe'

-- Sorting:
from Artist a order by a.name
```

Retrieving objects

- HQL features:
  - Object oriented syntax;
  - Navigation of the object graph;
  - Polymorphic queries;
  - Retrieval of only some attributes of the objects (instead of the entire entity);
  - Sorting and pagination of results;
  - Aggregation functions, group-by and having;
  - Joins and sub-queries;
  - Call native functions and stored procedures.

Retrieving objects

- Using the Criteria API:
  - Use of objects instead of strings;
  - More OO, less readable, more extensible.

```
// Session open/close not shown.
Criteria criteria =
session.createCriteria(Artist.class);

criteria.add(Expression.eq("country", "Italy"));
List list = criteria.list(); // List of Artists
```

Retrieving objects

- Query by example:
  - Creation of a criterion based on an semi-filled domain object;
  - Good for searches with many options.

```
// Session open/close not shown.
Artist example = new Artist();
exemple.setCountry("Italy");
Criteria criteria =
session.createCriteria(Artist.class);

criteria.add(Example.create(example));
List list = criteria.list();
```

Object fetching strategy

- When we use JDBC, we know exactly when an object is retrieved from the database;
- One of the main challenges of an ORM solution is to determine this moment;
  - Retrieve all objects of the graph at once?
  - Retrieve a little bit at a time, on demand?
- This decision has great impact on performance and code readability.

Object fetching strategy

- Hibernate allows us to specify the fetching strategy on the mapping and also change it during runtime:
  - Immediate fetching: sequential database reads;
  - Lazy fetching: on demant;
  - Eager fetching: use of outer joins;
  - Batch fetching: many objects at once.

Object fetching strategy

- The mapping depends on the cardinality (one or many);
- For simple associations (cardinality 1):
  - Use of proxies.

```
<!-- Completely disables lazy loading. -->
<class name="Artist" lazy="false" />

<!-- Defines the interface that the proxy will
implement (can be the class itself). -->
<class name="Artist" proxy="ArtistProxy" />
```

Object fetching strategy

```
<!-- Use of a defined proxy. -->
<many-to-one name="artist" lazy="proxy" />
<!-- Use of bytecode instrumentation. -->
<many-to-one name="artist" lazy="no-proxy" />
<!-- Forces eager fetching. -->
<many-to-one name="artist" lazy="false" />

<!-- Lazy if it has a proxy, eager if not. -->
<many-to-one name="artista" outer-join="auto" />
<!-- Forces eager fetching. -->
<many-to-one name="artista" outer-join="true" />
<!-- Forces immediate fetching. -->
<many-to-one name="artista" outer-join="false" />

<!-- Batch fetching. -->
<many-to-one ... lazy="true" batch-size="9" />
```

Object fetching strategy

- For collections (cardinality “many”):
  - Collections always have proxies;
  - Use of lazy and fetch on the mapping.

```
<!-- Lazy fetching (default and recommended). -->
<set ... lazy="true" />

<!-- Immediate fetching. -->
<set ... lazy="false" fetch="select" />

<!-- Eager fetching. -->
<set ... lazy="false" fetch="join" />

<!-- Batch fetching. -->
<set ... lazy="true" batch-size="9" />
```

Object fetching strategy

- Eager fetching uses outer joins;
- We can define its depth:
  - Indicates how many table joins will be performed in a single SELECT command;
  - Default is 1, it's not recommended to use a number greater than 4;
  - Global configuration on `hibernate.cfg.xml`:

```
<!-- Depth of eager fetching. -->
<property name="max_fetch_depth">3</property>
```

Object fetching strategy

- About lazy fetching:
  - Objects and collections are initialized when used for the first time;
  - At the moment of the object fetching, it must be associated to an open session;
  - Can be manually initialized:

```
Hibernate.initialize(artist.getCds());

// We can also check if it is initialized:
if (Hibernate.isInitialized(artist.getCds())) {
   // ...
}
```

Object fetching strategy

- These configurations are complex and can influence on the final result of your application;
- If needed, ask for specialized help of a DBA to understand performance impact;
- Monitor your application and make tests with various configurations to determine the best one.

Conclusions of part V

- Objects managed by Hibernate have well defined scope and life cycle;
- This scope leads to a equivalence problem, with many possible solutions;
- With that understood, we learned about the persistence manager:
  - Basic CRUD;
  - Many ways to retrieve an object.
- The fetching strategy can also be configured, which is as difficult as important.

Conclusions of the basic tutorial

- In this basic tutorial, we have seen:
  - What is object/relational mapping (ORM);
  - How to install and configure Hibernate;
  - Hibernate's internal architecture;
  - Tips on how to write a domain layer;
  - How to map domain classes to tables using XML and annotations;
  - How to save, delete and retrieve objects using Hibernate's persistence manager.