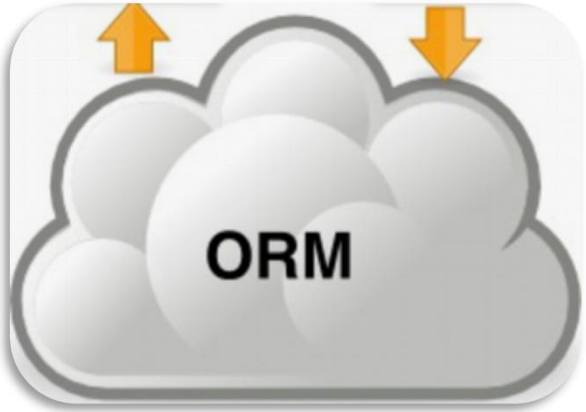
Object-Relational Mapping (ORM)



Course Outline

- **Introduction to ORM**
  - What is JDBC?
  - Why ORM?
  - What is ORM?
  - Java ORM/Persistent Frameworks
- **Hibernate Overview**
  - What is Hibernate?
  - Why Hibernate?
  - Supporting Database Management Systems Engine
- **Hibernate Architecture**
  - Hibernate Architecture and API

Course Outline (Ex.)

- **Hibernate Configuration**
  - Hibernate Installation/Setup
  - Configuration Properties
  - Mapping File(s)
  - Development Strategies
- **First Example (Hello Hibernate)**
- **Hibernate Entity Life-Cycle**
  - Entity Life-Cycle (Object States)
  - Session Operations

Course Outline (Ex.)

- **Entities Associations**
  - Many-to-one (Uni-directional and Bi-directional)
  - One-to-one (Uni-directional and Bi-directional)
  - Many-to-many (Uni-directional and Bi-directional)
  - Inheritance Mapping Strategies
  - Table per concrete classes
  - Table per subclasses
  - Shared Table per concrete class with unions
  - Table per joined subclasses

Course Outline (Ex.)

- **Transitive Persistence**
  - Lazy and Eager loading
  - Fetching Strategies
    - Join Fetching Strategy
    - Select Fetching Strategy
    - Sub-Select Fetching Strategy
    - Batch Size Fetching Strategy
  - The Second Level Cache
    - Read-Only Cache
    - Read-Write Cache
    - Nonstrict-Read-Write Cache

Course Outline (Ex.)

- **Hibernate Query (by HQL)**
  - Hibernate Query
  - HQL Parameter Binding
  - HQL Restrictions
  - HQL Comparison Expressions
  - Standard HQL Functions
  - Added HQL Functions
  - HQL Ordering
  - HQL Projections
  - HQL Join(s)
  - HQL Grouping & Group Restrictions
  - HQL Dynamic Instantiation
  - HQL Sub-Queries

Course Outline (Ex.)

- **Hibernate Query (by Criteria)**
  - Parameter Binding
  - Restrictions
  - Comparison Expressions
  - String Matching
  - Logical Operators
  - Sub-Queries
  - Join(s)
  - Projections
  - Aggregation and Grouping & Group Restrictions
  - Query By Example

Course Outline (Ex.)

- **Transactions and Concurrency**
  - The Java Transaction
  - Hibernate Transaction Configuration
  - Hibernate Transaction API
  - Concurrency
  - Isolation Levels
  - Optimistic Locking
  - Versioning
  - Optimistic Locking without Versioning
  - Pessimistic Locking

Course Outline (Ex.)

- **Hibernate Annotation Configuration**
  - Annotation Introduction
  - Configuration
  - Mapping Files
  - Demo
- **Advanced Topics**
  - Interceptor
  - Event Listeners
  - Naming Strategy
  - ServiceRegistry
- **Hibernate Core Migration Notes**

Introduction to ORM

Lesson Outline

- What is JDBC?
- Why ORM?
- What is ORM?
- Java ORM/Persistent Frameworks

Lesson Outline

- What is JDBC?
- Why ORM?
- What is ORM?
- Java ORM/Persistent Frameworks

What is JDBC?

- Stand for Java Database Connectivity
- Set of Java API for accessing the relational databases from Java program.
  - Java API that enables Java programs to execute SQL statements and interact with any SQL-compliant database.
- Possible to write a single database application that can run on different platforms and interact with different DBMS.

Pros and Cons of JDBC

- **Pros**
  - Clean and simple SQL processing
  - Good performance with large amounts of data.
  - Very good for small applications

JDBC Pros and Cons. (Ex.)

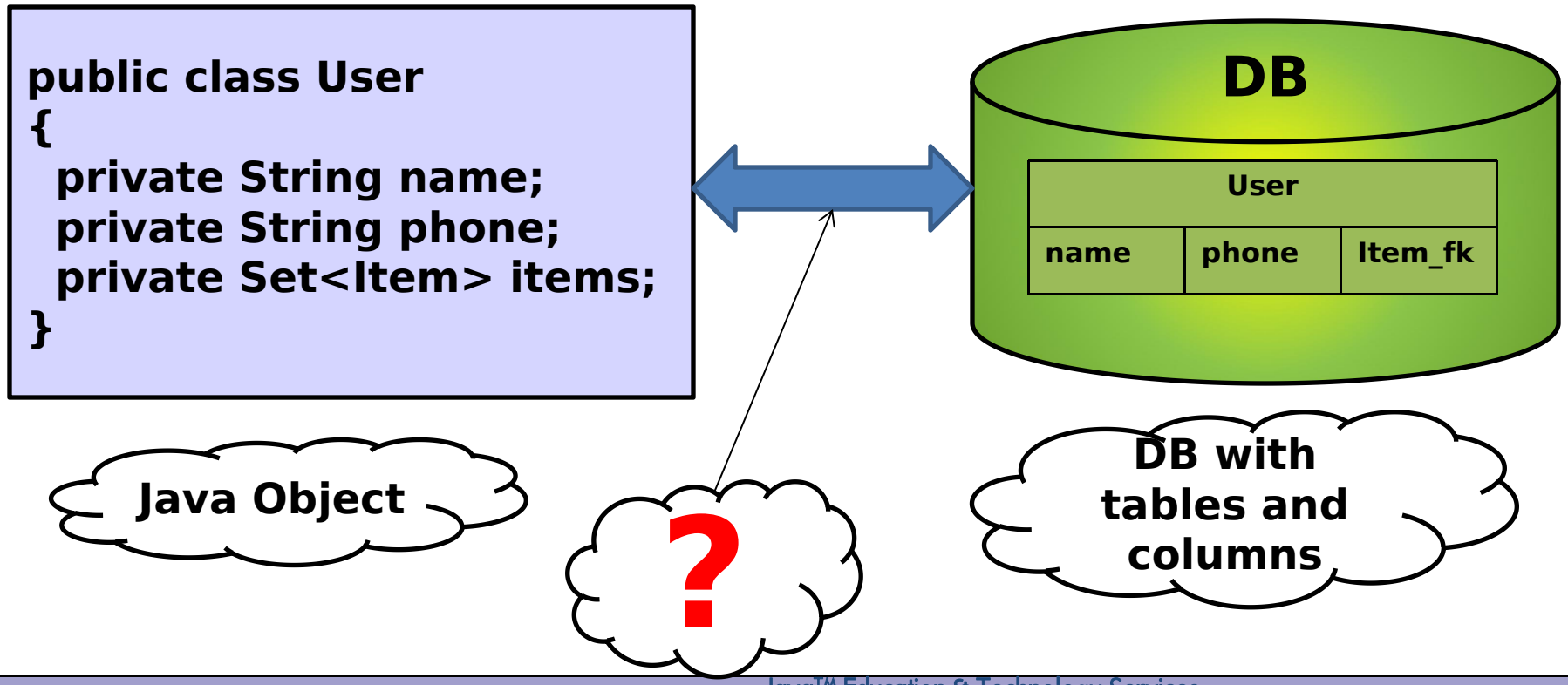
- **Cons**
  - Large programming overhead in large projects.
  - Transactions and concurrency must be hand-coded
  - Handling the JDBC connections and closing the connection is also a big issue
  - No encapsulation
  - Hard to maintain
  - Query is DBMS specific
  - Hard to implement MVC concept

Lesson Outline

- What is JDBC?
- Why ORM?
- What is ORM?
- Java ORM/Persistent Frameworks

Why ORM ? (Problem Area)

- When working with object-oriented systems,
  - There's a mismatch between
    The object model & the relational database.



Problem Area (Ex.)

```
public void addUser( User user )
{
 String sql = "INSERT INTO user
              (name,address)
              VALUES('" + user.getName()+ "','"
              + user.getAddress() + "')";

      // Initiate a Connection,
      // create a Statement,
      // and execute the statement
}
```

Problem Area (Ex.)

```
public User getUser( User user )
{
 String sql = "select * from user where
       name = '" + user.getName() +
             "' and age > " + user.getAge();

       // Initiate a Connection,
       // create a Statement,
       // and execute the query

       return user;
}
```

Problem Area (Ex.)

- Write SQL conversion methods by hand using JDBC:
  - Tedious and requires lots of code
  - Extremely error-prone
  - Non-standard SQL ties the application to specific databases
  - Difficult to represent associations between objects

The preferred solution

- Use a Object-Relational Mapping (ORM) System.
- Provides a simple API for storing and retrieving Java objects directly to and from the database.
- **Non-intrusive:** No need to follow specific rules or design patterns.
- **Transparent:** Your object model is unaware

The preferred solution

Lesson Outline

- What is JDBC?
- Why ORM?
- What is ORM?
- Java ORM/Persistent Frameworks

What is ORM ?

- **Object-Relational Mapping (ORM)**
  - is a programming technique for converting data between relational databases and object-oriented programming languages.
- Lets business code access objects rather than DB tables
- Hides details of SQL queries from OO logic
- Based on JDBC ‘under the hood’

ORM Pros

- No need to deal with the database implementation
  - Little need to write SQL statements.
- Entities based on business concepts rather than database structure
- Fast development of application
- Transaction management and automatic key generation.

Lesson Outline

- What is JDBC?
- Why ORM?
- What is ORM?
- Java ORM/Persistent Frameworks

Java ORM/Persistent Frameworks

- More than 23 Implementation for ORM.

- Some of the most common ORM Frameworks:
  - Hibernate, open source ORM framework, widely used
  - Java Persistence API (JPA), Standard
  - TopLink by Oracle
  - EclipseLink, Eclipse persistence platform
  - iBATIS(MyBatis), maintained by ASF
  - Enterprise Objects Framework, Mac OS X/Java, part of Apple WebObjects

- See more on the following Link.

Hibernate Overview

Lesson Outline

- What is Hibernate?
- Why Hibernate?
- Supporting Database Management Systems Engine

Lesson Outline

- What is Hibernate?
- Why Hibernate?
- Supporting Database Management Systems Engine

What is Hibernate?

- Hibernate is an ORM solution for JAVA.
- It is a powerful, high performance object/relational persistence and query service.
- It allows us to develop persistent classes following object-oriented idiom – including association, inheritance and polymorphism.

Lesson Outline

- What is Hibernate?
- Why Hibernate?
- Supporting Database Management Systems Engine

Why Hibernate?

- Hibernate maps Java classes to database tables using XML files
  - No need to write code for this.
  - If there is change in Database or in any table then the only need to change XML file properties.

- Abstract away the unfamiliar SQL types and provide us to work around familiar Java Objects.

Why Hibernate? (Ex.)

- Manipulates Complex associations of objects of your database.
- Provides Simple querying of data.
- Minimize database access with smart fetching strategies.
- Caching.
- Easy transaction handling.

Lesson Outline

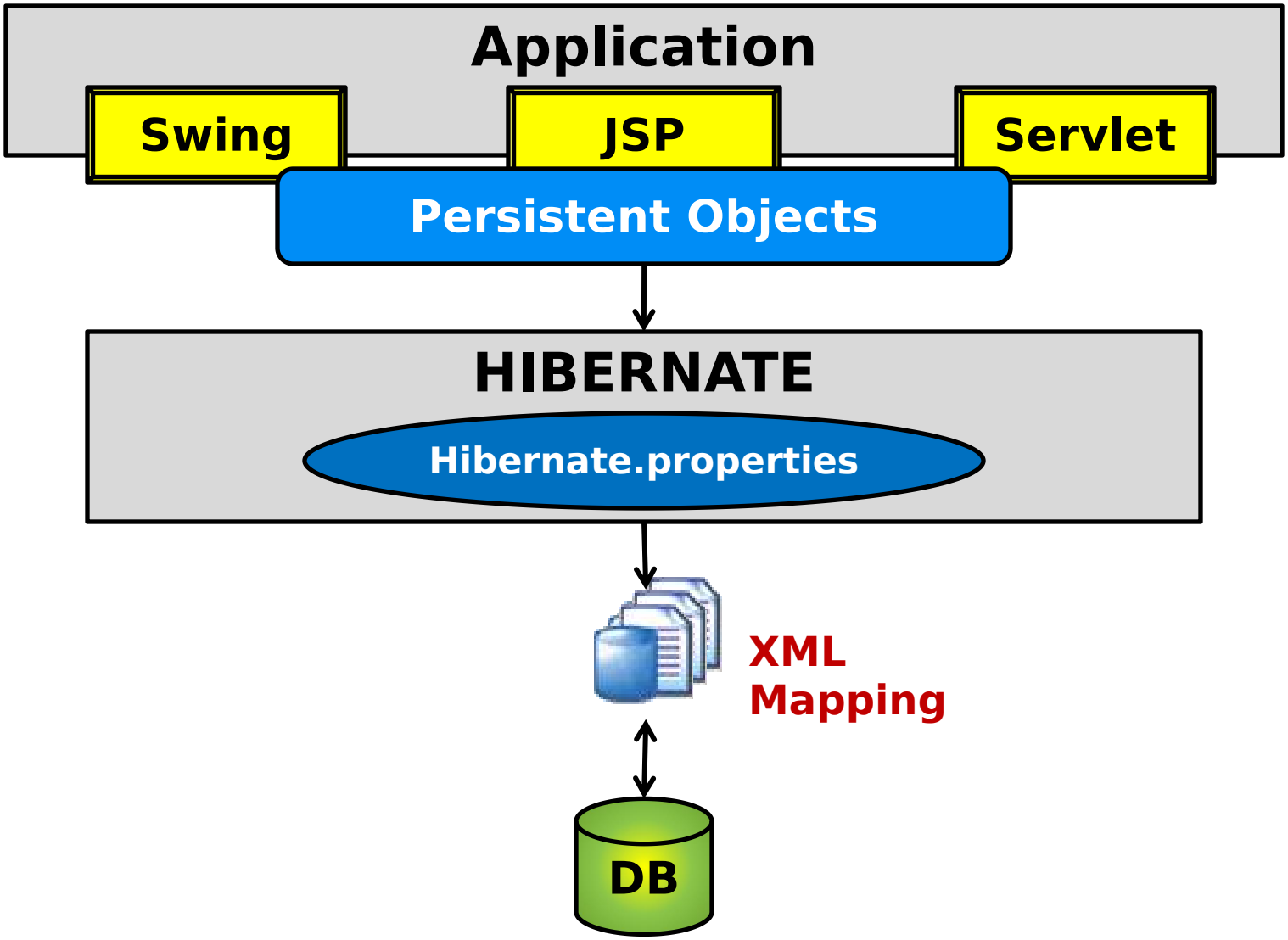
- What is Hibernate?
- Why Hibernate?
- Supporting Database Management Systems Engine

Supporting Database Engines

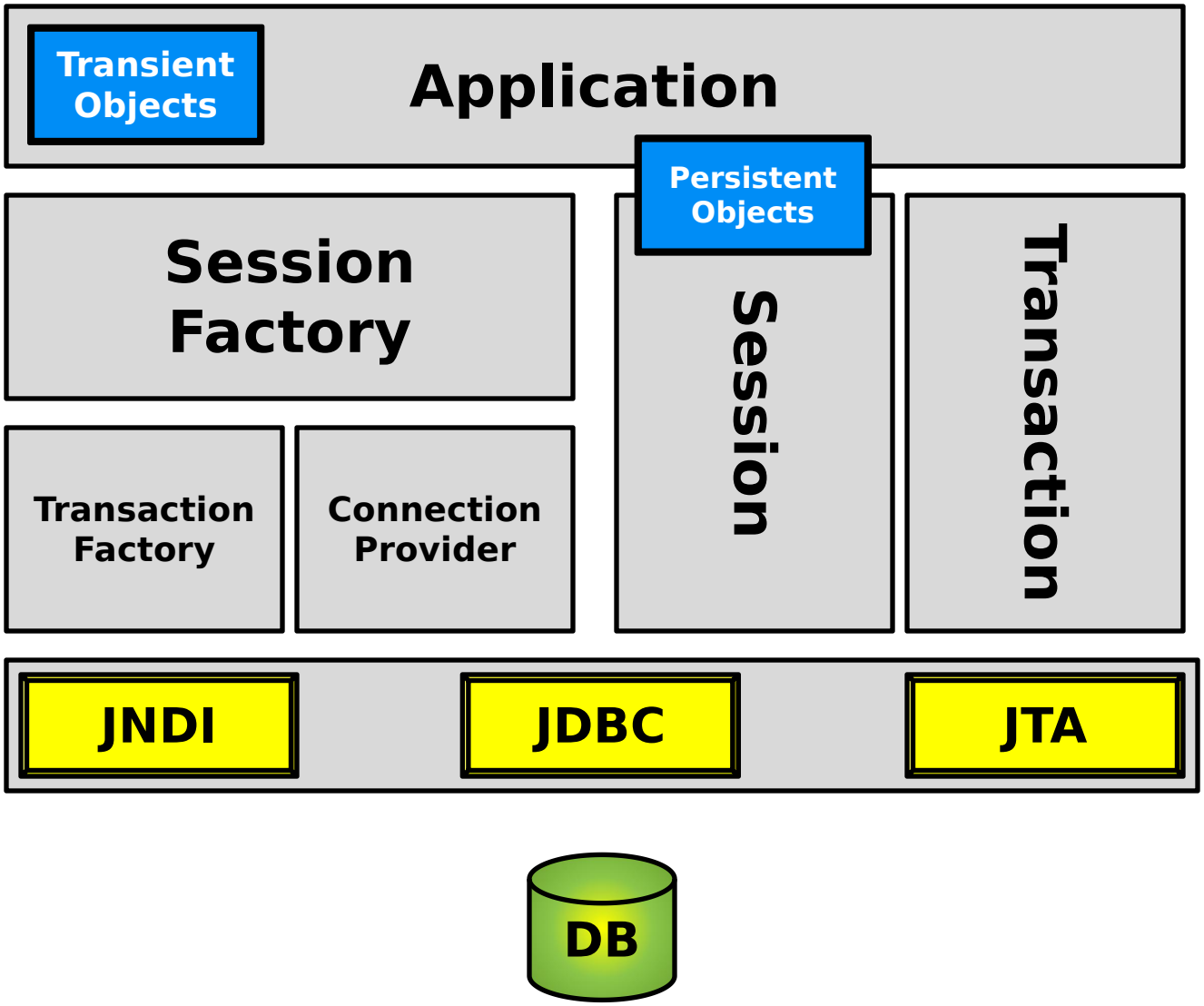
- Support more than 27 different DBMS with different base (dependant, XML)
- Some of most common DBMS:
  - **Oracle 8i, 9i, 10g, 11g**
  - **DB2 7.1, 7.2, 8.1, 9.1**
  - **Microsoft SQL Server 2000**
  - **Sybase 12.5 (JConnect 5.5)**
  - **MySQL 3.23, 4.0, 4.1, 5.0**
  - **SAP DB 7.3**
- All supported DBMS on https://community.jboss.org/wiki/SupportedDatabases2

Hibernate Architecture

Hibernate Architecture "High Level"



Hibernate Architecture "Low Level"



Hibernate Architecture

- **Session Factory :**
  - Caches mappings for a single database.
  - A factory for Session.
  - to Create the SessionFactory from hibernate.cfg.xml

```
SessionFactory fact = new Configuration()
                .configure("/hibernate.cfg.xml")
                .buildSessionFactory();
```

Hibernate Architecture (Ex.)

- **Session:**
  - A single-threaded, short-lived object representing a conversation between the application and the persistent.
  - A wrapper for the JDBC Connection.
  - A factory for Transaction
    - sessionFactory.openSession();
    - sessionFactory.getCurrentSession();

Hibernate Architecture (Ex.)

- **Transaction:**
  - Specifies atomic units of work on DB Level.
- **ConnectionProvider:**
  - A factory for (and pool of) JDBC connections
  - It is NOT exposed to the application level.
- **TransactionFactory:**
  - A factory for Transaction instances.
  - It is NOT exposed to the application level.

Hibernate Configuration

Lesson Outline

- Hibernate Installation/Setup
- Configuration Properties
- Mapping File(s)
- Development Strategies

Lesson Outline

- Hibernate Installation/Setup
- Configuration Properties
- Mapping File(s)
- Development Strategies

Hibernate Installation/Setup

- As we know hibernate is pluggable framework so to use hibernate in your application:
  - Create your project as you want (Desktop, Web, Enterprise)
  1. Add the Hibernate Core Lib
  2. Add the hibernate dependencies (mandatory for some versions)
  - However after this steps your application is ready to run hibernate, but for more usability you must configure your IDE to support Hibernate tools like eclipse.
  - Netbeans automatic enable HTools.

Lesson Outline

- Hibernate Installation/Setup
- Configuration Properties
- Mapping File(s)
- Development Strategies

Hibernate Configuration file

- **XML configuration file:**
  - Specify a full configuration in a file with standard name " **hibernate.cfg.xml** ".
  - The most important configuration parameters are:
    - Database name
    - Database dialect (type)
    - Database credentials
    - Database source name
    - Hibernate mapping files.
    - Caching settings
    - Connection Pooling settings

Hibernate Configuration file (Ex.)

```
<hibernate-configuration>
  <session-factory>
      <property name = "hibernate.connection.driver_class">
        org.gjt.mm.mysql.Driver                    </property>
      <property name = "hibernate.connection.url">
        jdbc:mysql://localhost:3306/helloworlddb
      </property>
      <property name = "hibernate.connection.username">
        root                                       </property>
      <property name ="hibernate.connection.password">
        root                                       </property>
      <property name ="hibernate.dialect">
        org.hibernate.dialect.MySQLInnoDBDialect
      </property>
   <mapping resource="dao/Person.hbm.xml"/>
 </session-factory>
</hibernate-configuration>
```

Lesson Outline

- Hibernate Installation/Setup
- Configuration Properties
- Mapping File(s)
- Development Strategies

Hibernate Mapping file

- XML documents that defines
  - the mapping between the java class DAO and the correspondence database table.
- Can be generated from the Java source , database using specific tools

Hibernate Mapping file (Ex.)

- It Defines the mapping of these to the database.
  - Mapping Classes to Tables
  - Mapping Properties to Columns
  - Mapping Id field
  - Generating object identifiers
  - Key Generation approach.
  - object relationships types (one to one, one to many ,...)
  - Fetching strategies

Hibernate Mapping file (General Form)

```
<hibernate-mapping>
  <class name="class-name" table="corresponding-table-
      name">
      <id name="attribute-name" column="column-name"
        type="hibernate-type" >
        <generator class="generator-class"/>
      </id>
      <property name="property-name" column="column-name"
         type="hibernate-type"/>
      <property name="property-name" column="column-name"
         type="hibernate-type"/>
  </class>
</hibernate-mapping>
```

Hibernate Mapping file (Example)

```
<hibernate-mapping>
  <class name="dao.Person" table="person"
      catalog="helloorm">
      <id name="id" column="id" type="int" >
        <generator class="identity"/>
      </id>
      <property name="name" column="name" type="string"/>
      <property name="address" column="address"
       type="string"/>
      <property name="phone" column="phone" type="string"/>
      <property name="email" column="email" type="string"/>
      <property name="birthday" column="birthday"
type="date"/>
  </class>
</hibernate-mapping>
```

Generator

- The optional **<generator>** element used to generate unique identifiers for instances of the persistent class.

- There are built-in generators :

  1. Increment
  2. Identity
  3. Sequence
  4. Native

Generator

- **increment**
  - generates identifiers of type **long**, **short** or **int** that are unique only when no other process is inserting data into the same table.
  - *Do not use in a cluster.*
- **identity**
  - supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL.
  - The returned identifier is of type **long**, **short** or **int**.

Generator

- **sequence**
  - uses a sequence in DB2, PostgreSQL, Oracle, SAP DB, McKoi or a generator in Interbase.
  - The returned identifier is of type **long**, **short** or **int**

- **native**
  - picks identity, sequence or hilo depending upon the capabilities of the underlying database.

Custom Generators

- All generators implement the interface **org.hibernate.id.IdentifierGenerator**
- some applications may choose to provide their own specialized implementations by implement **IdentifierGenerator** interface.

Lesson Outline

- Hibernate Installation/Setup
- Configuration Properties
- Mapping File(s)
- Development Strategies

Development Strategies

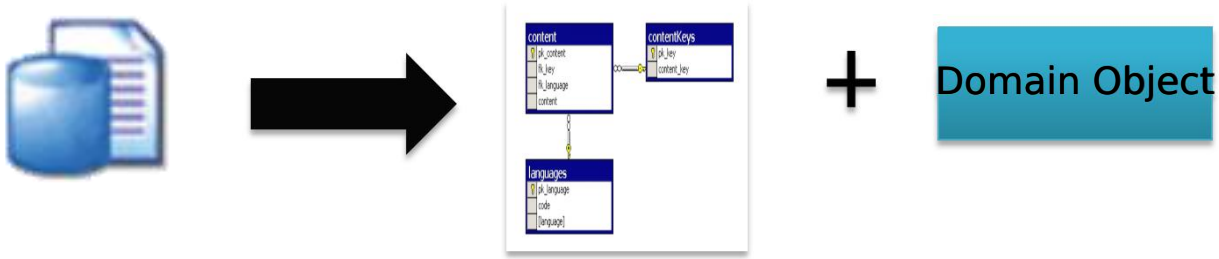
- Selecting a development Strategy :
  - **Top down**
    - Create your java domain object,
    - Create xml mapping files and
    - Generate the database schema.

Development Strategies (Ex.)

- **Bottom up**
  - Create database and then
  - Generate the xml mapping files and java domain objects.

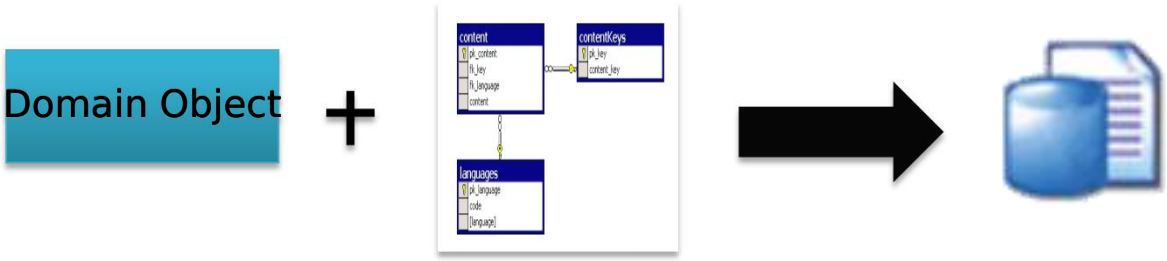
Development Strategies (Ex.)

- **Middle out**
  - Write xml mapping files and then
  - Generate java domain objects and database



Development Strategies (Ex.)

- **Meet in the middle**
  - You already have java classes and database.
  - This scenario usually requires at least some re-factoring of the Java classes, database schema, or both.



First Example (Hello Hibernate)

Hello Hibernate

- Steps to create your first application using Hibernate:
  - Create your java project
  - Add the required jars to your project
  - Create a java bean that'll represent the corresponding table
  - Create the XML mapping file which should be saved as **className.hbm.xml** and located near the class
  - Create **hibernate.cfg.xml** to configure the Hibernate
  - Create a test class to insert object from the DB.

Hello Hibernate (Ex.)

```
public class Account{
    private int id;
    private String userName;
    private String fullName;
    private String phone;
    private String address;
    private String password;
    private Date birthday;

    public Account()
    {   }

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    // getter & setter for all attributes
}
```

Account.java

Hello Hibernate (Ex.)

```
<hibernate-mapping>
  <class name="dao.Account" table="account"
      catalog="helloworlddb">
      <id name="id" column="id" type="int" >
        <generator class="identity"/>
      </id>
      <property name="userName" column="user_name"
type="string"/>
      <property name="fullName" column="full_name"/>
      <property name="password" column="password"/>
      <property name="address" column="address"/>
      <property name="phone" column="phone"
type="string"/>      <property name="birthday"
column="birthday"                  type="date"/>
  </class>
</hibernate-mapping>
```

Account.hbm.xml

Hello Hibernate (Ex.)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "...">
<hibernate-configuration>
  <session-factory>
      <property name = "hibernate.connection.driver_class">
        org.gjt.mm.mysql.Driver                          </property>
      <property name = "hibernate.connection.url">
       jdbc:mysql://localhost:3306/helloworlddb  </property>
      <property name = "hibernate.connection.username">
        root                                             </property>
      <property name ="hibernate.connection.password">
        root                                             </property>
      <property name ="hibernate.dialect">
        org.hibernate.dialect.MySQLInnoDBDialect      </property>
   <mapping resource="dao/Account.hbm.xml"/>
 </session-factory>
</hibernate-configuration>
```

hibernate.cfg.xml

Hello Hibernate (Ex.)

```
public class Test {
   public static void main(String[] args) {
        SessionFactory sessionFactory = new Configuration()
          .configure().buildSessionFactory();
       Session session = sessionFactory.openSession();

          Account account = new Account();
          account.setName("Medhat");
          account.setPhone("0235355637");
          account.setBirthday(new Date());
          account.setEmail("ahyousif@mcit.gov.eg");

          session.beginTransaction();
          session.persist(account);
          session.getTransaction().commit();
          System.out.println("Insertion Done");
   }
}
```

Test.java

Hibernate
Entity Life-Cycle

Lesson Outline

- Entity Life-Cycle (Object States)
- Session Operations

Lesson Outline

- Entity Life-Cycle (Object States)
- Session Operations

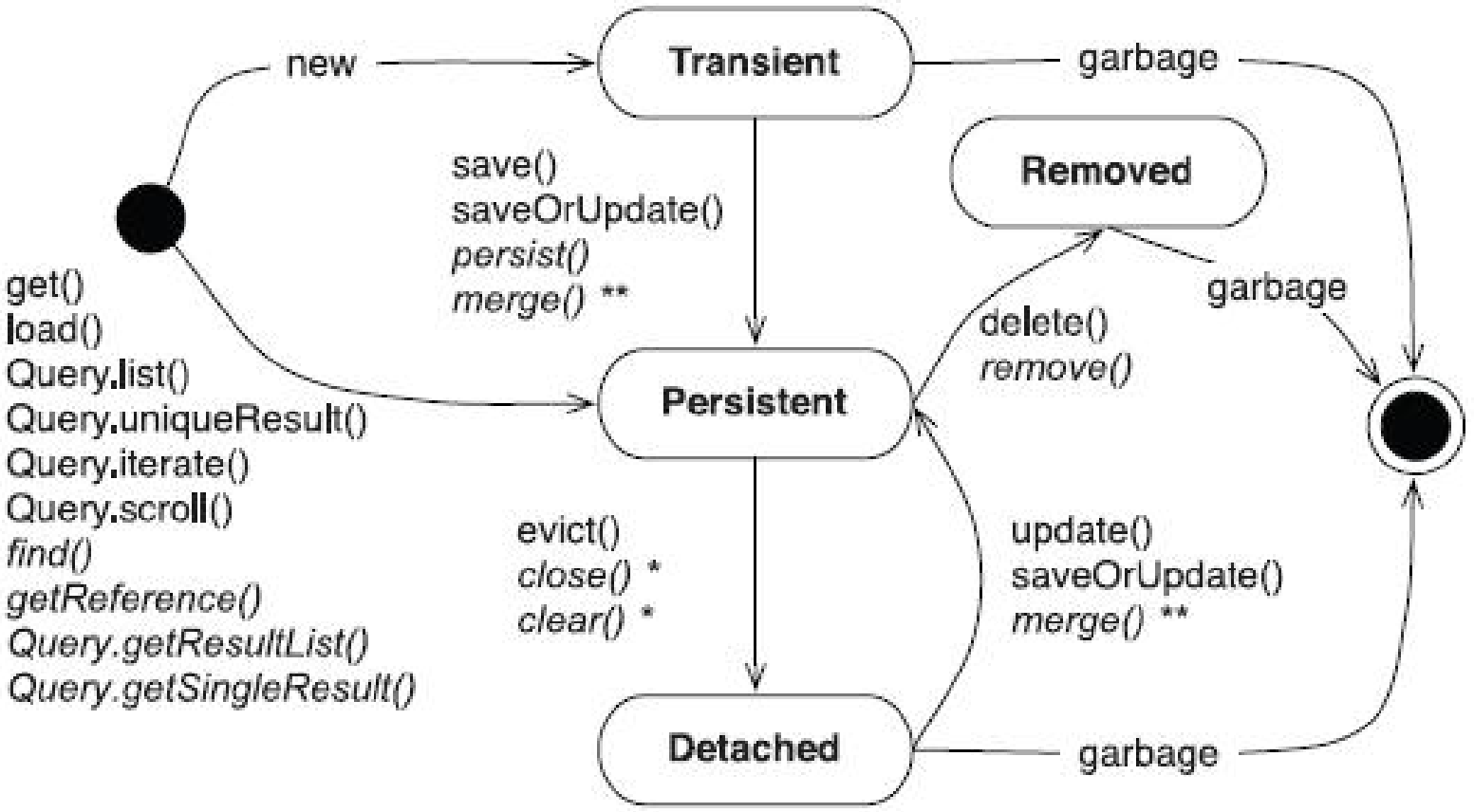
Entity Life-Cycle (Object states)

- Any application with persistent state must call Hibernate interfaces to store and load objects.

- it's necessary for the application to concern itself with the state and lifecycle of an object with respect to persistence.

  - We refer to this as the ***Persistence lifecycle***

Entity Life-Cycle (Object states) Diagram



* Hibernate & JPA, affects all instances in the persistence context
** Merging returns a persistent instance, original doesn't change state

Transient Objects

- Objects instantiated using the new operator **aren't** immediately persistent.
  - Their state is *transient*, which means they aren't associated with any database table row
- Hibernate consider all transient instances to be non transactional;
  - any modification of a transient instance isn't known to Session and doesn't propagated to DB.
- Hibernate doesn't provide any roll-back functionality for transient objects.

Persistent Objects

- A persistent instance is an entity instance with a database identity
- That means a persistent and managed instance has a primary key value set as its database identifier
- Hibernate caches them and can detect whether they have been modified by the application.
  - And changes are reflected to the database tables.

Detached Objects

- When Session closed
  - All *persistent* instance become detached instance.
- Which means that their state is no longer guaranteed to be synchronized with database state;
  - they’re no longer attached to a Session . They still contain persistent data (which may soon be stale).
- You can continue working with a detached object and modify it.

Detached Objects (Ex.)

- However, at some point you probably want to make those changes persistent
- In other words, bring the detached instance back into persistent state.
- Hibernate offers two operations, ***reattachment*** and ***merging***, to deal with this situation.

Removed Objects

- An object is in the *removed* state if it has been scheduled for deletion at the end of a unit of work.

Lesson Outline

- Entity Life-Cycle (Object States)
- Session Operations

Session Operations

- Session interface provides methods for these operations :
  - Saving objects

    Person p = new Person();

    session.save(p); or session.persist(p);

  - Loading objects

    load(Class theClass, Serializable id)

    - This method will **throw an exception** if the unique id is not found in the database

  - Getting objects

    get(Class theClass, Serializable id)

    - This method will **return null** if the unique id is not found in the database

Session Operations (Ex.)

- Refreshing objects
  - When Your Hibernate application is not the only application working with this data you can use it

    refresh(Object object)
- Updating objects
  - **update(Object object)**
  - **saveOrUpdate(Object)**
  - **merge(Object object);**
  - saveOrUpdate and merge methods create a new one if the object is not persisted
- Deleting objects
  - Removes an object from the database
  - **delete (Object object)**
- Querying objects

Useful Tips …

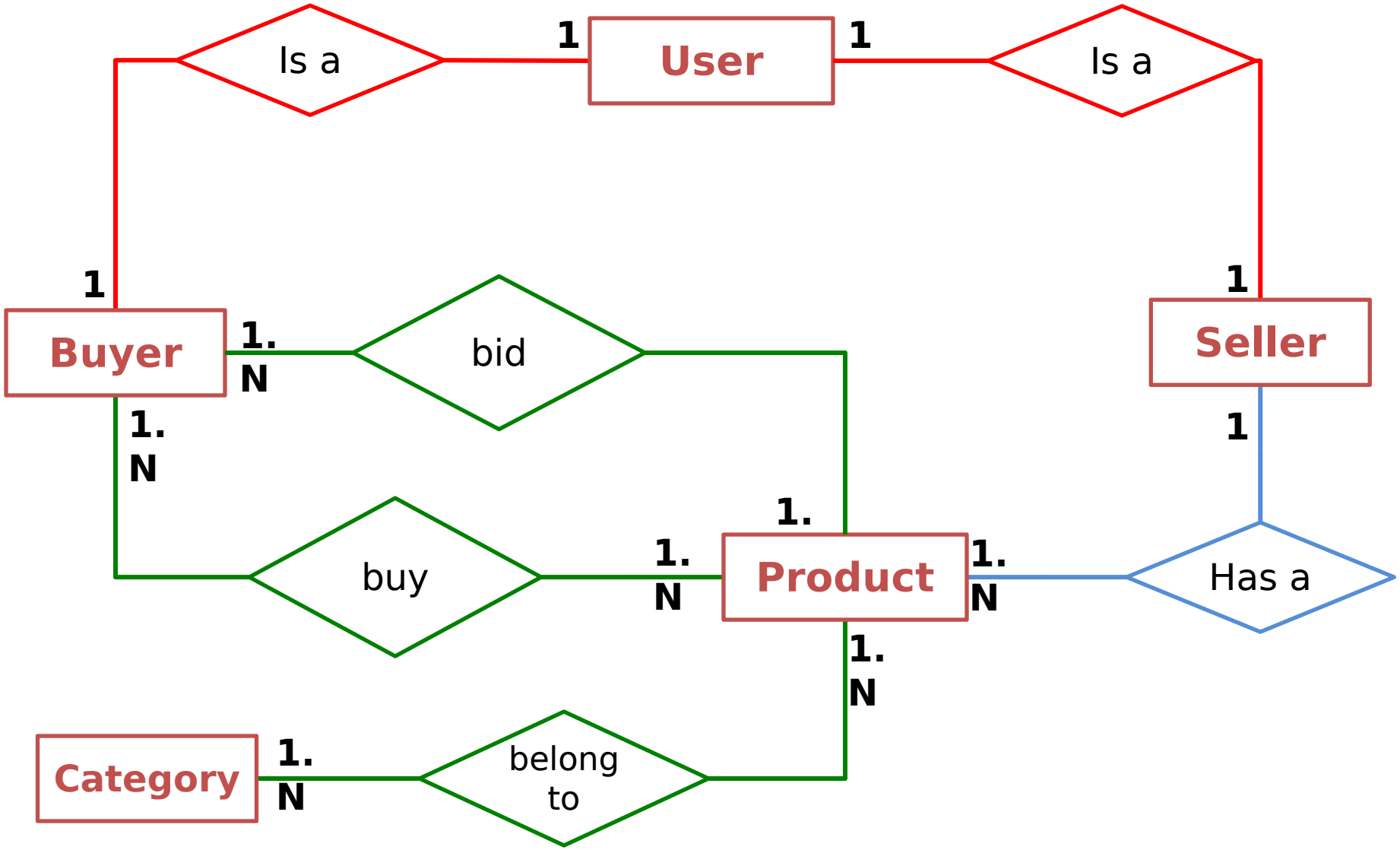
- Generating Schema from xml configuration files :
  - Configuration configuration = **new** Configuration();
  - configuration = configuration.configure (*CONFIG_FILE_LOCATION*);
    SchemaExport schemaExport = **new** SchemaExport(configuration);
    schemaExport.create (**false**, **true**);
- Printing the generated SQL :
  - <property name="hibernate.format_sql">true</property>
  - <property name="hibernate.show_sql">true</property>
- Enable the getCurrentSession() :
  - <property name = "hibernate.current_session_context_class">
    thread
    </property>

Entities Associations

Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

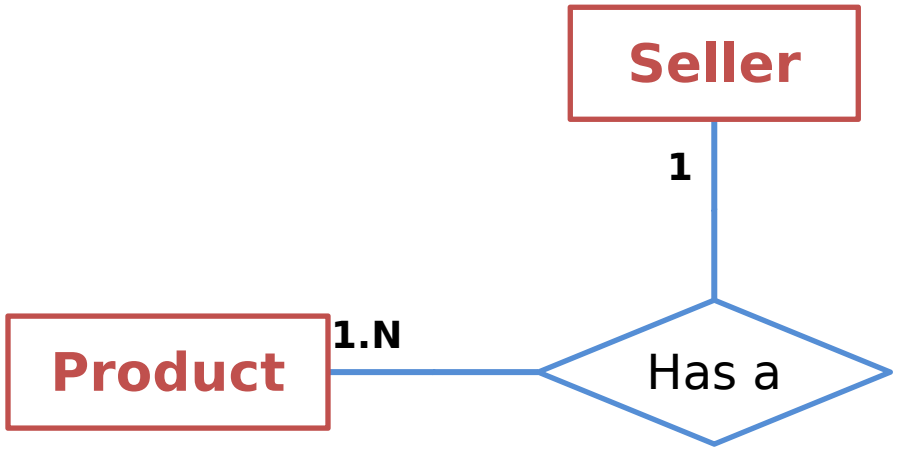
DB Diagram



Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

DB Diagram (Ex.)



Many-to-one Association

- Association from product to seller is a **many-to-one** association.

- Since associations are directional(Uni-directional), you classify the inverse association from seller to product as: a **one-to-many** association, And it's called bidirectional

- To map association from student to group,
  - We need two properties in two classes.
    - One is a collection of references, and
    - The other a single reference.

Many-to-one Uni-directional

```
public class Product{
    ...
    private Seller seller;

    public Seller getSeller() {
        return seller;
    }
    public void setSeller(Seller seller) {
        this.seller = seller;
    }
    ...
}
```

Product.java

Many-to-one Uni-directional (Ex.)

```
<hibernate-mapping>
   <class name="dao.Product" table="product"
      catalog="biddingschema">

      ...

      <many-to-one name="seller" class="dao.Seller">

        <column name="seller_id"/>

      </many-to-one>

      ...

   </class>

</hibernate-mapping>
```

Product.hbm.xml

Many-to-one Uni-directional (Ex.)

- If you need the seller instance for which a particular product was selected,
  - call prodcutObject.getSeller(), utilizing the entity association you created.
  - On the other hand, if you need all products that have been offered by a specific seller, you can write a query (in whatever language Hibernate supports).
- One of the reasons you use a tool like Hibernate
  - is, of course, that you don't want to write that query.

Many-to-one Bi-directional

- You want to be able to fetch all products offered by a particular seller without an explicit query,
  - By : sellerObject.getProducts().iterator() **

Many-to-one Bi-directional (Ex.)

```
public class Seller{
    ...
    private set products = new HashSet();

    public set getProducts() {
        return products;
    }
    public void setProducts(set products) {
        this.products = products;
    }
    ...
}
```

Seller.java

Many-to-one Bi-directional (Ex.)

```
<hibernate-mapping>
  <class name="dao.Seller" table="seller"
    catalog="biddingschema">
    ...
    <set name="products" table="product“ inverse="true">
        <key><column name="seller_id" /></key>
        <one-to-many class="dao.Product" />
    </set>
    ...
  </class>
</hibernate-mapping>
```

Seller.hbm.xml

Many-to-one Bi-directional (Ex.)

- The content of the collection is mapped with element, <one-to many>.
- The column mapping defined by the <key> element is the foreign key column seller_id of the product table,
  - The same column you already mapped on the other side of the relationship

- The inverse attribute tells Hibernate that
  - the collection is a mirror image of the <many-to-one> association on the other side.

Many-to-one Bi-directional (Ex.)

- Without the inverse attribute,
  - Hibernate tries to execute two different SQL statements, both updating the same foreign key column,
- when use inverse="true",
  - you explicitly tell Hibernate which end of the link it should not synchronize with the database.
    - In this example, you tell Hibernate that it should propagate changes made at the product end of the association to the database,
    - ignoring changes made only to the products collection.

Many-to-one Bi-directional (Ex.)

```
public class Seller{
    ...
    private set products = new HashSet();

    public set getProducts() {
        return products;
    }
    public void setProducts(set products) {
        this.products = products;
    }
    public void addProduct(Product product) {
        product.setSeller(this);
        products.add(product);
    }
    ...
}
```

Seller.java

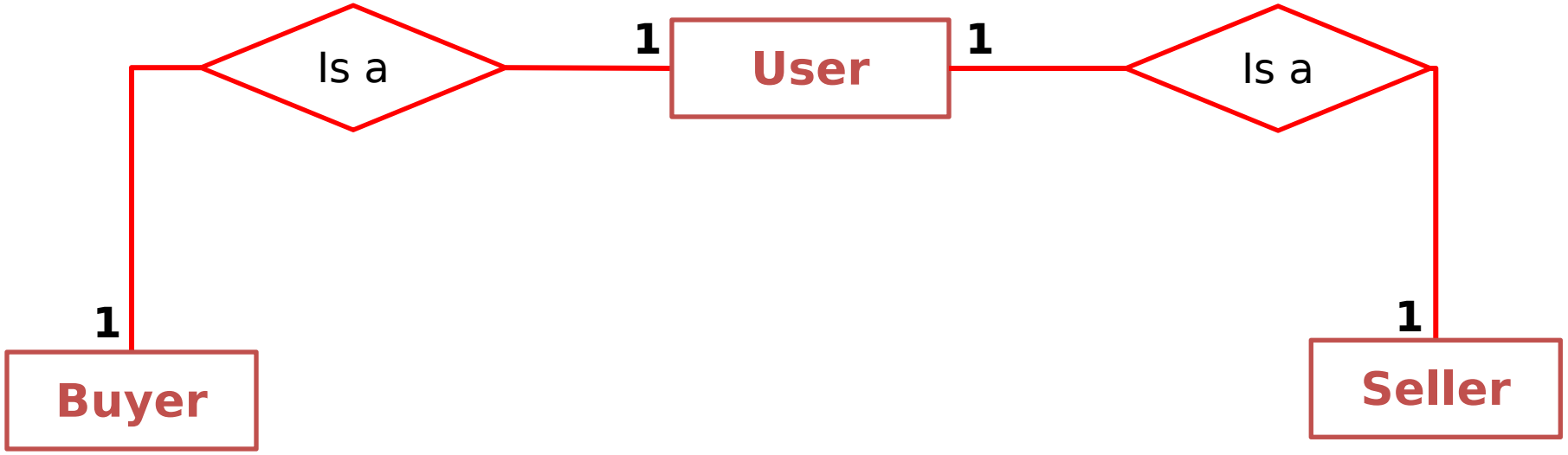
Many-to-one Bi-directional (Ex.)

- If you only call sellerObject.getProducts().add(Product),
  - no changes are made persistent!
- You get what you want only if the other side,
  - productObject.setSeller(sellerObject), is set correctly.
- It's the primary reason why you need convenience methods such as
  - addStudent()
    - they take care of the bi-directional references in a system without container- managed relationships.

Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

DB Diagram (Ex.)



One-to-One

- Rows in two tables related by a primary key association.
  - share the same primary key values.
- The main difficulty with this approach is
  - ensuring that associated instances are assigned the same primary key value when the objects are saved.

One-to-One (Ex.)

```
public class User{
    ...
    private Seller seller;

    public Seller getSeller() {
        return seller;
    }
    public void setSeller(Seller seller) {
        this.seller = seller;
    }
    ...
}
```

User.java

One-to-One (Ex.)

```
<hibernate-mapping>
   <class name="dao.User" table="user"
      catalog="biddingschema">

      ...

      <one-to-one name="seller" class="dao.Seller"/>    ...

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

User.hbm.xml

One-to-One (Ex.)

- When save User and its Seller
  - Hibernate inserts a row into the User table and a row into the Seller table.
  - But How can Hibernate possibly know that the record in the Seller table needs to get the same primary key value as the User row?
  - To do that
    - You need to enable a special identifier generator.

- So, If a Seller instance is saved, it needs to get the primary key value of a User object
  - You can't enable a regular identifier generator
    - Like database sequence

One-to-One (Ex.)

```
public class Seller{
    ...
    private User user;
    private int id;
    public int getId()                  {   return id;          }
    public void setId(int id) {   this.id = id; }
    public User getUser() {
        return user;
    }
    public void setUser(User user) {
        this.user = user;
    }
}
```

Seller.java

One-to-One (Ex.)

```
<hibernate-mapping>
  <class name="dao.Seller" table="seller"
      catalog="biddingschema">
      <id name="id" column="id" type="int" >
            <generator class="foreign">
                  <param name="property">user</param>
            </generator>
      </id>
     <one-to-one name="user" class="dao.User"
constrained="true"/>

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

Seller.hbm.xml

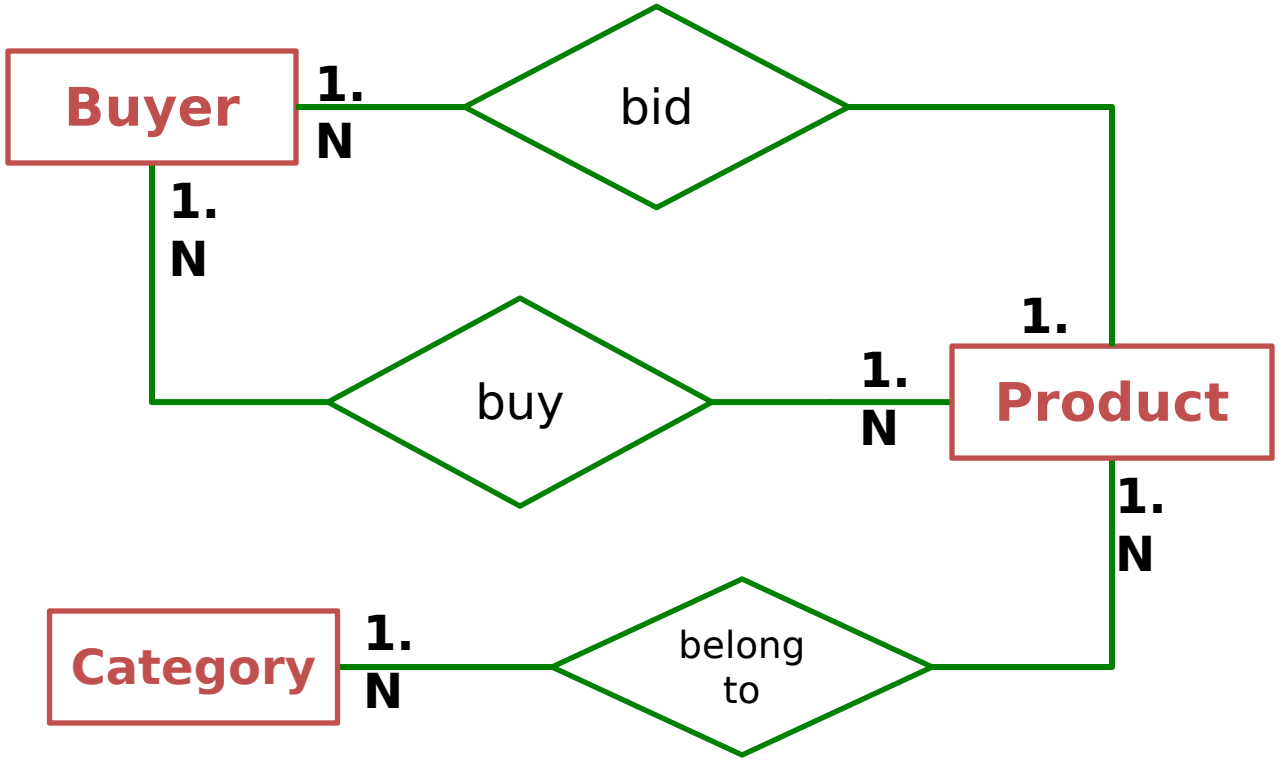
One-to-One (Ex.)

- With constrained = "true",
  - adds a foreign key constraint linking the primary key of the Seller table to the primary key of the user table.
- You can now use the special foreign identifier generator for Seller objects.
- When a Seller is saved, the primary key value is taken from the user property.
  - The user property is a reference to a User object;
  - hence, the primary key value that is inserted is the same as the primary key value of that instance.
- Note: The other way to handle relation one-to-one in db does supported in hibernate as many-to-one relation.

Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

DB Diagram (Ex.)



Many-to-Many

- A many-to-many association may always be represented as two many-to-one associations to an intervening class.
  - This model is usually more easily extensible,
  - so we tend not to use many-to-many associations in applications
- Also, remember that you don't have to map *any collection of entities,*
  - *You* can always write an explicit query instead of direct access through iteration.

Many-to-Many (Ex.)

- The join table *has two columns: the foreign* keys of the Product and Category tables.
- The primary key is a composite of both columns.
- Creating a link between a Product and a Category is easy:
  - productObject.getCategories().add(categoryObject);
  - categoryObject.getProducts().add(productObject);

Many-to-Many Uni-directional

```
public class Product{
    ...
    private set categories = new HashSet();

    public set getCategories() {
        return categories;
    }
    public void setCategories(set categories) {
        this.categories = categories;
    }
    ...
}
```

Product.java

Many-to-Many Uni-directional (Ex.)

```
<hibernate-mapping>
  <class name="dao.Product" table="product"
     catalog="biddingschema">
     ...
   <set name="categories" table="product_category">
        <key>  <column name="product_id" />   </key>
        <many-to-many entity-name="dao.Category" >
         <column name="category_id" not-null="true"/>
        </many-to-many>
     </set>
</class>
</hibernate-mapping>
```

Product.hbm.xml

Many-to-Many Bi-directional

- An association between a Product and a Category is represented in memory by
  - the Category instance in the categories collection of the Product,
  - And the Product instance in the products collection of the Category.

- The code to create the object association also changes:
  - productObject.getCategories().add(categoryObject);
  - categoryObject.getProducts().add(productObject);

Many-to-Many Bi-directional (Ex.)

```
public class Category{
    ...
    private set products = new HashSet();

    public set getProducts() {
        return products;
    }
    public void setProducts(set products) {
        this.products = products;
    }
    ...
}
```

Category.java

Many-to-Many Bi-directional (Ex.)

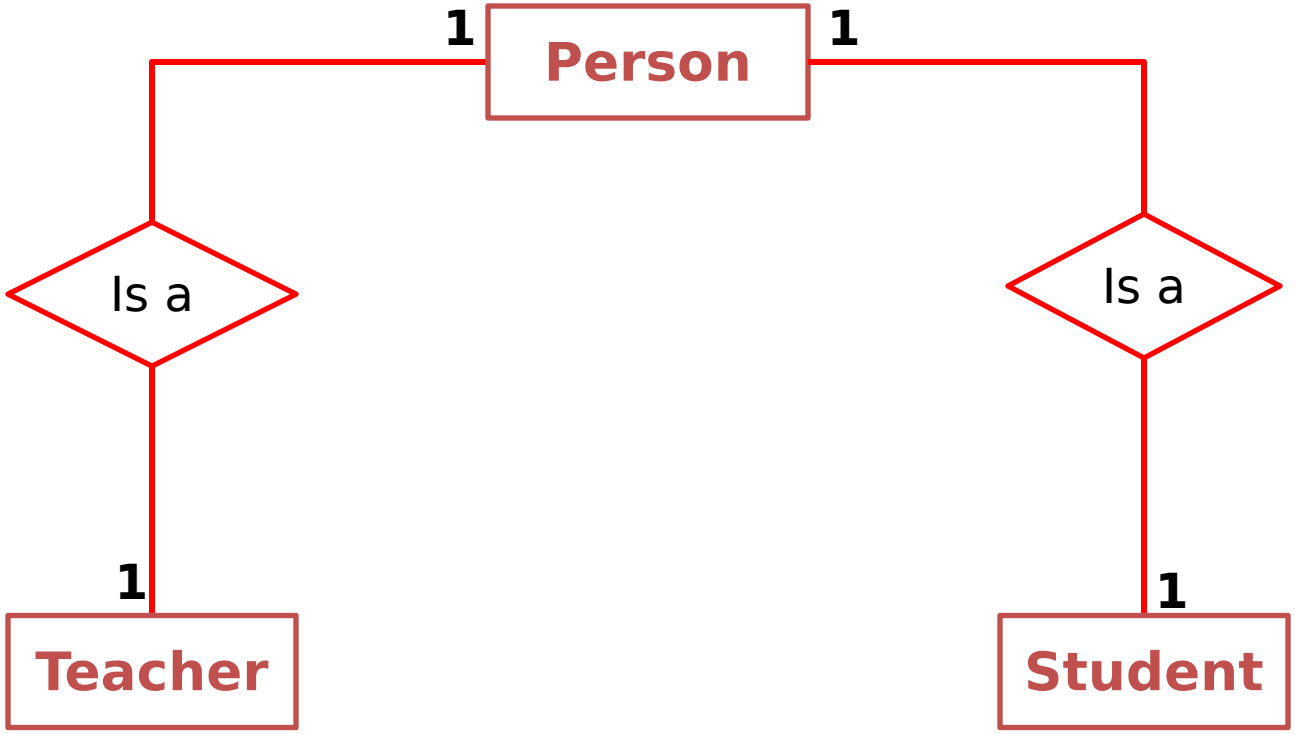
```
<hibernate-mapping>
  <class name="dao.Category" table="category"
     catalog="biddingschema">
     ...
   <set name="products" table="product_category" inverse="true">
        <key>  <column name="category_id" />   </key>
        <many-to-many entity-name="dao.Product" >
         <column name="product_id" not-null="true"/>
        </many-to-many>
     </set>
</class>
</hibernate-mapping>
```

Category.hbm.xml

Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

DB Diagram 2 (Inheritance)

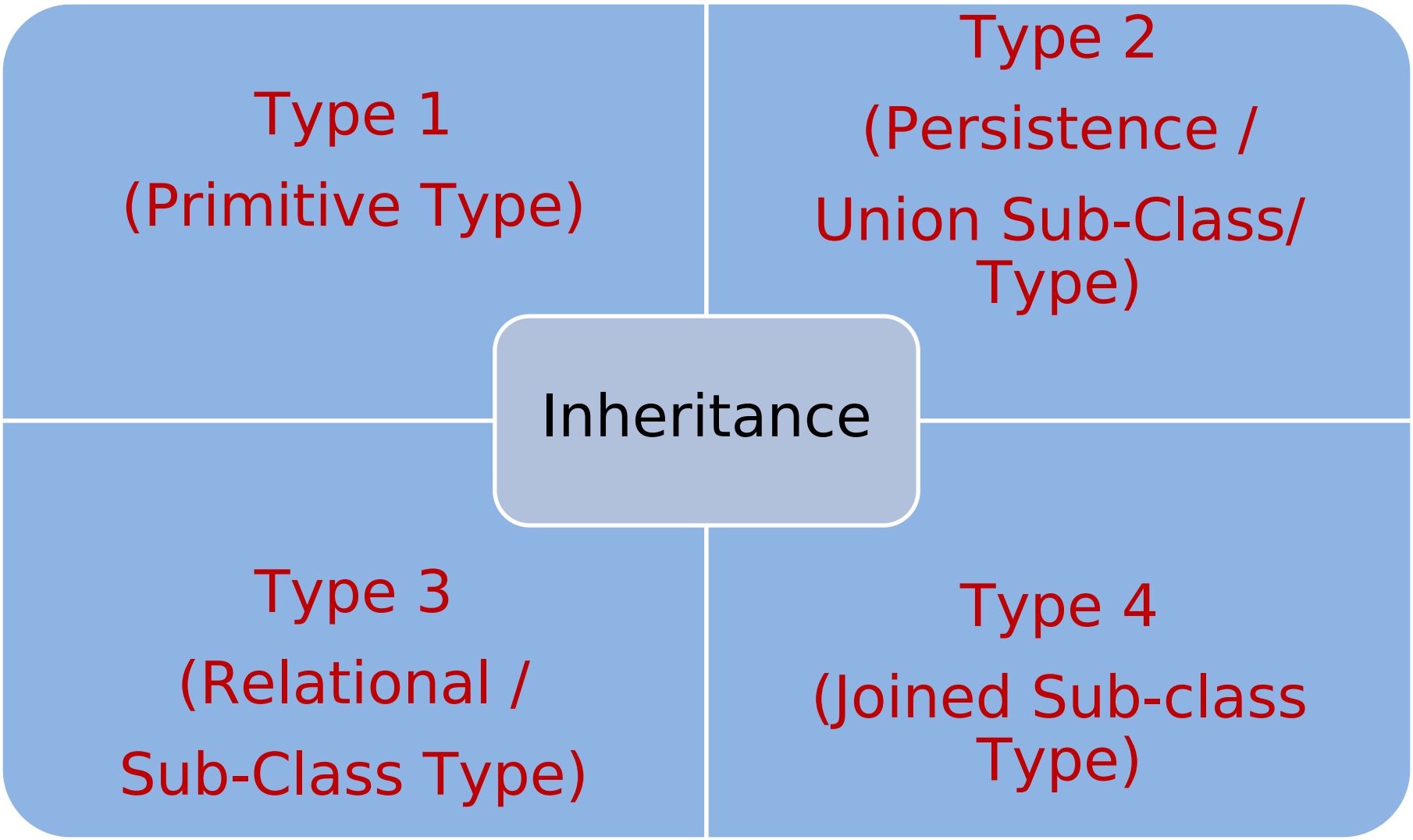


Inheritance Association

- There are 4 ways to represent the Inheritance Relation.

- Each of them are different in three areas.
  - Database.
  - Classes.
  - Relations Mapping.

Inheritance Association



Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

Type 1 (Primitive Type)

- Database: Represented in three tables.

Person (Base)		Teacher (Derived)		Student (Derived)	
id	Integer	id	Integer	id	Integer
first_name	String	first_name	String	first_name	String
last_name	String	last_name	String	last_name	String
		hire_date	String	department	String

- First Three column are repeated with the same name & type.

Type 1 (Primitive Type) (Ex.)

- Classes: Represented in three classes
  - Parent Class (Person):
    - Contains abstraction of data (id, first_name, last_name)
  - Child Class (Teacher):
    - Extends Parent & contains added properties as (hire_date)
  - Child Class (Student):
    - Extends Parent & contains added properties as (department)

Type 1 (Primitive Type) (Ex.)

- Relations Mapping: Represented in three separated classes with no relation between them.

```
<hibernate-mapping package="pojo">
      <class name="Person" table="person">
       <id name="id" column="id">
          <generator class="increment"/>
        </id>
        <property name="firstName" column="first_name">
        <property name="lastName" column="last_name">
      </class>
</hibernate-mapping>
```

Parent

Type 1 (Primitive Type) (Ex.)

```
<hibernate-mapping package="pojo">
  <class name="Student" table="student">
      <id name="id" column="id">
         <generator class="increment"/>
    </id>
    <property name="firstName" column="first_name">
    <property name="lastName" column="last_name">
    <property name="department" column="department">
  </class>
</hibernate-mapping>
```

Child(1)

Type 1 (Primitive Type) (Ex.)

```
<hibernate-mapping package="pojo">
  <class name="Teacher" table="teacher">
      <id name="id" column="id">
         <generator class="increment"/>
    </id>
    <property name="firstName" column="first_name">
    <property name="lastName" column="last_name">
    <property name="hireDate" column="hire_date"
        type="date">
  </class>
</hibernate-mapping>
```

Child(2)

Type 1 (Primitive Type) (Ex.)

- Cons:
  - Relations aren’t represented in database.
  - Relations aren’t represented in mapping files.
  - Column are repeated with the same name & type in each derived.

Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

Type 2 (Union Sub-Class Type)

- Database: Represented in three tables. (Same as type1)

Person (Base)		Teacher (Derived)		Student (Derived)	
id	Integer	id	Integer	id	Integer
first_name	String	first_name	String	first_name	String
last_name	String	last_name	String	last_name	String
		hire_date	String	department	String

- First Three column are repeated with the same name & type.

Type 2 (Union Sub-Class Type) (Ex.)

- Classes: Represented in three classes. (Same as type1)
  - Parent Class (Person):
    - Contains abstraction of data (id, first_name, last_name)
  - Child Class (Teacher):
    - Extends Parent & contains added properties as (hire_date)
  - Child Class (Student):
    - Extends Parent & contains added properties as (department)

Type 2 (Union Sub-Class Type) (Ex.)

```
<hibernate-mapping package="pojo">
 <class name="Person" table="person">
      <id name="id" column="id">
       <generator class="increment"/></id>
      <property name="firstName" column="first_name">
      <property name="lastName" column="last_name">
      <union-subclass name="Student" table="student">
        <property name="department" column="department">
      </union-subclass>
      <union-subclass name="Teacher" table="teacher">
        <property name="hireDate" column="hire_date"
                  type="date">
      </union-subclass>
 </class>
</hibernate-mapping>
```

Type 2 (Union Sub-Class Type) (Ex.)

- Pros:
  - Relations are represented in mapping files.
- Cons:
  - Relations aren’t represented in database.
  - Column are repeated with the same name & type in each derived.

Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

Type 3 (Sub-Class Type)

- Database: Represented in one table only.

Person (Base, Derived 1,Derived 2)	
id	Integer
first_name	String
last_name	String
hire_date	String
department	String
person_type	String

Type 3 (Sub-Class Type) (Ex.)

- Classes: Represented in three classes
  - Parent Class (Person):
    - Contains abstraction of data (id, first_name, last_name)
  - Child Class (Teacher):
    - Extends Parent & contains added properties as (hire_date)
  - Child Class (Student):
    - Extends Parent & contains added properties as (department)
- There is no property for person_type.

Type 3 (Sub-Class Type) (Ex.)

- Relations Mapping:
  - Represented in separated classes also derived classes.
- Discriminator:
  - It’s an indicator to specify which instance is created from this tabled.
- Note: Any added properties must be enabled null.

Type 3 (Sub-Class Type) (Ex.)

```
<hibernate-mapping package="pojo">
 <class name="Person" table="person">
      <id name="id" column="id">
       <generator class="increment"/></id>
      <discriminator column="person_type" type="String"/>
      <property name="firstName" column="first_name"/>
      <property name="lastName" column="last_name"/>

 </class>
 <subclass name="Student" extends="person"
           discriminator-value="Student">
      <property name="department" column="department">
 </subclass>
 <subclass name="Teacher" extends="person"
           discriminator-value="Teacher">
      <property name="hireDate" column="hire_date"
                type="date">
 </subclass>
</hibernate-mapping>
```

Type 3 (Sub-Class Type) (Ex.)

- Pros:
  - Relations are represented in mapping files.
- Cons:
  - Any added properties in derived classes must be null-enabled.
  - Relations aren’t represented in database.
  - No separation between data in tables for all (base & derived).

Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

Type 4 (Joined Sub-Class Type)

- Database: Represented in three tables.

Person (Base)		Teacher (Derived)		Student (Derived)	
id	Integer	id	Integer	id	Integer
first_name	String	hire_date	String	department	String
last_name	String				

- First column is repeated with the same name & type.

Type 4 (Joined Sub-Class Type) (Ex.)

- Classes: Represented in three classes.
  - Parent Class (Person):
    - Contains abstraction of data (id, first_name, last_name)
  - Child Class (Teacher):
    - Extends Parent & contains added properties as (hire_date)
  - Child Class (Student):
    - Extends Parent & contains added properties as (department)

Type 4 (Joined Sub-Class Type) (Ex.)

```
<hibernate-mapping package="pojo">
 <class name="Person" table="person">
      <id name="id" column="id">
       <generator class="increment"/></id>
      <property name="firstName" column="first_name">
      <property name="lastName" column="last_name">
      <joined-subclass name="Student" table="student">
        <key column="id"/>
        <property name="department" column="department">
      </joined-subclass>
      <joined-subclass name="Teacher" table="teacher">
        <key column="id"/>
        <property name="hireDate" column="hire_date"
                  type="date">
      </joined-subclass>
 </class></hibernate-mapping>
```

Type 4 (Joined Sub-Class Type) (Ex.)

- Pros:
  - All other Cons Are solved.
- Note:
  - It’s recommended to add an attribute to parent table to specify which row represent what type of child.

Lab Assignment

- Perform these steps for each type:
  - Restore Inheritance Schema.
  - Create the mapping files to map these classes to domain models (java objects).
  - Perform CRUD operations on Person, Student, Teacher.

Lab Assignment

- Restore Bidding Schema.
- Create the mapping files to map these classes to domain models (java objects).
- Insert New Product & make bids on it and buy a product.
- Load a Product and update its values.

Hibernate
Fetching Strategies

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Lazy & Eager Loading

- Lazy loading means that all associated entities and collections aren't initialized if you load an entity object.
- This feature is enabled by default in Hibernate.
- It can be disabled by setting the attribute lazy = "false" in the mapping file.

Lazy & Eager Loading (Example)

```
<hibernate-mapping>
  <class name="dao.Seller" table="seller"
    catalog="biddingschema">
    ...
    <set name="products" table="product“
         lazy="true">
        <key>  <column name="seller_id" />  </key>
    <one-to-many class="dao.Product" />
    </set>
    ...
  </class>
</hibernate-mapping>
```

Seller.hbm.xml

Lazy & Eager Loading (Ex.)

- If the lazy attribute set to “false” when retrieving the Seller’s data

```
Seller s = (Seller)session.load(Seller.class,1);
```

- It will retrieve the list of Products as well
  - So calling the previous method will return the list of products whether in the persisted or detached mood.

Lazy & Eager Loading (Ex.)

- You can set lazy attribute to “Extra” when retrieving the Seller’s data

```
Seller s = (Seller)session.load(Seller.class,1);
```

- Individual elements of the collection are accessed from the database as needed. Hibernate tries not to fetch the whole collection into memory unless absolutely needed. It is suitable for large collections.

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Fetching Strategies

- Fetching:
  - It defines if and how an associated object or a collection should be loaded,
  - When the owning entity object is loaded, and when you access an associated object or collection.

Fetching Strategies (Example)

```
<hibernate-mapping>
  <class name="dao.Seller" table="seller"
    catalog="biddingschema">
    ...
    <set name="products" table="product“
          lazy="true" fetch="select">
        <key>  <column name="seller_id" />  </key>
    <one-to-many class="dao.Product" />
    </set>
    ...
  </class>
</hibernate-mapping>
```

Seller.hbm.xml

Fetching Strategies (Ex.)

- The main goal is to minimize the number of SQL statements, so that querying can be as efficient as possible.
- You do this by applying the best fetching strategy for each collection or association
- Fetching Strategies:
  - Join fetching
  - Select fetching
  - Batch fetching

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Fetching Strategies (Ex.)

- Join fetching:
  - Hibernate retrieves the associated instance or collection in the same SELECT, using an OUTER JOIN.

```
<class name="dao.Seller" table="seller">
    ...
    <set name="products" table="product“
         lazy="true" fetch="join">
        <key column="seller_id"/>
    <one-to-many class="dao.Product" />
    </set>
</class>
```

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Fetching Strategies (Ex.)

- Select fetching:
  - a second SELECT is used to retrieve the associated entity or collection.
  - Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you actually access the association.

```
<class name="dao.Seller" table="seller">
    <set name="products" table="product“
     lazy="true" fetch="select">
        <key column="seller_id"/>
    <one-to-many class="dao.Product" />
    </set>
</class>
```

Fetching Strategies (Ex.)

```
Seller s = (Seller)session.load(Seller.class,1);
```

- By default (the lazy attribute set to true) and if the object is detached (when the session is closed)
  - Calling **seller.getProducts()** will throw an Exception
- If the object is still in the persistence state, calling the previous method will hit the DB and retrieve the list of Products.

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Fetching Strategies (Ex.)

- Sub-Select fetching:
  - a second SELECT is used to retrieve the associated collections for all entities retrieved in a previous query.
  - Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you access the association.

```
<class name="dao.Seller" table="seller">
    <set name="products" table="product“
     lazy="true" fetch=“subselect">
        <key column="seller_id"/>
    <one-to-many class="dao.Product" />
     </set>
</class>
```

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Fetching Strategies (Ex.)

- Batch fetching :
  - an optimization strategy for select fetching - Hibernate retrieves a batch of entity instances or collections in a single SELECT.

```
<class name="dao.Seller" table="seller">
    ...
    <set name="products" table="product“
         lazy="true" batch-size="3">
        <key column="seller_id"/>
    <one-to-many class="dao.Product" />
    </set>
</class>
```

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Second Level Cache

- How did hibernate will cache the persisted & detached Objects?
- We have the option to tell Hibernate which caching implementation to use by specifying the name of a class that implements org.hibernate.cache.CacheProvider using the property hibernate.cache.provider_class.
- You can also implement your own and plug it in.

Second Level Cache (Ex.)

- Cache Mappings:
  - **<cache usage="read-write"/>**

    usage="read-write | nonstrict-read-write | read-only | transactional"

    region="RegionName"

    include="all | non-lazy" />
  - usage**: specifies the caching strategy.
  - region: specifies the name of the second level cache region
  - include: specifies that properties of the entity mapped with lazy="true" cannot be cached when attribute-level lazy fetching is enabled

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Second Level Cache (Ex.)

- Read-Only cache:
  - If your application needs to read, but not modify, instances of a persistent class, a read-only cache can be used.
  - This is the simplest and optimal performing strategy. It is even safe for use in a cluster.

```
<class name="dao.Seller" table="seller"
mutable="false">
    <cache usage="read-only"/>
</class>
```

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Second Level Cache (Ex.)

- Read-Write cache:
  - If the application needs to update data, a read-write cache might be appropriate.
  - This cache strategy should never be used if serializable transaction isolation level is required.

```
<class name="dao.Seller" table="seller">
    <cache usage="read-write"/>
    ...
    <set name="products" table="product">
     <cache usage="read-only"/>
     ...
    </set>
</class>
```

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Second Level Cache (Ex.)

- nonstrict-read-write cache:
  - If the application needs to update data, a read-write cache might be appropriate.
  - This cache strategy should never be used if serializable transaction
  - Cache is not locked at all.

```
<class name="dao.Seller" table="seller">
    <cache usage="nonstrict-read-write"/>
    ...
</class>
```

Hibernate Query (by HQL)

Query in Hibernate

- Hibernate offer query interfaces and methods on these interfaces
  - to prepare and execute arbitrary data retrieval operations.

- org.hibernate.Query & org.hibernate.Criteria define several methods for controlling execution of a query.

Query in Hibernate (Ex.)

- In Hibernate Query Language (HQL):
  - We don't deal with either table name or column name
  - We deal with them by Classes name and properities names.
  - `Query hqlQuery = session.createQuery("from Product");`
  - The previous example will select all instances from Product classes.

Parameter Binding

- There are two approaches to parameter binding:
  - Using named parameters.
  - Using positional.
- With named parameters, write the query as
  - String queryString = "from Product p where p.name like :name";
- The colon followed by a parameter name indicates a named parameter.
- Then, bind a value to the search parameter:
  - Query q = session.createQuery(queryString) .setString("name", productName);

Parameter Binding (Ex.)

- This code is cleaner, much safer, and performs better.
  - because a single compiled SQL statement can be reused if only bind parameters change.
  - String queryString = "from Product p where p.name like :name and p.manufacturingDate > :date";
  - Query q = session.createQuery(queryString)
    .setString("name", productName)
    .setDate("date", manufacturingDate);

Parameter Binding (Ex.)

- With Positional:
  - You can use positional parameters instead in Hibernate and Java Persistence:
  - String queryString = "from Product p where p.name like ? and p.manufacturingDate > ?";
  - Query q = session.createQuery(queryString)
    .setString(0, productName)
    .setDate(1, manufacturingDate);

Parameter Binding (Ex.)

- Hibernate supports setParameter() method for binding of parameters
- This method is a generic operation that can bind all types of arguments,
  - it only needs a little help for temporal type.

Parameter Binding (Ex.)

- A useful method is setEntity(),
  - which lets you bind a **persistent** entity.
  - Query q = session.createQuery("from User u where u.seller = :sellerKey").setEntity("sellerKey", sellerObject);

Executing a query

- Once you've created and prepared a Query.
- you're ready to execute it and retrieve the result into memory.
  - the list() method executes the query and returns the results as a java.util.List:
  - List result = myQuery.list();

Basic HQL queries

- We apply ***selection*** to name the data source, ***restriction*** to match records to the criteria, and ***projection*** to select the data you want returned from a query
- The simplest query in HQL is a selection of a single persistent class:
  - from Product p
  - This query generates the following SQL:
  - select p.name, p.manufacturingDate, ...

    from Product p

Restriction

- The WHERE clause is used to express a restriction in HQL
- This is WHERE clause that restricts the results to all User objects with the given email address:
  - String queryString = "from Product p where p.name like 'hp printer' ";
- Notice that
  - the constraint is expressed in terms of a **property, name, of the Product class,** and that you use an object-oriented notion for this.

Comparison Expressions

- HQL and JPA QL support the same basic comparison operators as SQL.
- Here are a few examples
  - from BuyerBidProduct bid where bid.amount between 200 and 300
  - from BuyerBidProduct bid where bid.amount > 300
  - from Product p where p.name in ('hp','dell')
- HQL and JPA QL provide an SQL-style IS [NOT] NULL operator:
  - from Product p where p.manufacturingDate is null
  - from Product p where p.manufacturingDate is not null

Comparison Expressions (Ex.)

- The LIKE operator allows wildcard searches, where the wildcard symbols are % and _, as in SQL:
  - from Product p where p.name like 'H%'
  - This expression restricts the result to products with a name starting with a capital H.
  - from Product p where p.name not like ‘%H%‘
  - This expression restricts the result to products with a name doesn’t include a capital H at any place.

- You can define an escape character if you want a literal percentage or underscore:
  - from Product p where p.name like '\%JETS%' escape ‘\’

Comparison Expressions (Ex.)

- HQL and JPA QL support **arithmetic expressions**:
  - from BuyerBidProduct bid where (bid.amount * 0.5)-100.0 > 0.0
- **Logical operators** (and parentheses for grouping) are used to combine expressions:
  - from Product p where p.name like 'H%' or p.manufacturingName like 'C%'
  - from Product p where (p.name like 'H%' or p.manufacturingName like 'C%') and p.quantity > 0

Comparison Expressions (Ex.)

- The precedence of operators, from top to bottom.
- The listed operators and their precedence are the same in HQL *and* JPA QL.

Operator	Description
.	Navigation path expression operator
+, -	Unary positive or negative signing (all unsigned numeric values are considered positive)
*, /	Regular multiplication and division of numeric values
+, -	Regular addition and subtraction of numeric values
=, <>, <, >, >=, <=, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL,	Binary comparison operators with SQL semantics
IS [NOT] EMPTY, [NOT] MEMBER [OF]	Binary operators for collections in HQL and JPA QL
NOT, AND, OR	Logical operators for ordering of expression evaluation

Comparison Expressions (Ex.)

- This query returns all Item instances that have an element in their bids collection.
  - from Product p

    where p.categories is not empty
- You can also express that you require a particular element to be present in a collection.
  - from Product p

    where ? member of p.categories "

Calling Functions

- An extremely powerful feature of HQL is
  - the ability to call SQL functions and user-defined functions in the WHERE clause.
  - from Product p where upper(p.name) like 'H%' or p.manufacturingName like 'C%‘
  - from Person p where concat(p.firstName,p.lastName) Like ‘A% M%’
  - Condition on the size of a collection:
  - from Product p where size(p.categories) > 6

Calling Functions (Ex.)

Table 14.2 Standardized JPA QL functions

Function	Applicability
`UPPER(s), LOWER(s)`	String values; returns a string value
`CONCAT(s1, s2)`	String values; returns a string value
`SUBSTRING(s, offset, length)`	String values (offset starts at 1); returns a string value
`TRIM( [[BOTH|LEADING|TRAILING] char [FROM]] s)`	Trims spaces on `BOTH` sides of `s` if no `char` or other specification is given; returns a string value
`LENGTH(s)`	String value; returns a numeric value
`LOCATE(search, s, offset)`	Searches for position of `ss` in `s` starting at `offset`; returns a numeric value
`ABS(n), SQRT(n), MOD(dividend, divisor)`	Numeric values; returns an absolute of same type as input, square root as `double`, and the remainder of a division as an `integer`
`SIZE(c)`	Collection expressions; returns an `integer`, or 0 if empty

Calling Functions (Ex.)

Table 14.3 Additional HQL functions

Function	Applicability
`BIT_LENGTH(s)`	Returns the number of bits in `s`
`CURRENT_DATE()`, `CURRENT_TIME()`, `CURRENT_TIMESTAMP()`	Returns the date and/or time of the database management system machine
`SECOND(d)`, `MINUTE(d)`, `HOUR(d)`, `DAY(d)`, `MONTH(d)`, `YEAR(d)`	Extracts the time and date from a temporal argument
`CAST(t as Type)`	Casts a given type `t` to a Hibernate `Type`
`INDEX(joinedCollection)`	Returns the index of joined collection element
`MINELEMENT(c)`, `MAXELEMENT(c)`, `MININDEX(c)`, `MAXINDEX(c)`, `ELEMENTS(c)`, `INDICES(c)`	Returns an element or index of indexed collections (maps, lists, arrays)
Registered in `org.hibernate.Dialect`	Extends HQL with other functions in a dialect

Ordering Query Results

- HQL and JPA QL provide an ORDER BY clause, similar to SQL.
  - This query returns all users, ordered by username:
  - from Person p order by p.firstName
- You specify ascending and descending order using asc or desc:
  - from Person p order by p.firstName desc
- You may order by multiple properties:
  - from Person p order by p.firstName asc, p.lastName asc

Projection

- The SELECT clause in HQL and JPA QL
  - allows you to specify exactly which objects or properties of objects you need in the query result.
  - Query q = session.createQuery("from Product p, BuyerBidProduct bid");

```
Iterator pairs = q.list().iterator();
while ( pairs.hasNext() ) {
        Object[] pair = (Object[]) pairs.next();
        Product product = (Product) pair[0];
        BuyerBidProduct bid = (BuyerBidProduct) pair[1];
}
```

Projection (Ex.)

- The following explicit SELECT clause also returns a collection of Object[ ]s:
  - select p.firstName, p.lastName from Person p where p.registrationDate > current_date()
- The Object[ ]s returned by this query contain
  - a String at index 0,
  - a String at index 1.

Aggregation Functions

- The aggregate functions that are used in HQL and standardized in JPA QL are
  - count(), min(), max(), sum() and avg().
- This query counts all the Items:
  - select count(p) from Product p
- The next variation of the query counts all Products which have a relation with Seller instance.
  - (null values are eliminated).
  - select count(p.seller) from Product p

Aggregation Functions (Ex.)

- This query calculates the total of all the successful payments:
  - select sum(buy.amount) from BuyerBuyProduct buy
- The next query returns the min and max payment value for payed products:
  - select min(buy.amount), max(buy.amount) from BuyerBuyProduct buy
  - The result is an ordered pair of Floats

Joining relations and associations

- Joining data is also the basic operation that enables you to fetch several associated objects and collections in a single query.

- We now show you how basic join operations work and how you use them to write a *dynamic fetching strategy*.

Joins in hibernate

- HQL provide three ways of expressing (inner and outer) joins:
  - An implicit association join
  - An ordinary join in FROM clause
  - A theta-style join in WHERE clause

Implicit association joins

- HQL support multipart property path expressions with a dot notation for two different purposes:
  - Querying components
  - Expressing implicit association joins

- The first use is straightforward:
  - from Seller s where s.user.email = 'mail@gmail.com'
  - You reference parts of the mapped component email with a dot notation.
  - No tables are joined in this query explicitly;

Implicit association joins (Ex.)

- The second usage of multipart path expressions is implicit association joining:
  - from Seller s where s.user.email like '%@gmail.com'
  - This results in an implicit join on the one-to-one associations from Seller to User
- Note
  - Implicit joins are always directed along many-to-one or one-to-one associations,
    - never through a collection-valued association (you can’t write product.categories.value).

Ordinary join in FROM clause

- If you want to retrieve Products, restrict the result to products that have name containes "hp" and also published by seller who's value contains "JETS":
  - from Product p join p.seller s where s.value like '%JETS%' and p.name like '%hp%'
- The query returns all combinations of associated Products and Sellers as ordered pairs:
- This query returns a List of Object[] arrays.
  - At index 0 is the Product, and at index 1 is the Seller.
  - A particular Seller may appear multiple times, once for each associated Product.
    - These duplicate Products are duplicate in-memory references, not duplicate instances!

Ordinary join in FROM clause (Ex.)

- To make outer join
  - from Product p left join p.seller s with s.value like '%JETS%' where p.name like '%hp%'
- The change is the additional join condition following the WITH keyword.
- If you place the s.value like '%JETS%' expression into the WHERE clause
  - you'd restrict the result to product instances that offered by Seller.

Theta-style join in WHERE clause

- In HQL, the theta-style syntax is useful when
  - your join condition isn't a foreign key relationship mapped to a class association.
- For example, suppose you store the User's name in log records
  - from Person p, LogRecord log where p.firstName = log.firstName
- The join condition here is a comparison of firstName,
  - present as an attribute in both classes

Comparing identifiers

- The identifier comparison in object-oriented terms,
  - is comparing object references.
- HQL and JPA QL support the following:
  - from Product p, Seller s where p.seller = s and s.value like '%JETS%'
- p.seller refers to the foreign key to the Seller table in the Product table (on the id column).
- And id refers to the primary key of the Seller table (on the id column).

Comparing identifiers (Ex.)

- The previous query uses a theta-style join and is equivalent to the much preferred
  - from Product p join p.seller s where s.value like '%JETS%' and p.name like '%hp%'

- On the other hand, the following theta-style join *can't be re-expressed as a FROM* clause join:
  - from Seller s, Buyer b
  - where s.products = b.products
  - s.products and b.products are both foreign keys of the Product table.

Grouping Aggregated results

- Just like in SQL,
  - any property or alias that appears in HQL outside of an aggregate function in the SELECT clause **must** also appear in the GROUP BY clause.
- The next query counts the number of Products with each Buyers' Name:
  - select b.name, count(b.products) from Buyer b group by b.name

Restricting groups with having

- The WHERE clause is used to perform the relational operation of restriction upon rows.
- The HAVING clause performs restriction upon groups.
- For example, the next query counts the number of Products with each Buyers' Name that contains "Ahmed" grouped by buyers' Name:
  - select b.name, count(b.products) from Buyer b group by b.name having b.name like '%Ahmed%'

Restricting groups with having

- The next query counts the number of Products with each Buyers' name grouped by buyers' name where the buyers' products greater than 4 products :
  - select b.name, count(b.products)
    from Buyer b
    group by b.name
    having count(b.products) > 4

Dynamic Instantiation

- If you define a class called AbstractBuyedProduct
  - with a constructor takes a String, an int, and a Date.
- the following query may be used:
  - select new AbstractBuyedProduct(p.name,

    size(p.buyerBuyProducts), p.manufacturingDate)

    from Product p

    where p.buyerBuyProducts is not null

Dynamic Instantiation (Ex.)

- The result of query,
  - each element is an instance of AbstractBuyedProduct, which contains summary data of a Product.
  - Product Name, counts of buying transactions on this product, Product Manufacturing Date.

- Note
  - you have to write a fully qualified classname here, with a package name.

Using Sub-selects

- A sub-select is a select query embedded in another query,
- HQL support
  - sub-queries in the WHERE clause
- The result of a sub-query may contain either a single row or multiple rows

Using Sub-selects (Ex.)

- The outer query returns all Sellers who offered products more than 2 products that it's name contains "hp":
  - from Seller s

    where (select count(p)

    from s.products p

    where p.name Like "%hp%") > 2

Using Sub-selects (Ex.)

- If a sub-query returns **multiple** rows,
  - it's combined with ***quantification***.
- HQL, define the following quantifiers:

  -- ALL --
  - The expression evaluates to true
    - if the comparison is true for all values in the result of the sub-query.
  - It evaluates to false
    - if a single value of the sub-query result fails the comparison test.

Using Sub-selects (Ex.)

-- ANY --

- The expression evaluates to true
  - if the comparison is true for some (any) value in the result of the subquery.
- it evaluates to false
  - If the subquery result is empty or no value satisfies the comparison.
- The keyword SOME is a synonym for ANY.

-- IN --

- This binary comparison operator can compare a list of values against the result of a subquery
- And evaluates to true
  - if all values are found in the result.

Using Sub-selects (Ex.) Examples

- All:
  - from Product p where p.seller = all(
    from Seller s where s.name Like “%Ahmed%”)
- Any:
  - from Product p where p.seller = any(
    from Seller s where s.name Like “%Ahmed%”)
- Some:
  - from Product p where p.seller = some(
    from Seller s where s.name Like “%Ahmed%”)
- So does this one:
  - from Product p where p.seller in (
    from Seller s where s.name Like “%Ahmed%”)

Hibernate Functions

- HQL supports a shortcut syntax for subqueries that operate on elements or indices of a collection.
- The following query uses the special HQL **elements()** function:
  - List result = session.createQuery("from Seller s where :givenProduct in elements(s.products)") .setEntity("givenProduct", product).list()

Hibernate Functions (Ex.)

- The previous query returns all students who involved in courses
- And it is equivalent to the following HQL (and valid JPA QL), where the subquery is more explicit:
  - List result = session.createQuery("from Seller s where :givenProduct in (select p from s.products p)") .setEntity("givenProduct", product).list();

Hibernate Functions (Ex.)

- HQL provides indices(), maxelement(), minelement(), maxindex(), minindex(), and size(),
  - each of which is equivalent to a certain correlated subquery against the passed collection.
- Refer to the Hibernate documentation for more information about these special functions;
  - **they're rarely used.**

Hibernate Query Language (HQL)

Syntax and features :

- HQL looks very much like SQL. But it is fully object-oriented, understanding notions like inheritance, polymorphism and association.
- **Case Sensitivity:**
  - Queries are case-insensitive, *except for names of Java classes and properties*. So SeLeCT is the same as sELEct is the same as SELECT but org.hibernate.eg.FOO is not org.hibernate.eg.Foo and foo.barSet is not foo.BARSET.
- **The from clause:**
  - *"from eg.Cat"* or *"from Cat"* and you can use *alias* as well "from Cat cat" or you can use multiple classes *"from Student, Subject"*
- **Associations and joins (The same like SQL):**

  - inner join
  - left outer join
  - right outer join
    - *"from Cat as cat inner join cat.mate as mate left outer join cat.kittens as kitten"*
  - If the join keyword is not used in the syntax , it is considered "inner join"

Hibernate Query Language (HQL) (Ex.)

- **The select clause:**
  - The select clause picks which objects and properties to return in the query result set.
    - *"select mate from Cat as cat inner join cat.mate as mate"*
      - This query returns multiple objects and/or properties as an array of type Object[],
    - *from Cat as cat inner join cat.mate as mate"*
      - This query returns multiple list of objects of type Cat
    - *"select new Family(mother, mate, offspr) from DomesticCat as mother"*
      - This query returns multiple list of objects of type Family
- **Aggregate functions:**
  - The supported aggregate functions are :
    - avg(...), sum(...), min(...), max(...)
    - count(*)
    - count(...), count(distinct ...), count(all...)
  - Example : *"select cat.weight + sum(kitten.weight)"*

Hibernate Query Language (HQL) (Ex.)

- **The 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:
    - *from Cat where name='Fritz'*
    - *from Cat as cat where cat.name='Fritz'*
    - from Cat cat where cat.mate.name is not null
  - The = operator may be used to compare not only properties, but also instances:
    - *from Cat cat, Cat rival where cat.mate = rival.mate*
    - *select cat, mate from Cat cat, Cat mate where cat.mate = mate*
- **Expressions:**
  - mathematical operators +, -, *, /
  - binary comparison operators =, >=, <=, <>, !=, like
  - logical operations and, or, not
  - Parentheses ( ), indicating grouping
  - Example *"from DomesticCat cat where cat.name between 'A' and 'B'"*
- **The Order By Clause :**
  - *from DomesticCat catorder by cat.name asc, cat.weight desc, cat.birthdate*

Hibernate Query Language (HQL) (Ex.)

- **The Group By Clause :**
  - *select foo.id, avg(name), max(name) from Foo foo join foo.names name group by foo.id*
- **Sub queries:**
  - *from Cat as fatcat where fatcat.weight > (select avg(cat.weight) from DomesticCat cat)*
- **DML Operations:**
  - The pseudo-syntax for UPDATE and DELETE statements is: (UPDATE | DELETE) FROM? EntityName (WHERE where_conditions)?
    - "update Customer set name = :newName where name = :oldName
    - delete Customer c where c.name = :oldName
  - The pseudo-syntax for INSERT statements is: INSERT INTO EntityName properties_list select_statement
  - Only the INSERT INTO ... SELECT ... form is supported; not the INSERT INTO ... VALUES ... form

List list = session.createQuery("hql query").list();

Hibernate Query (by Criteria)

Query in Hibernate (Ex.)

- To create a new Criteria instance,
  - call createCriteria(),
    - passing the class of the objects you want the query to return.
  - This is also called the *root entity of the criteria* query,

```
Criteria myCriteria = session.createCriteria(Product.class);
```

- Once you've created and prepared Criteria object,
  - you're ready to execute it and retrieve the result into memory.
  - the list() method executes the query and returns the results as a java.util.List:
  - List result = myCriteria.list();

Querying using Criteria

- Querying with criteria and example objects is the preferred solution when queries get more complex
- Example: - if you want to implement a search mask in your application,
  - with many check boxes, input fields, and switches the user can enable.
  - You must create a database query from the user's selection.
  - The traditional way to do this is
    - to create a query string through concatenation,
    - **or** maybe to write a *query builder that can construct the SQL query* string.

Querying using Criteria (Ex.)

- Criteria queries are more difficult to read if they get more complex:
  - a good reason to prefer them for dynamic and programmatic query generation,
- But it is prefer to use externalized HQL and JPA QL for predefined queries

Basic Criteria queries

- The simplest criteria query looks like this:
  - session.createCriteria(Person.class);
- It retrieves all persistent instances of the Person class.
  - This is called the ***root entity*** of the criteria query.
- The Criteria interface also supports ordering of results with the addOrder() method and the Order criterion:
  - session.createCriteria(Person.class)
    .addOrder(Order.asc("lastName"))
    .addOrder(Order.asc("firstName"));

Applying restrictions

- For a criteria query, you must construct **Criterion** object to express a constraint.
- The **Restrictions** class provides factory methods for built-in Criterion types.
- Let’s search for Person objects with a particular firstName.
  - Criterion name = Restrictions.eq(“firstName", "Ahmed");
  - Criteria criteria = session.createCriteria(Person.class);
  - criteria = criteria.add(name);
  - Person person = (Person)criteria.uniqueResult();

Comparison Expressions

- You can also name a property of a component with the usual dot notation:
  - session.createCriteria(Seller.class)

    .add(Restrictions.eq ("user.firstName", "Ahmed"));
- All regular SQL (and HQL) comparison operators are also available via the Restrictions class:
  - Restrictions.between("amount",100,200));
  - Restrictions.gt("amount", 100)

Comparison Expressions (Ex.)

- A search in list :
  - this query returns all Users who had email address with in the list:
  - String[] emails = { "JETS@hibernate.org", "Medhat@hibernate.org" };
  - session.createCriteria(User.class)

    .add(Restrictions.in("email", emails));
- A ternary logic operator is also available;
  - this query returns all Users with **no** email address:
  - session.createCriteria(User.class)

    .add(Restrictions.isNull("email"));

Comparison Expressions (Ex.)

- Query to find users who had an email address:
  - session.createCriteria(User.class)

    .add(Restrictions.isNotNull("email"));
- Test a collection with isEmpty(), isNotEmpty(), or its actual size:
  - session.createCriteria (Seller.class)

    .add(Restrictions.isEmpty("products"));
  - session.createCriteria (Seller.class)

    .add(Restrictions.sizeGt("products",3));

String Matching

- These two queries are equivalent:
  - session.createCriteria (User.class)

    .add(Restrictions.like("firstName", "Ah%"));
  - session.createCriteria (User.class)

    .add(Restrictions.like("firstName", "J", MatchMode.START));
- The allowed MatchModes are
  - START - END
  - ANYWHERE - EXACT.

String Matching (Ex.)

- To perform case-insensitive string matching
  - session.createCriteria(User.class)

    .add(Restrictions.eq("firstName", "JETS")
    .ignoreCase());

Combining Expressions

- If we add multiple Criterion instances to the one Criteria instance,
  - they’re applied conjunctively (using and):
  - session.createCriteria(User.class)

```
.add( Restrictions.like ("firstName", "A%"))
.add( Restrictions.like ("lastName", "M%"));
```

- To use disjunction (or),
  - there are two options.
    - The first is to use Restrictions.or() and Restrictions.and():

Logical Operators

- The first option is to use:
  - Restrictions.or() and Restrictions.and():
  - session.createCriteria(User.class)

```
.add(Restrictions.or(
        Restrictions.and (
                Restrictions.like("firstName", "A%"),
                Restrictions.like("lastName", "M%")
        ),
        Restrictions.in("email", emails)
));
```

Logical Operators (Ex.)

- The second option is to use
  - Restrictions.disjunction() and Restrictions.conjunction():
  - session.createCriteria(User.class)

```
.add(Restrictions.disjunction(
        Restrictions.conjunction(
            Restrictions.like("firstName", "A%"),
            Restrictions.like ("lastName", "M%")
        ),
        Restrictions.in("email", emails)
));
```

Sub-queries

- A subquery in a criteria query is a WHERE clause subselect
- The following subquery returns
  - the total number of products sold by a seller which have bids.
  - the outer query returns all sellers who have bids on their products more than 5 products.

Sub-queries (Ex.)

- DetachedCriteria subquery =

  DetachedCriteria.forClass (Product.class, "p");

- subquery = subquery.add(Restrictions.eq("p.seller.id", "s.id"))

  .add(Restrictions.isNotNull("p.buyerBidProducts"))

  .setProjection(Property.property("p.id").count());

- Criteria criteria = session.createCriteria(Seller.class, "s")

  .add(Subqueries.gt(5, subquery));

Joins

- There are two ways to express a join in the Criteria API;
  - hence there are two ways in which you can use aliases for restriction.
- The first is the createCriteria() method
  - This basically means you can nest calls to createCriteria():

Joins (Ex.)

- Criteria sellerCriteria = session.createCriteria(Seller.class);
- sellerCriteria = sellerCriteria.add(

  Restrictions.like ("name","M",
  MatchMode.ANYWHERE));
- Criteria productCriteria =
  sellerCriteria.createCriteria("products");
- productCriteria = productCriteria.add(

  Restrictions.like("name", "hp",
  MatchMode.ANYWHERE);
- List result = productCriteria.list();

Joins (Ex.)

- The creation of a **Criteria** for the courses **of the** Student results in
  - an inner join between the tables of the two classes.
- Note that you may call list() on either Criteria instance
  - without changing the query result.
- Nesting criteria works not only for collections (such as products),
  - but also for single-valued associations (such as user):

Joins (Ex.)

- This query returns all Sellers that are have an email on domain “hibernate.org”.
  - List result = session.createCriteria(Seller.class)

    .createCriteria("user")

    .add(Restrictions.like("email","%@hibernate.org")).list();

Joins (Ex.)

- The second way to express inner joins with the Criteria API
  - is to assign an alias to the joined entity:
  - session.createCriteria(Seller.class)

```
.createAlias("products", "p").add(
Restrictions.like("name", "%Ahmed%"))
.add( Restrictions.like("p.name", "%hp%");
```

Joins (Ex.)

- Properties of the joined entity must then be qualified by the alias assigned in createAlias() method,
  - such as p.name

- Properties of the root entity (Seller)
  - may be referred to without the qualifying alias,
  - OR with the alias "this":

Projection

- The Criteria API also supports projection,
  - You can select exactly which objects or properties of objects you need in the query result
  - and how you possibly want to aggregate and group results for a report.
- The following criteria query returns
  - only the identifier values of Products instances which are still opened for bids:
  - session.createCriteria(Product.class).add(

    Restrictions.lt ("finishDate", new Date()))

    .setProjection(Projections.id());

Projection (Ex.)

- session.createCriteria(User.class)

```
.setProjection(Projections.projectionList()
                .add(Projections.id())
                .add(Projections.property("firstName"))
                .add(Projections.property("lastName"))
);
```

- This query returns a List of Object[],
  - just like HQL

Aggregation and Grouping

- A straightforward method counts the number of rows in the result:
  - session.createCriteria(User.class)

    .setProjection(Projections.rowCount());

Aggregation and Grouping (Ex.)

- Query finds product name, manufacturingName, and the number of bids on each product:
  - session.createCriteria(Seller.class)

```
.createAlias ("products", "p").setProjection(
Projections.projectionList()
    .add(Property.forName("id").group())
    .add(Property.forName("p.name").group())
    .add(Property.forName("p.manufacturingName").group())
    .add(Property.forName("p.buyerBidProducts").count())
);
```

Aggregation and Grouping (Ex.)

- An alternative version that produces the same result is as follows:
  - session.createCriteria(Seller.class)

```
.createAlias ("products", "p").setProjection(
Projections.projectionList()
    .add(Property.groupProperty("id"))
    .add(Property.groupProperty("p.name"))
    .add(Property.groupProperty("p.manufacturingName"))
    .add(Property.count("p.buyerBidProducts"))
);
```

Aggregation and Grouping (Ex.)

- For ordering of the result:
  - session.createCriteria(Seller.class)

```
.createAlias ("products", "p").setProjection(
Projections.projectionList()
    .add(Property.groupProperty("id"))
    .add(Property.groupProperty("p.name"))
    .add(Property.groupProperty("p.manufacturingName"))
    .add(Property.count("p.buyerBidProducts"))
).addOrder(Order.asc("p.name"));
```

Hibernate Query (by Criteria) *** Using Example ***

Query by example

- It’s common for criteria queries to be built programmatically by
  - combining several optional criterions depending on user input.
    - For example, a system administrator may wish to search for users by any combination of first name or last name and retrieve the result ordered by username.
- Using HQL or JPA QL, you can build the query using string manipulations:

Example on HQL

```
public List findUsers(String firstname,
                      String lastname) {

    StringBuffer queryString = new StringBuffer();
    boolean conditionFound = false;

    if (firstname != null) {
        queryString.append("lower(u.firstname) like :firstname ");
        conditionFound=true;
    }
    if (lastname != null) {
        if (conditionFound) queryString.append("and ");
        queryString.append("lower(u.lastname) like :lastname ");
        conditionFound=true;
    }

    String fromClause = conditionFound ?
                        "from User u where " :
                        "from User u ";

    queryString.insert(0, fromClause).append("order by u.username");

    Query query = getSession()
                      .createQuery( queryString.toString() );

    if (firstname != null)
        query.setString( "firstName",
                         '%' + firstname.toLowerCase() + '%' );
    if (lastname != null)
        query.setString( "lastName",
                         '%' + lastname.toLowerCase() + '%' );

    return query.list();
}
```

Example on Criteria

- This code is pretty tedious and noisy.
- Lets make it with Criteria API
  - public List findUsers (String firstName, String lastName) {

```
Criteria criteria = getSession().createCriteria(User.class);
if(firstName != null){
        criteria = criteria.add(
                Restrictions.ilike("firstName",
                firstName, MatchMode.ANYWHERE));
}
```

Example on Criteria (Ex.)

```
        if (lastName != null) {
                criteria = criteria.add(
                        Restrictions.ilike("lastName",
                lastName, MatchMode.ANYWHERE));
        }
        criteria = criteria.addOrder( Order.asc ("userName") );
        return criteria.list();
}
```

Example on Criteria (Ex.)

- This code is much shorter.
- the ilike() operator performs a case-insensitive match.
- For search screens with many optional search criteria, there is an even better way.
  - As you add new search criteria, the parameter list of findUsers() grows.
- It would be better to capture the searchable properties as an object.

Query by example

- With Query by example (QBE)
  - You provide an instance of the queried class with some properties initialized,
    - and the query returns all persistent instances with matching property values.
- Hibernate implements QBE as part of the Criteria query API:

Query by example (Ex.)

```
public List findUsersByExample (User user){
   Example exampleUser = Example.create(user)
         .ignoreCase().enableLike(MatchMode.ANYWHERE)
         .excludeProperty("password");
   return getSession().createCriteria(User.class)
                  .add(exampleUser).list();
}
```

Query by example (Ex.)

- The call to create() returns
  - a new instance of Example for the given instance of User.
- The ignoreCase() method puts
  - the example query into a case-insensitive mode for all string-valued properties.
- The call to enableLike() specifies
  - that the SQL like operator should be used for all string-valued properties, and specifies a MatchMode.

Query by example (Ex.)

- The excludeProperty() method
  - exclude particular properties from the search
- By default, all value-typed properties, excluding the identifier property, are used in the comparison.
- The values that contains null are excluded from search quary

Query by example (Ex.)

- The Example is just an ordinary Criterion.
  - You can mix and match query by example with query by criteria.

- The following query restrict the search results.

Query by example (Ex.)

```
public List findUsersByExample (User user){
   Example exampleUser = Example.create(user)
        .ignoreCase().enableLike(MatchMode.ANYWHERE)
        .excludeProperty("password");
   return getSession().createCriteria(User.class)
                .add(exampleUser).createCriteria ("seller")
                .add(Restrictions.isNull("products"))
                .list();
}
```

Against ORM

- Can be slow with large Inserts & Updates (batch processing)
- Initially steep learning curve
- Sometimes hard to track and optimize

Hibernate Annotation

Hibernate Model

- Hibernate Core
  - offers native API's & object/relational mapping with XML metadata
- Hibernate Annotations
  - offers JDK 5.0 code annotations as a replacement or in addition to XML metadata
- Hibernate EntityManager
  - involves standard JPA for Java SE and Java EE

Annotations

- is a special form of syntactic metadata that can be added to Java source code.
  - Classes.
  - Methods.
  - Variables.
  - Parameters.
  - Packages.
  - Annotations
- Java annotations can be reflective:
  - They can be embedded in class files generated by the compiler and may be retained by the Java VM or other framework to be made retrievable at run-time

Annotations (Ex.)

- Annotation Syntax
  - @AnnotationName
  - @AnnotationName ("value")
  - @AnnotationName (element1 = "value1", element2 = "value2")

Built-In Annotations

- Annotations applied to java code:
  - @Override
    - Checks that the function is an override.
    - Causes a compile error if the function is not found in one of the parent classes.
  - @Deprecated
    - Marks the function as obsolete.
    - Causes a compile warning if the function is used.
  - @SuppressWarnings
    - Instructs the compiler to suppress the compile time warnings specified in the annotation parameters
  - Annotations applied to other annotations:

Built-In Annotations (Ex.)

- Annotations applied to other annotations:
  - @Retention
    - Specifies how the marked annotation is stored.
    - Whether in code only, compiled into the class, or available at runtime through reflection.
  - @Documented
    - Marks another annotation for inclusion in the documentation.
  - @Target
    - Marks another annotation to restrict what kind of java elements the annotation may be applied to
  - @Inherited
    - Marks another annotation to inherit features from a parent annotation

Hibernate Annotations

- Available from Hibernate 3.5
- Basic annotations that implement the JPA standard:

• @Entity	Declares this an entity bean
• @Id • @EmbeddedId • @GeneratedValue	Identity
• @Table • @Column	Database Schema Attributes
• @OneToOne • @ManyToOne • @OneToMany	Relationship mappings

Hibernate Annotations (Ex.)

- Hibernate extension annotations:
  - Contained in org.hibernate.annotations package
    - @org.hibernate.annotations.Entity
    - @org.hibernate.annotations.Table
    - @BatchSize
    - @Where
    - @Check

Hibernate Techniques

- Old Technique:
  - Hibernate configuration file.
  - Hibernate Mapping Files.
  - Hibernate Mapping Classes (Entities).
  - Hello with Login Example.
- New Technique (+ Annotation):
  - Hibernate configuration file.
  - Hibernate Mapping Classes (Entities).
  - Hello with Login Example.

New Technique Type(1) at *.cfg

```
<hibernate-configuration>
  <session-factory>
        <property name = "hibernate.connection.driver_class">
          org.gjt.mm.mysql.Driver                          </property>
        <property name = "hibernate.connection.url">
          jdbc:mysql://localhost:3306/helloworlddb
        </property>
        <property name = "hibernate.connection.username">
          root                                             </property>
        <property name ="hibernate.connection.password">
          root                                             </property>
        <property name ="hibernate.dialect">
          org.hibernate.dialect.MySQLInnoDBDialect
        </property>
  <mapping resource="dao/Person.hbm.xml" class="dao.Person"/>
  </session-factory>
</hibernate-configuration>
```

New Technique Type(1) at Main class

```
public class Test {
   public static void main(String[] args) {
         SessionFactory sessionFactory =
                        new AnnotationConfiguration().configure()
                        .buildSessionFactory();
      Session session = sessionFactory.openSession();

          Account account = new Account();
          account.setName("Medhat");
          account.setPhone("0235355637");
          account.setBirthday(new Date());
          account.setEmail("ahyousif@mcit.gov.eg");

          session.beginTransaction();
          session.persist(account);
          session.getTransaction().commit();
          System.out.println("Insertion Done");
   }
}
```

Test.java

New Technique Type(2) at *.cfg

```
<hibernate-configuration>
  <session-factory>
      <property name = "hibernate.connection.driver_class">
        org.gjt.mm.mysql.Driver                    </property>
      <property name = "hibernate.connection.url">
        jdbc:mysql://localhost:3306/helloworlddb
      </property>
      <property name = "hibernate.connection.username">
        root                                       </property>
      <property name ="hibernate.connection.password">
        root                                       </property>
      <property name ="hibernate.dialect">
        org.hibernate.dialect.MySQLInnoDBDialect
      </property>
  <mapping resource="dao/Person.hbm.xml" class="dao.Person"/>
  </session-factory>
</hibernate-configuration>
```

New Technique Type(2) at Main class

```
public class Test {
  public static void main(String[] args) {
        SessionFactory sessionFactory =
                        new AnnotationConfiguration().configure()
          .addPackage("dao").addAnnotatedClass(Person.class)
          .buildSessionFactory();
        Session session = sessionFactory.openSession();

        Account account = new Account();
        account.setName("Medhat");
        account.setPhone("0235355637");
        account.setBirthday(new Date());
        account.setEmail("ahyousif@mcit.gov.eg");

        session.beginTransaction();
        session.persist(account);
        session.getTransaction().commit();
        System.out.println("Insertion Done");
 }}
```

Test.java

New Technique Type (1 & 2) Entity

```
@Entity
@Table(name = "account")
public class Account{
   private long id;
   private String userName;

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "id")
    public long getId() {          return id;      }

    @Column(name = "user_name", nullable = false)
    public String getUserName() {      return userName;     }
}
```

Account.java

Built-In Annotations

- With in hibernate version 3.5:
  - We use class AnnotationConfiguration to initiate the configuration of hibernate factory.
- With in hibernate version 4.*:
  - Class AnnotationConfiguration being deprecated and they enhance configuration class to use xml mapping & annotation maping.

Mapping Definitions Demo(s)

Advanced Topics (Interceptor)

Interceptor

- **There are two types of Interceptor based on their scope:**
  - Session-scoped Interceptor:
    - This interceptor is determined when SessionFactory.openSession() method is called that accepts an interceptor.
    - Entire persistent objects, will be affected associated within that particular session.
  - SessionFactory-scoped Interceptor:
    - This interceptor is specified with the Configuration object before the creation of SessionFactory.
    - This interceptor is used as a global interceptor and is applicable to all the sessions that are opened from same SessionFactory

Interceptor

- **Interceptor Declaration:**

```
public class AccountInterceptor extends EmptyInterceptor{
   public Void onDelete(...){
   }
   public boolean onFlush(...){
   }
   public boolean onLoad(...){
   }
   public boolean onSave(...){
   }
   public Void afterTransactionCompletion(...){
   }
}
```

AccountInterceptor.java

Interceptor

- Session-scoped Interceptor:

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

- SessionFactory-scoped Interceptor:

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

Advanced Topics (Event Listener)

Event Listener

- Available from Hibernate 3.0
- **Event Listener Declaration:**

```
public class AccountListener extends
DefaultSaveOrUpdateEventListener {
   public Void onSaveOrUpdate(SaveOrUpdateEvent event)
         throws HibernateException {
         if (event.getObject() instanceof MyAccount){
         }
   }
}
```

AccountListener.java

Event Listener

- **Event Listener Usage:**

```
Configuration cfg = new Configuration();
cfg.setListener("save",new AccountListener());
cfg.configure();
SessionFactory factory = cfg.buildSessionFactory();
Session session =  factory.openSession();
```

Advanced Topics (Naming Strategy)

Naming Strategy

- Available from Hibernate 3.1
- **Use Interface/Classe(s):**
  - **NamingStrategy (Interface)**
  - **DefaultNamingStrategy (Class)**
  - **ImprovedNamingStrategy (Class)**
  - If you want implemented a custom Naming Strategy.

```
<hibernate-mapping>
  <class name="dao.MyAccount" table="account">
      ...
  </class>
</hibernate-mapping>
```

Naming Strategy

- **Naming Strategy Declaration:**

```
public class MyNamingStrategy extends
DefaultNamingStrategy {
   public String classToTableName(String className){
         if (className.equals("dao.MyAccount"))
            return ("account");
         else
            return ("EMPTY");
   }
}
```

MyNamingStrategy.java

Naming Strategy

– **Naming Strategy Usage:**

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