Lightweight J2EE Framework

Struts, spring, hibernate

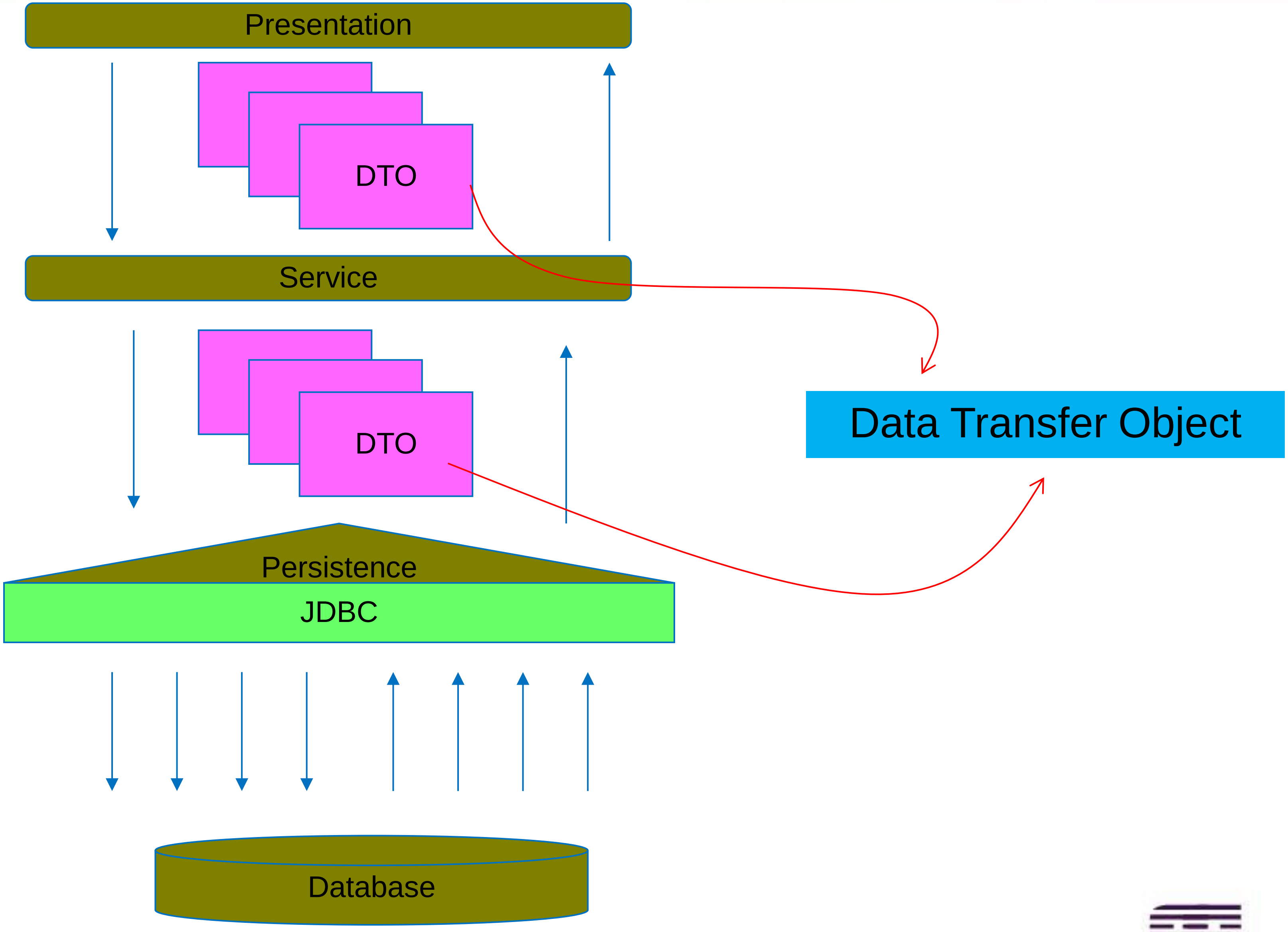
Software System Design
Zhu Hongjun

Session 4: Hibernate DAO

- Refresher in Enterprise Application Architectures
- Traditional Persistence and Hibernate
- Basic O/R Mapping
- Association and Collection Mapping
- Component and Inheritance Mapping

Refresher in enterprise application architectures

- Enterprise Application Architectures
  - N-Tier Architecture
    - Common Tiers
      - Presentation
        - Responsible for displaying data only, no business logic
      - Service
        - Responsible for business logic
      - Persistence
        - Responsible for retrieving/storing data



Presentation

DTO

Service

DTO

Data Transfer Object

Persistence

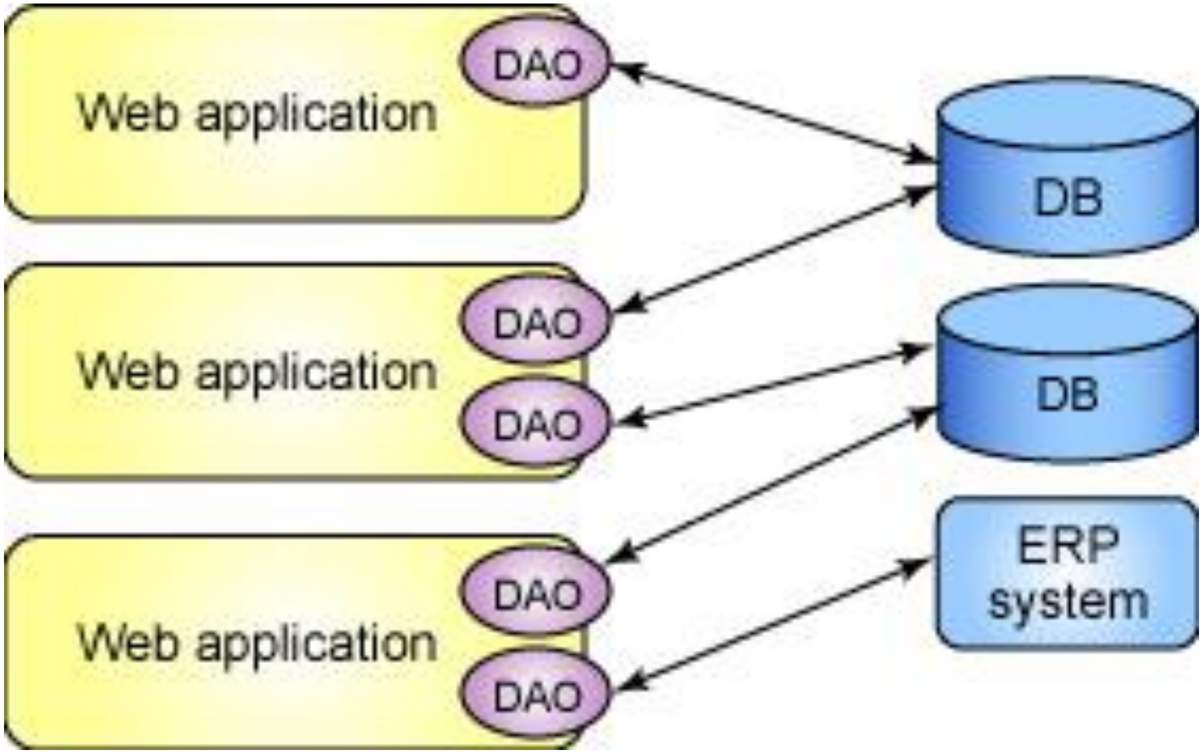
JDBC

Database

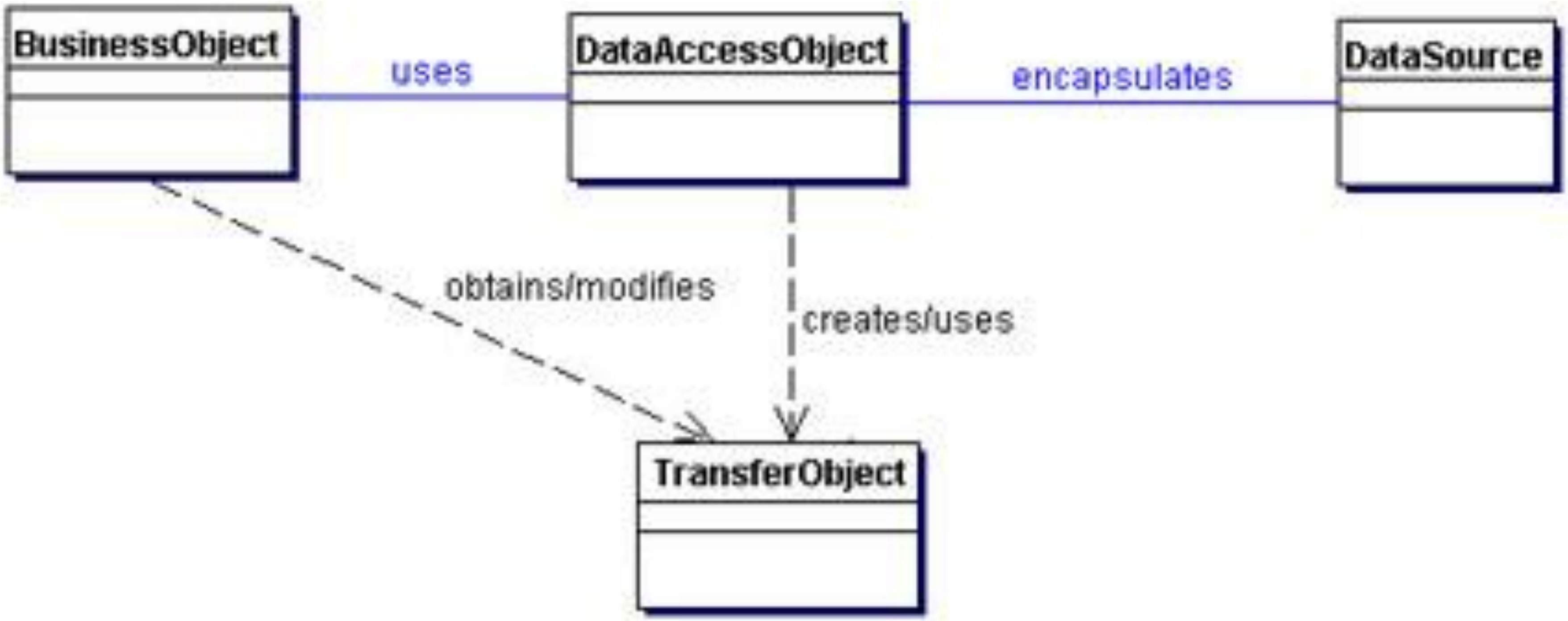
Refresher in enterprise application architectures

- Enterprise Application Architectures (cont.)
  - DAO Design Pattern
    - Data Access Object
      - Abstract CRUD (Create, Retrieve, Update, Delete) operations
    - Benefits
      - Allows different storage implementations to be ‘plugged in’ with minimal impact to the rest of the system
      - Decouples persistence layer
      - Encourages and supports code reuse

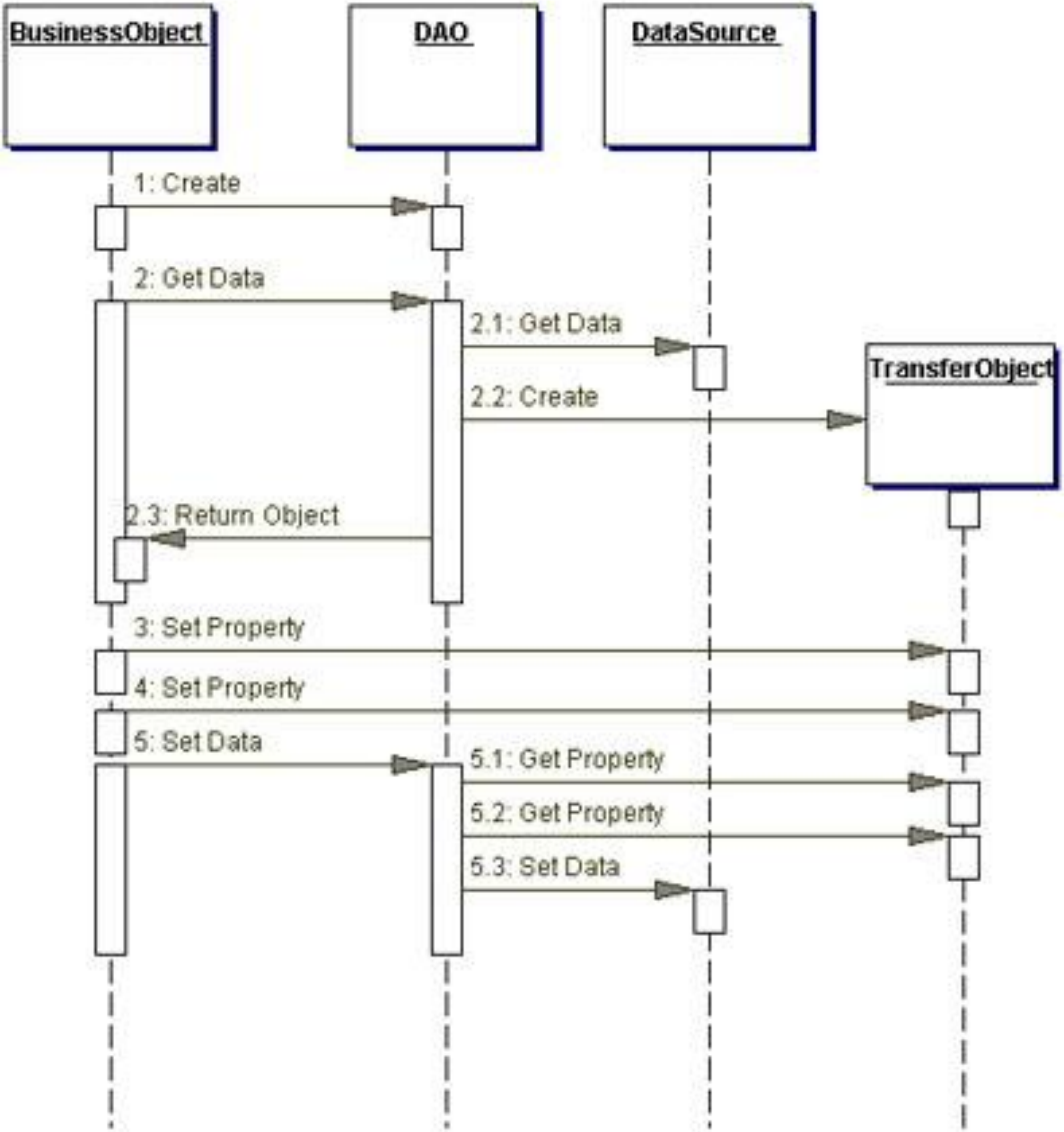
DAO pattern architecture



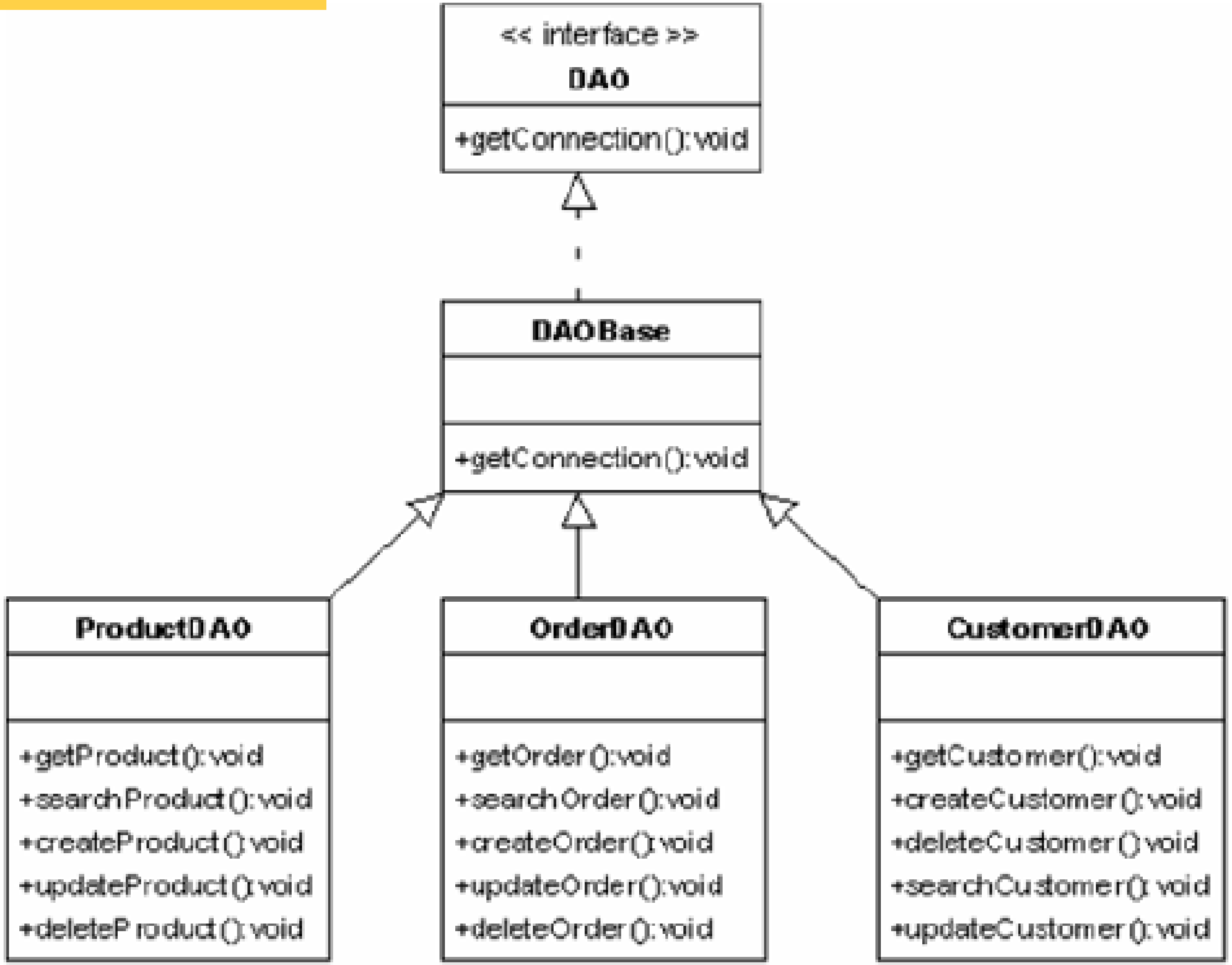
The simplest implementation of DAO pattern



Data Access Object sequence diagram



A DAO pattern with a DAO interface



<< interface >>
DAO

+getConnection():void

DAOBase

+getConnection():void

ProductDAO

+getProduct():void
+searchProduct():void
+createProduct():void
+updateProduct():void
+deleteProduct():void

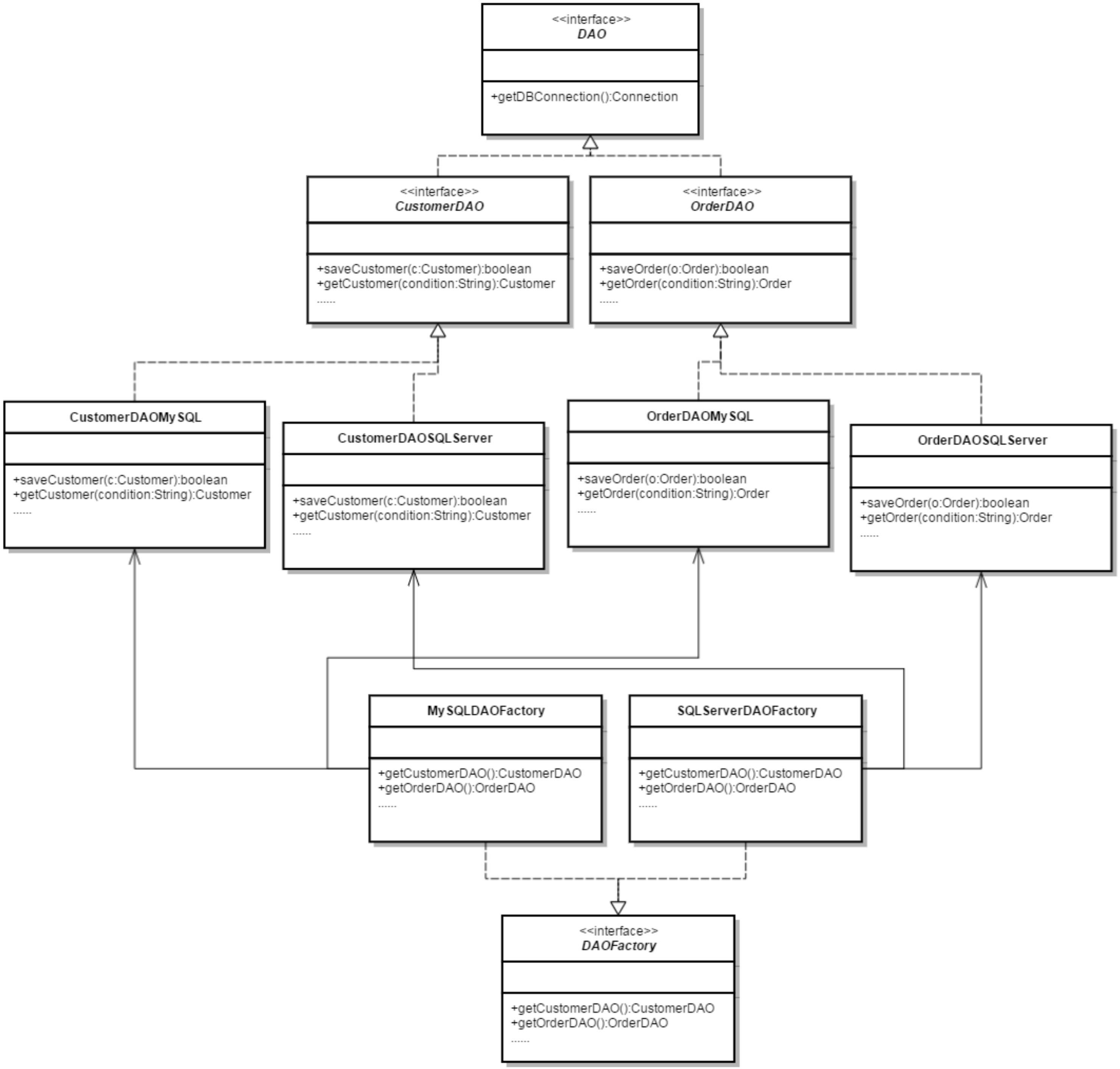
OrderDAO

+getOrder():void
+searchOrder():void
+createOrder():void
+updateOrder():void
+deleteOrder():void

CustomerDAO

+getCustomer():void
+createCustomer():void
+deleteCustomer():void
+searchCustomer():void
+updateCustomer():void

The DAO Pattern with the Abstract Factory Pattern



<<interface>>
DAO

+getDBConnection():Connection

<<interface>>
CustomerDAO

+saveCustomer(c:Customer):boolean
+getCustomer(condition:String):Customer
......

<<interface>>
OrderDAO

+saveOrder(o:Order):boolean
+getOrder(condition:String):Order
......

CustomerDAOMySQL

+saveCustomer(c:Customer):boolean
+getCustomer(condition:String):Customer
......

CustomerDAOSQLServer

+saveCustomer(c:Customer):boolean
+getCustomer(condition:String):Customer
......

OrderDAOMySQL

+saveOrder(o:Order):boolean
+getOrder(condition:String):Order
......

OrderDAOSQLServer

+saveOrder(o:Order):boolean
+getOrder(condition:String):Order
......

MySQLDAOFactory

+getCustomerDAO():CustomerDAO
+getOrderDAO():OrderDAO
......

SQLServerDAOFactory

+getCustomerDAO():CustomerDAO
+getOrderDAO():OrderDAO
......

<<interface>>
DAOFactory

+getCustomerDAO():CustomerDAO
+getOrderDAO():OrderDAO
......

Refresher in enterprise application architectures

- Enterprise Application Architectures (cont.)
  - Implementing Business Logic
    - Service Layer
      - Thin domain model
      - Procedural service layer
    - Domain Model/Domain Object
      - Thin service layer
      - Business logic primarily in the domain/business objects
    - Combination of the above two

Refresher in enterprise application architectures

- Enterprise Application Architectures (cont.)
  - Implementing Business Logic (cont.)
    - Design Approaches
      - D1: Service layer contains all business logic (no real domain model)
      - D2: Complex OO domain model/thin service layer
      - D3: Service layer contains use case logic that operates over thin or moderately complex domain model

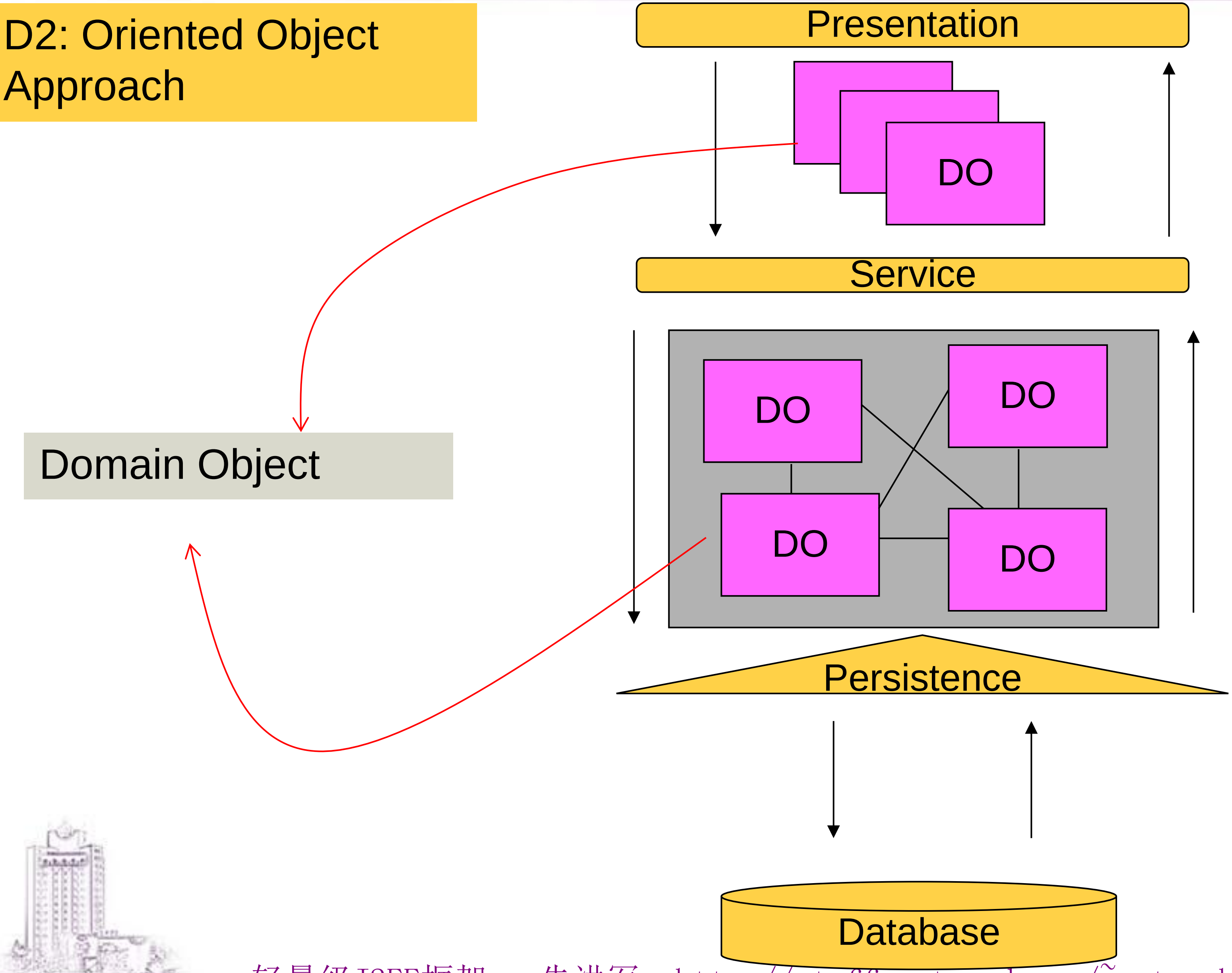
Refresher in enterprise application architectures

- Enterprise Application Architectures (cont.)
  - Implementing Business Logic (cont.)
    - D1
      - Service layer communicates directly to data access layer
        - No object model
        - Data access layer returns data transfer objects (DTOs) to service layer
      - Leverages commonly understood core technologies
        - JDBC, JavaBeans
      - Requires more low level code to persist transfer objects to the data store

Refresher in enterprise application architectures

- Enterprise Application Architectures (cont.)
  - Implementing Business Logic (cont.)
    - D2
      - Complex OO domain model/thin service layer
        - Rich object model utilizing standard design patterns, delegation, inheritance, etc.
        - Distinct API to domain model
      - May result in more maintainable code but updates are harder
        - What objects have been modified and need to be saved in the database
      - Need complex Data Mapper/Data Store since domain model and database schema are likely different
        - TopLink, JDO, Hibernate

D2: Oriented Object Approach



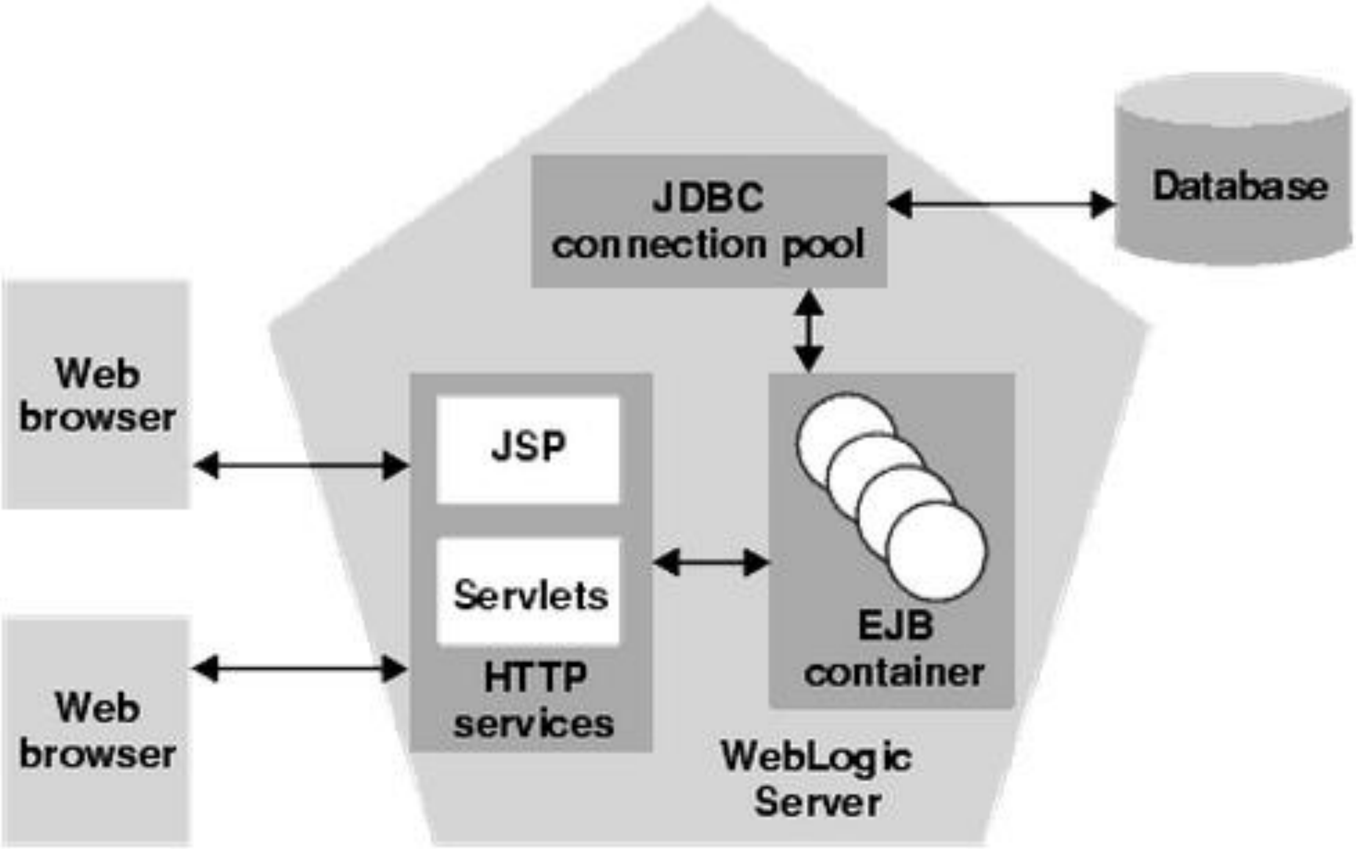
Refresher in enterprise application architectures

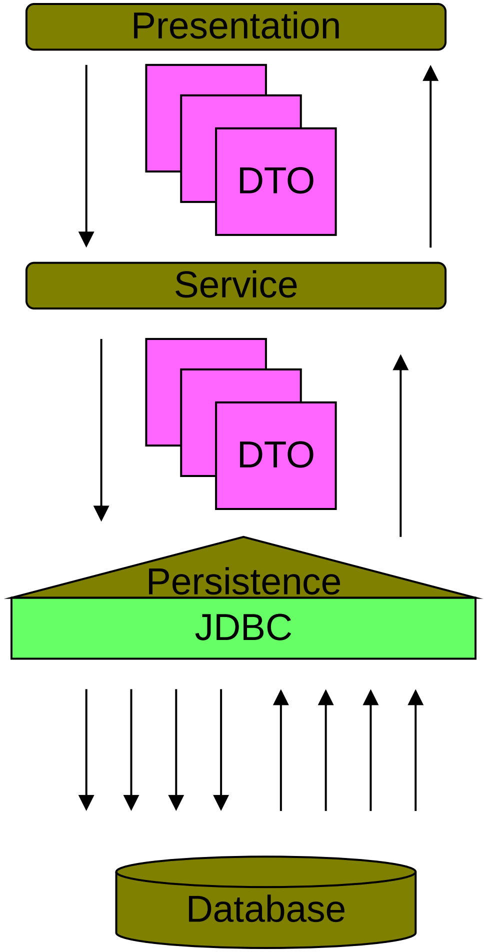
- Enterprise Application Architectures (cont.)
  - Implementing Business Logic (cont.)
    - D3
      - Object model can be basic to moderately complex
        - Simple model is just used as a data access/ORM layer
        - Model can take on business logic
      - Uses advantages of both extremes
      - Difficult to remain consistent within the same application

Traditional persistence and Hibernate

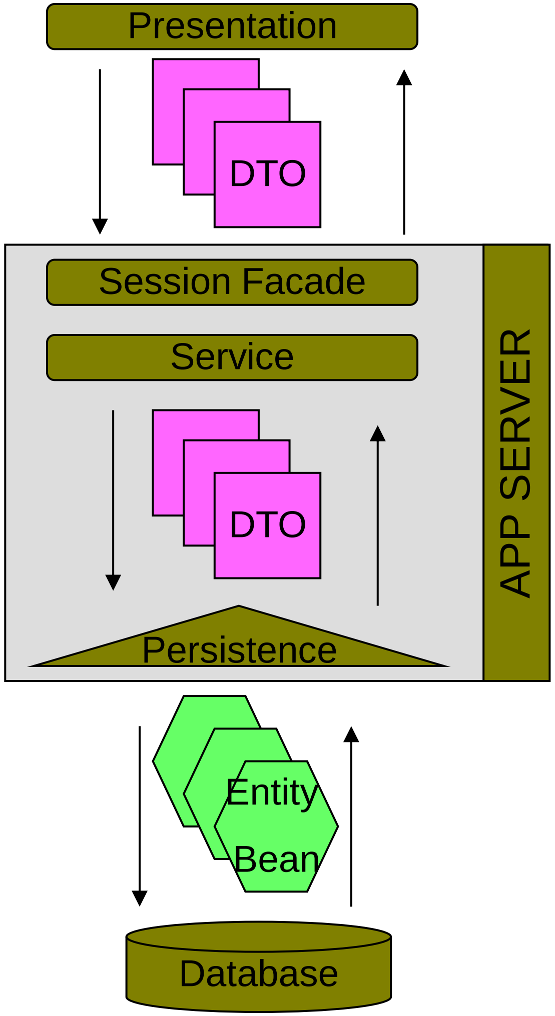
- Persistence with JDBC
  - Basic Steps to JDBC Operations
    - Load driver or obtain datasource
    - Establish connection using a JDBC URL
    - Create statement
    - Execute statement
    - Optionally, process results in result set
    - Close database resources
    - Optionally, commit/rollback transaction
- Persistence with EJB
    - Create your EJB
    - Setup deployment descriptors
    - In code, look up the EJB Home Interface
    - Create an instance of the EJB off the Home Interface, using attributes passed in through the method call

Web-based component application model in WebLogic server





Persistence with JDBC

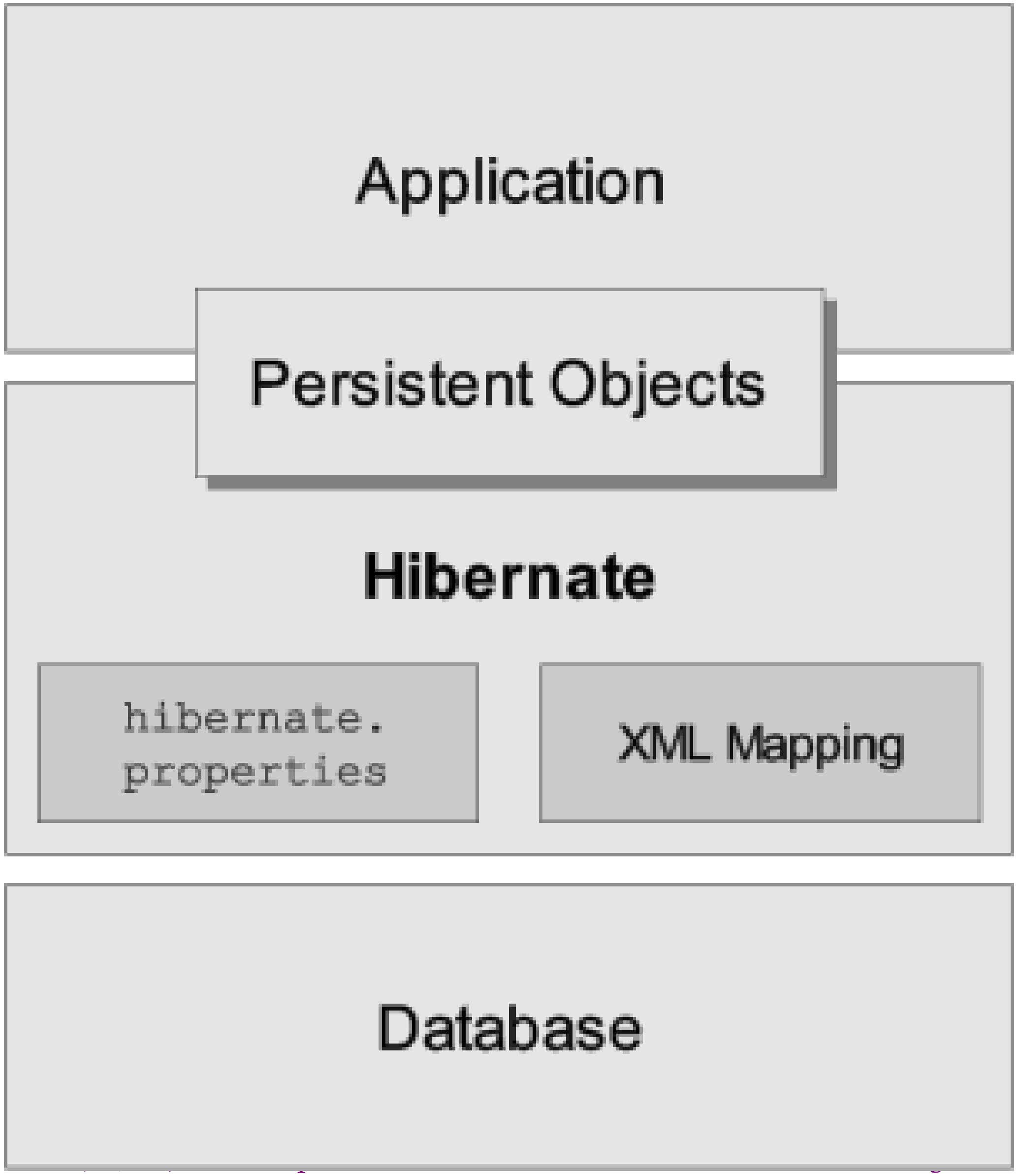


Persistence with EJB 2.x

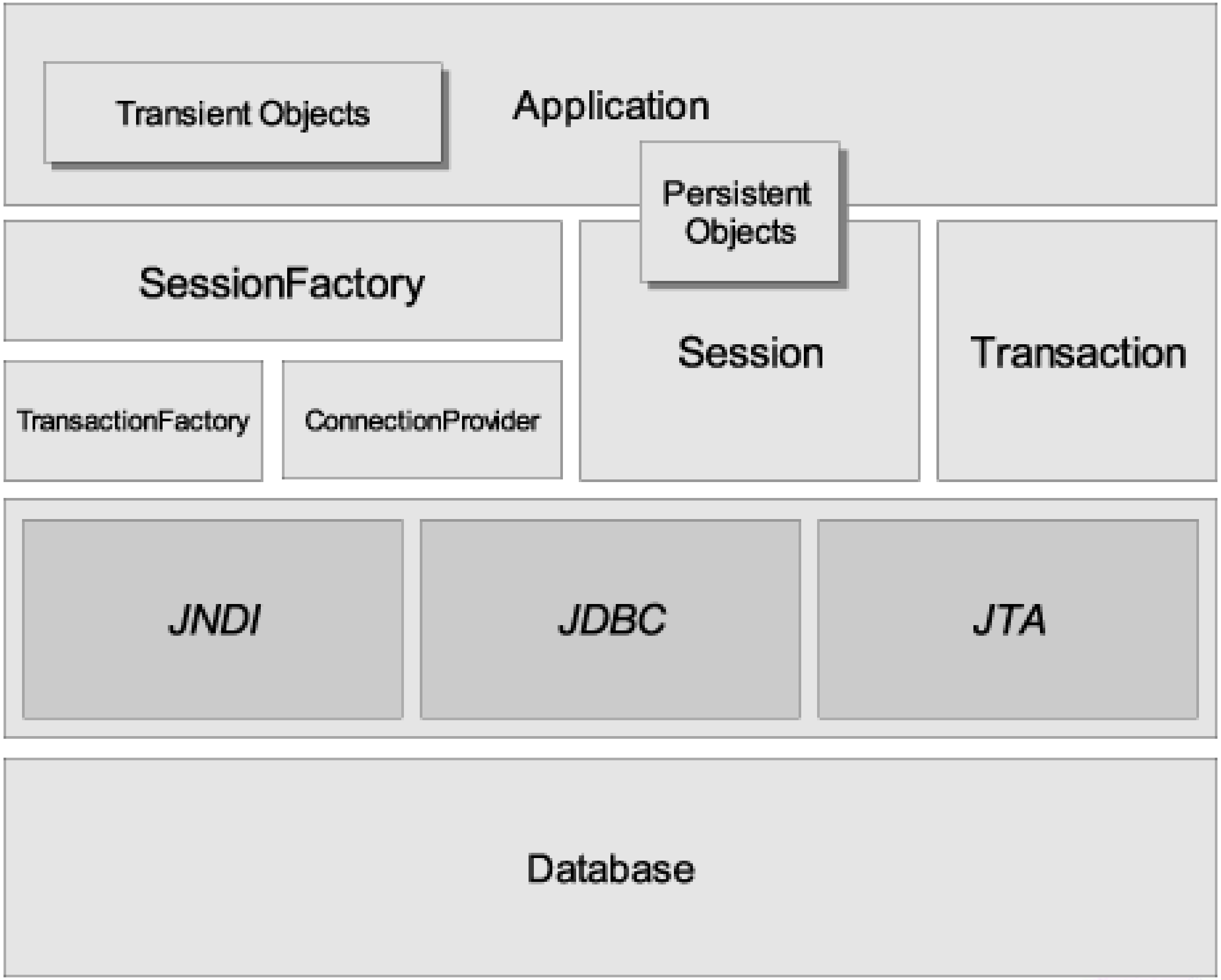
Traditional persistence and Hibernate

- Hibernate
  - Hibernate is a high-performance Object/Relational persistence and query service
  - It is most useful with object-oriented domain models and business logic in the Java-based middle-tier
  - It can significantly reduce development time otherwise spent with manual data handling in SQL and JDBC

Hibernate Architecture



Comprehensive Architecture of Hibernate

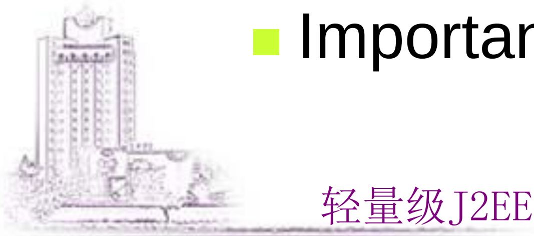


Traditional persistence and Hibernate

- Hibernate (cont.)
  - Basic APIs
    - SessionFactory
    - Session
    - Persistent Objects and Collections
    - Transient and Detached Objects and Collections
    - Transaction
    - ConnectionProvider
    - TransactionFactory
    - Extension Interfaces

Traditional persistence and Hibernate

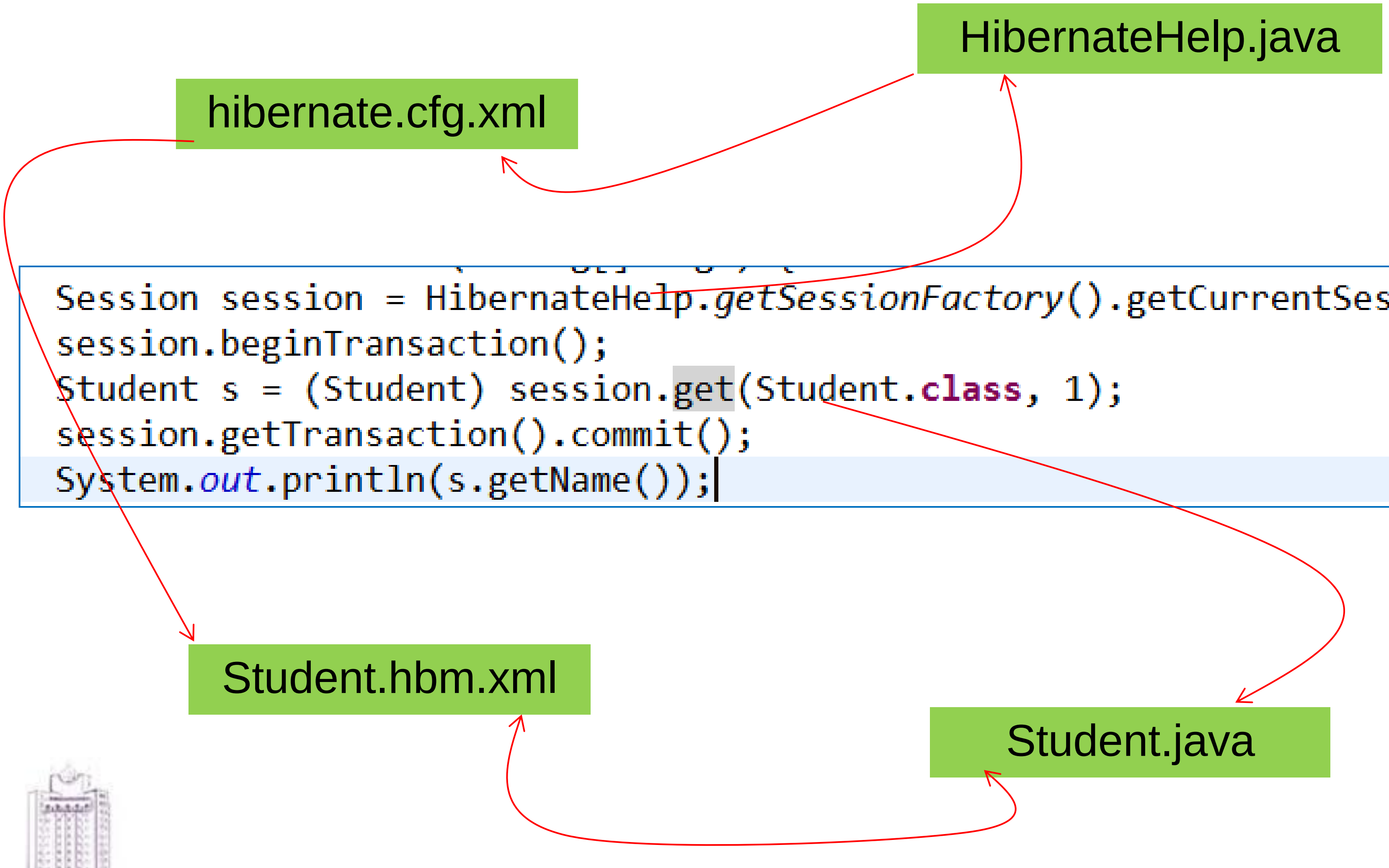
- Hibernate (cont.)
  - Goals
    - Prevent leakage of concerns
      - Domain model should only be concerned about modeling the business process, not persistence, transaction management and authorization
    - Transparent and automated persistence
    - Metadata in XML
    - Reduction in LOC
    - Importance of domain object model



Traditional persistence and Hibernate

- Hibernate (cont.)
  - Installation
    - Step 1
      - Download hibernate from http://www.hibernate.org/
    - Step 2
      - Copy hibernate core jars and jdbc jar of the DB to web-inf/lib directory of your project
    - Step 3
      - Define entity bean and mapping file
    - Step 4
      - Create hibernate configuration file

Use Hibernate Demo



HibernateHelp.java

hibernate.cfg.xml

```
Session session = HibernateHelp.getSessionFactory().getCurrentSession();
session.beginTransaction();
Student s = (Student) session.get(Student.class, 1);
session.getTransaction().commit();
System.out.println(s.getName());
```

Student.hbm.xml

Student.java

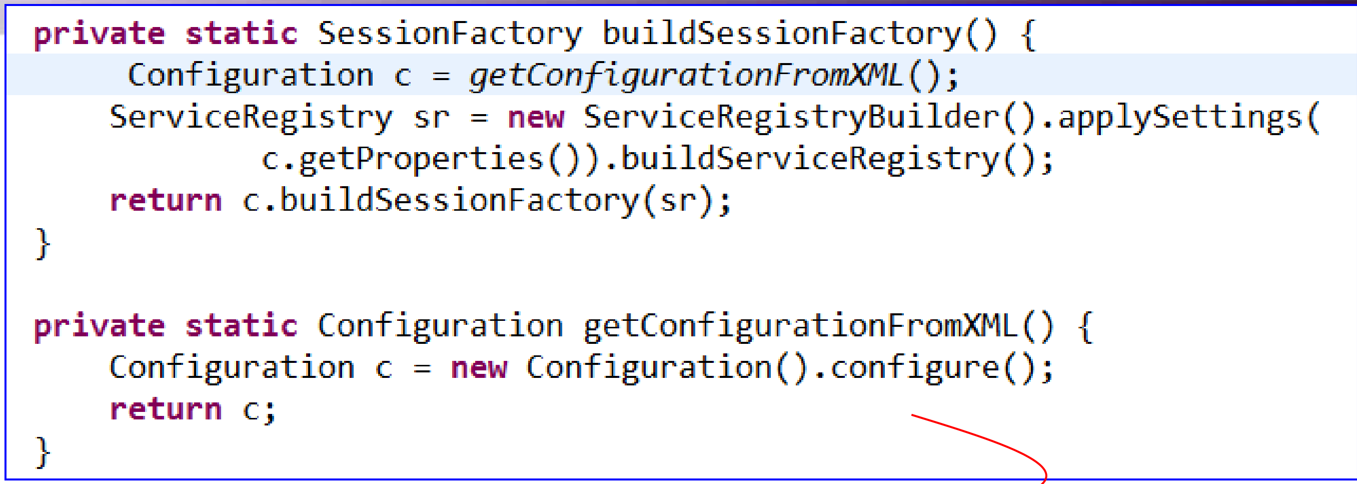
Traditional persistence and Hibernate

- Hibernate (cont.)
  - Connecting
    - Hibernate obtains JDBC connections as needed though the org.hibernate.service.jdbc.connections.spi.ConnectionProvider interface which is a service contract
    - You can configure database connections using a properties file, an XML deployment descriptor or programmatically

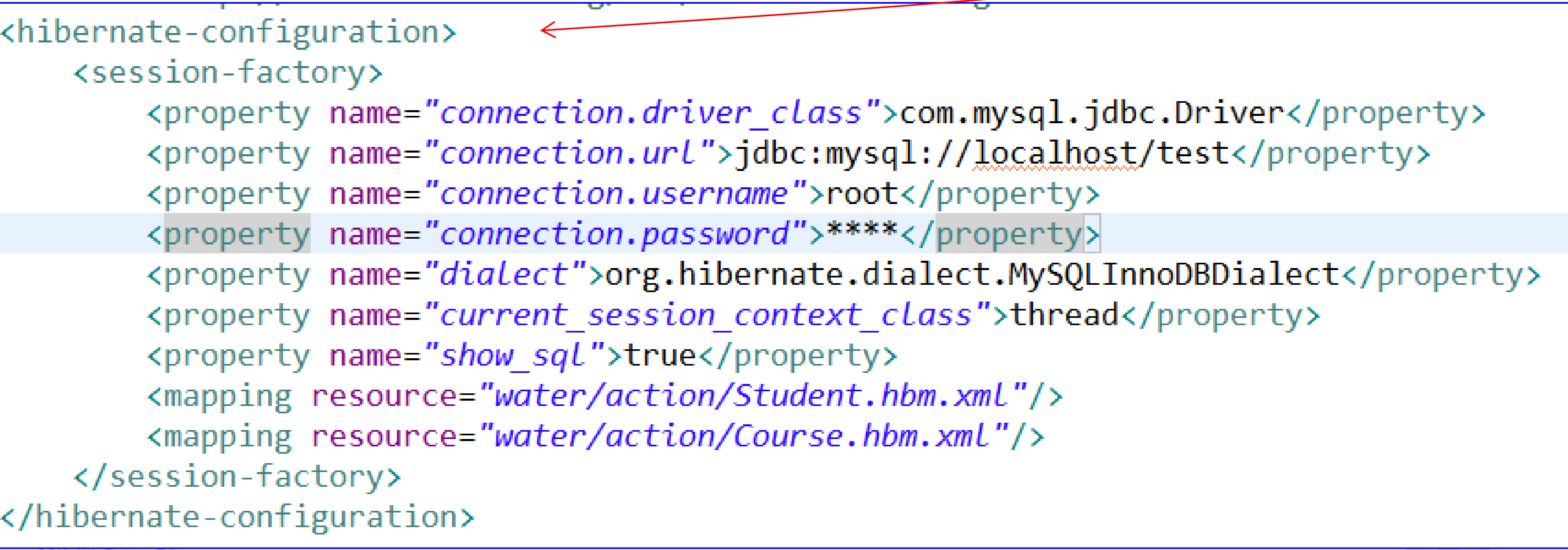
Traditional persistence and Hibernate

- Hibernate (cont.)
  - Configuration
    - An instance of org.hibernate.cfg.Configuration represents an entire set of mappings of an application's Java types to an SQL database
    - The org.hibernate.cfg.Configuration is used to build an immutable org.hibernate.SessionFactory and compiles the mappings from various XML mapping files

configure database connections using hibernate.cfg.xml

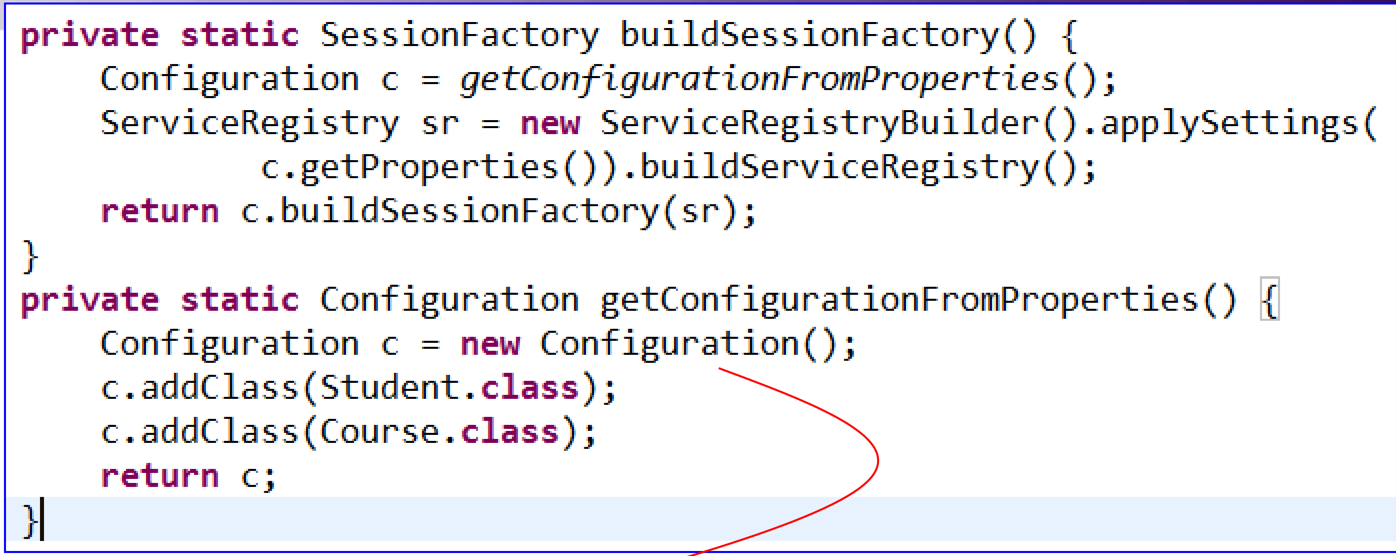


```java
private static SessionFactory buildSessionFactory() {
     Configuration c = getConfigurationFromXML();
    ServiceRegistry sr = new ServiceRegistryBuilder().applySettings(
            c.getProperties()).buildServiceRegistry();
    return c.buildSessionFactory(sr);
}

private static Configuration getConfigurationFromXML() {
    Configuration c = new Configuration().configure();
    return c;
}
```

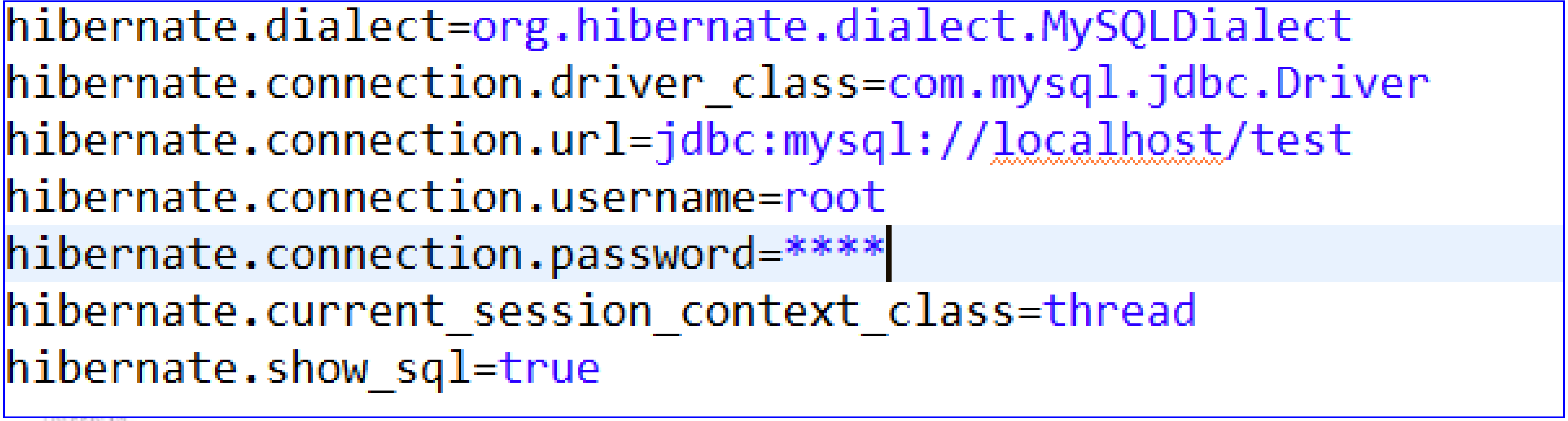


```xml
<hibernate-configuration>
    <session-factory>
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost/test</property>
        <property name="connection.username">root</property>
        <property name="connection.password">****</property>
        <property name="dialect">org.hibernate.dialect.MySQLInnoDBDialect</property>
        <property name="current_session_context_class">thread</property>
        <property name="show_sql">true</property>
        <mapping resource="water/action/Student.hbm.xml"/>
        <mapping resource="water/action/Course.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

configure database connections using hibernate.properties



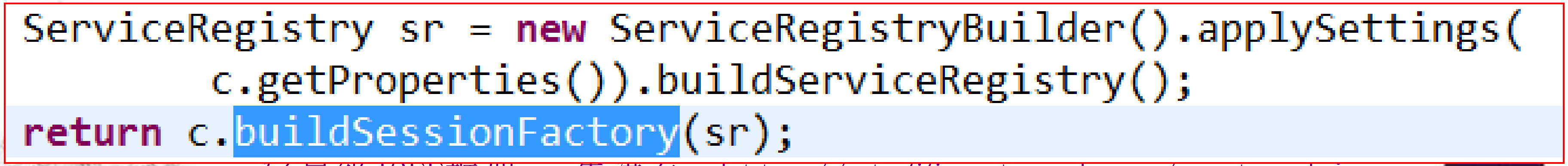
```java
private static SessionFactory buildSessionFactory() {
    Configuration c = getConfigurationFromProperties();
    ServiceRegistry sr = new ServiceRegistryBuilder().applySettings(
            c.getProperties()).buildServiceRegistry();
    return c.buildSessionFactory(sr);
}
private static Configuration getConfigurationFromProperties() {
    Configuration c = new Configuration();
    c.addClass(Student.class);
    c.addClass(Course.class);
    return c;
}
```



```
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/test
hibernate.connection.username=root
hibernate.connection.password=****
hibernate.current_session_context_class=thread
hibernate.show_sql=true
```

Traditional persistence and Hibernate

- Hibernate (cont.)
  - Obtaining a SessionFactory
    - When all mappings have been parsed by the org.hibernate.cfg.Configuration, the application must obtain a factory for org.hibernate.Session instances
    - Hibernate does allow your application to instantiate more than one org.hibernate.SessionFactory

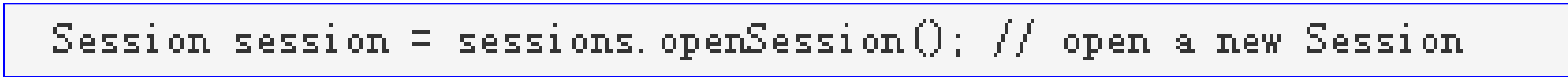


```
ServiceRegistry sr = new ServiceRegistryBuilder().applySettings(
        c.getProperties()).buildServiceRegistry();
return c.buildSessionFactory(sr);
```

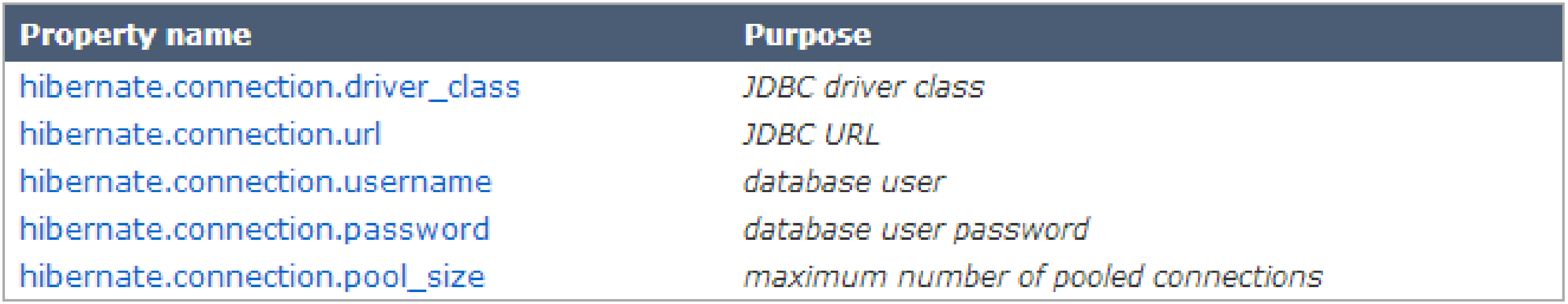
Traditional persistence and Hibernate

- Hibernate (cont.)
  - JDBC Connection
    - It is advisable to have the org.hibernate.SessionFactory create and pool JDBC connections for you
    - Once you start a task that requires access to the database, a JDBC connection will be obtained from the pool
    - you first need to pass some JDBC connection properties to Hibernate

Hibernate JDBC Properties



```
Session session = sessions.openSession(); // open a new Session
```

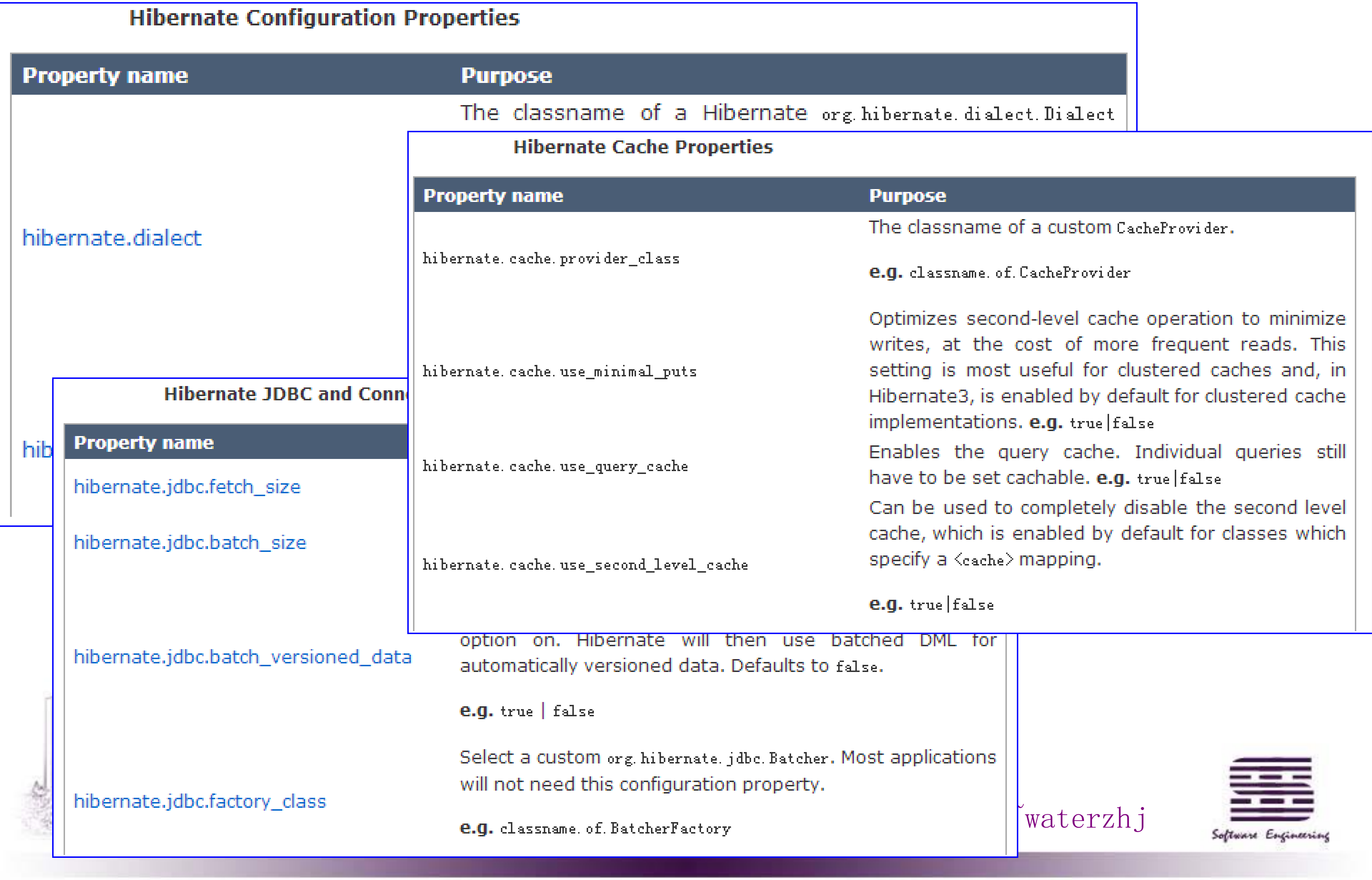


Property name	Purpose
hibernate.connection.driver_class	*JDBC driver class*
hibernate.connection.url	*JDBC URL*
hibernate.connection.username	*database user*
hibernate.connection.password	*database user password*
hibernate.connection.pool_size	*maximum number of pooled connections*

Traditional persistence and Hibernate

- Hibernate (cont.)
  - Optional Configuration Properties
    - There are a number of other properties that control the behavior of Hibernate at runtime. All are optional and have reasonable default values
    - Hibernate configuration properties
    - Hibernate JDBC and connection properties
    - Hibernate cache properties
    - Hibernate transaction properties
    - Etc.

Optional Configuration Properties



Hibernate Configuration Properties

Property name	Purpose
hibernate.dialect	The classname of a Hibernate org.hibernate.dialect.Dialect
hib	

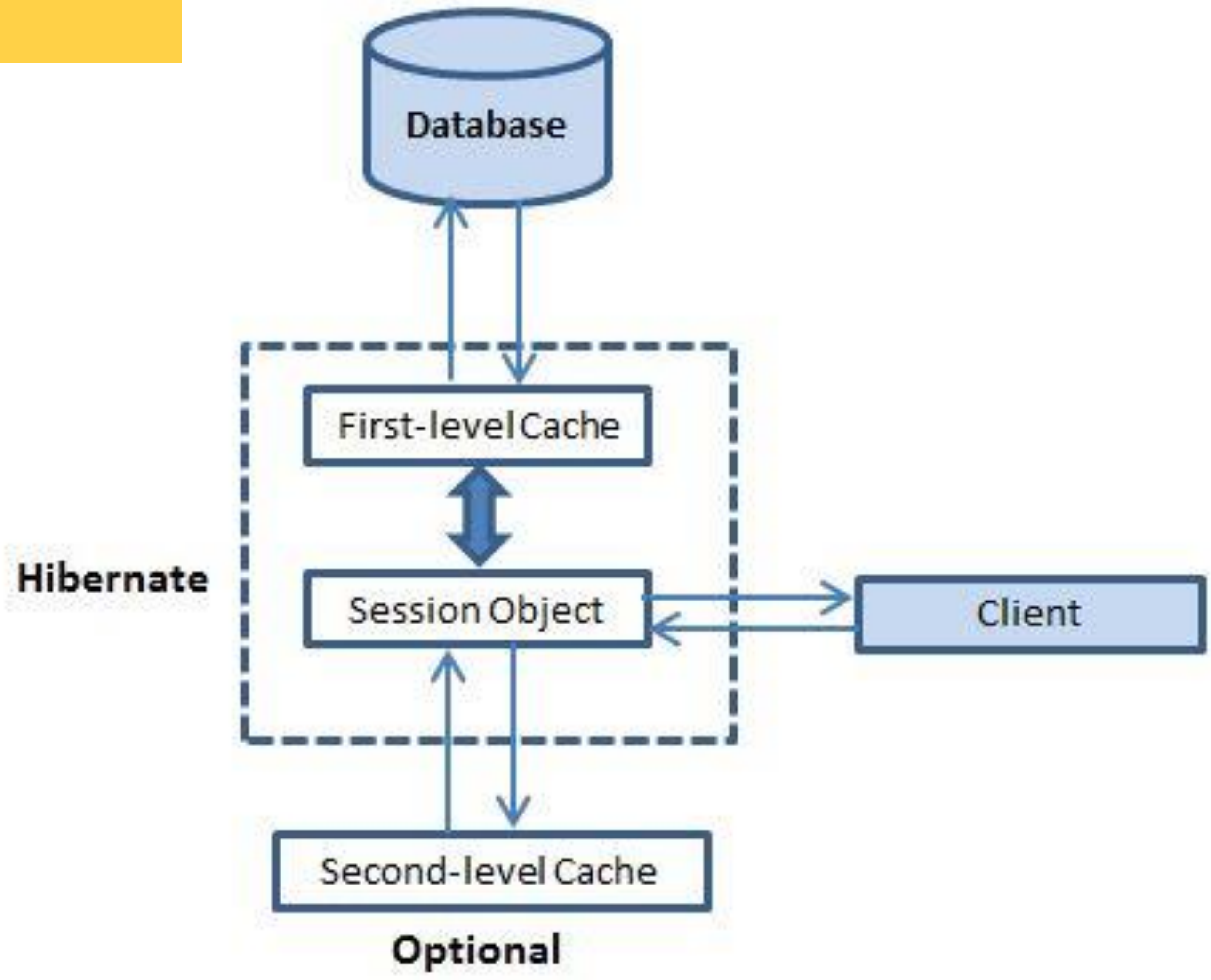
Hibernate Cache Properties

Property name	Purpose
hibernate.cache.provider_class	The classname of a custom CacheProvider. **e.g.** classname.of.CacheProvider
hibernate.cache.use_minimal_puts	Optimizes second-level cache operation to minimize writes, at the cost of more frequent reads. This setting is most useful for clustered caches and, in Hibernate3, is enabled by default for clustered cache implementations. **e.g.** true|false
hibernate.cache.use_query_cache	Enables the query cache. Individual queries still have to be set cachable. **e.g.** true|false
hibernate.cache.use_second_level_cache	Can be used to completely disable the second level cache, which is enabled by default for classes which specify a <cache> mapping. **e.g.** true|false

Hibernate JDBC and Conn

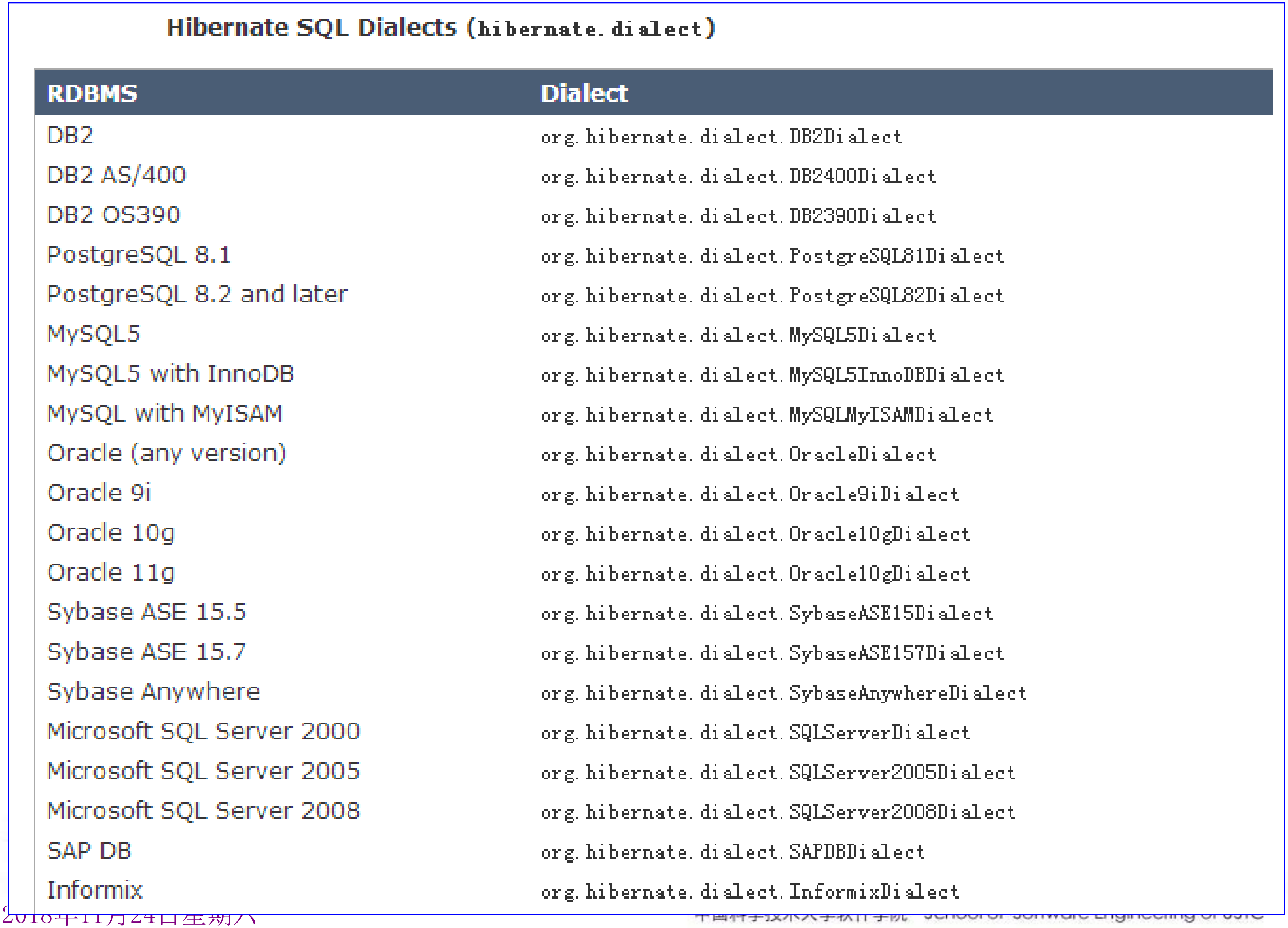
Property name	
hibernate.jdbc.fetch_size	
hibernate.jdbc.batch_size	
hibernate.jdbc.batch_versioned_data	option on. Hibernate will then use batched DML for automatically versioned data. Defaults to false. **e.g.** true | false
hibernate.jdbc.factory_class	Select a custom org.hibernate.jdbc.Batcher. Most applications will not need this configuration property. **e.g.** classname.of.BatcherFactory

Two level caches in Hibernate



Traditional persistence and Hibernate

- Hibernate (cont.)
  - SQL Dialect
    - Always set the hibernate.dialect property to the correct org.hibernate.dialect.Dialect subclass for your database
    - If you specify a dialect, Hibernate will use sensible defaults for some of the configuration properties, such as connection, cache, transaction properties. And you will not have to specify those properties manually



Hibernate SQL Dialects (hibernate.dialect)

RDBMS	Dialect
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
PostgreSQL 8.1	org.hibernate.dialect.PostgreSQL81Dialect
PostgreSQL 8.2 and later	org.hibernate.dialect.PostgreSQL82Dialect
MySQL5	org.hibernate.dialect.MySQL5Dialect
MySQL5 with InnoDB	org.hibernate.dialect.MySQL5InnoDBDialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Oracle 11g	org.hibernate.dialect.Oracle10gDialect
Sybase ASE 15.5	org.hibernate.dialect.SybaseASE15Dialect
Sybase ASE 15.7	org.hibernate.dialect.SybaseASE157Dialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect
Microsoft SQL Server 2000	org.hibernate.dialect.SQLServerDialect
Microsoft SQL Server 2005	org.hibernate.dialect.SQLServer2005Dialect
Microsoft SQL Server 2008	org.hibernate.dialect.SQLServer2008Dialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Informix	org.hibernate.dialect.InformixDialect

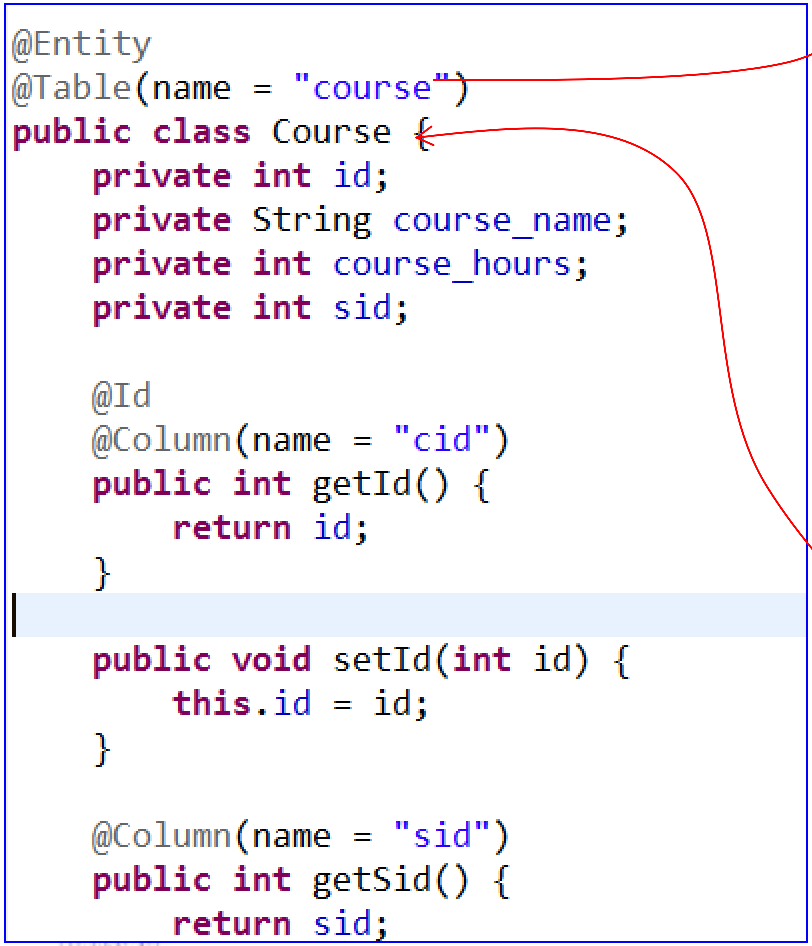
Basic O/R Mapping

- Persistent Classes
  - Persistent classes are classes in an application that implement the entities of the business problem
  - Rules
    - Implement a no-argument constructor
    - Provide an identifier property
    - Prefer non-final class
    - Declare accessors and mutators for persistent fields

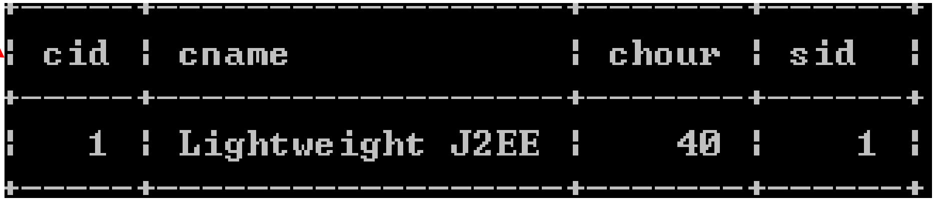
Basic O/R Mapping

- Mapping Declaration
  - Object/relational mappings can be defined in three approaches
    - using Java 5 annotations (via the Java Persistence 2 annotations)
    - using JPA 2 XML deployment descriptors
    - using the Hibernate legacy XML files approach known as hbm.xml

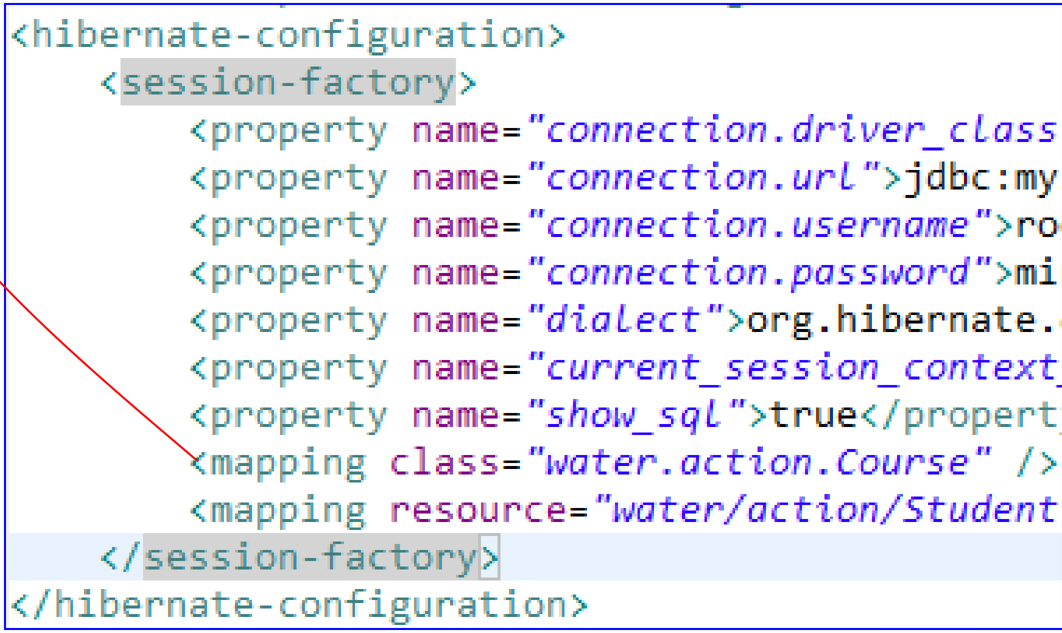
Mapping Declare by Using Java Annotations Demo



```
@Entity
@Table(name = "course")
public class Course {
    private int id;
    private String course_name;
    private int course_hours;
    private int sid;

    @Id
    @Column(name = "cid")
    public int getId() {
        return id;
    }

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

    @Column(name = "sid")
    public int getSid() {
        return sid;
```

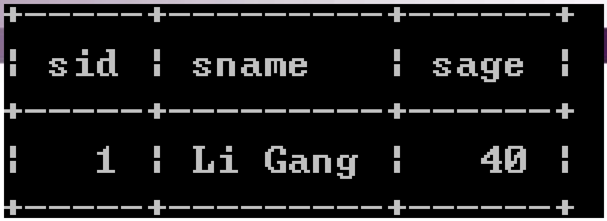


```
+-----+--------------------+-------+-------+
| cid | cname              | chour | sid   |
+-----+--------------------+-------+-------+
|   1 | Lightweight J2EE   |    40 |     1 |
+-----+--------------------+-------+-------+
```

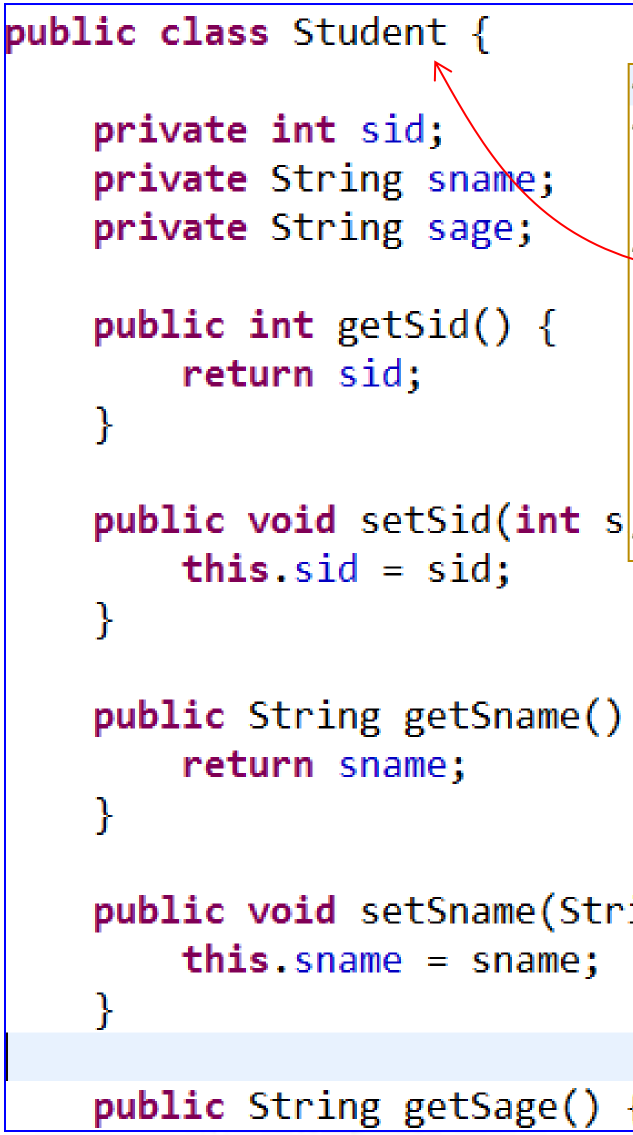


```
<hibernate-configuration>
    <session-factory>
        <property name="connection.driver_class
        <property name="connection.url">jdbc:my
        <property name="connection.username">ro
        <property name="connection.password">mi
        <property name="dialect">org.hibernate.
        <property name="current_session_context
        <property name="show_sql">true</propert
        <mapping class="water.action.Course" />
        <mapping resource="water/action/Student
    </session-factory>
</hibernate-configuration>
```

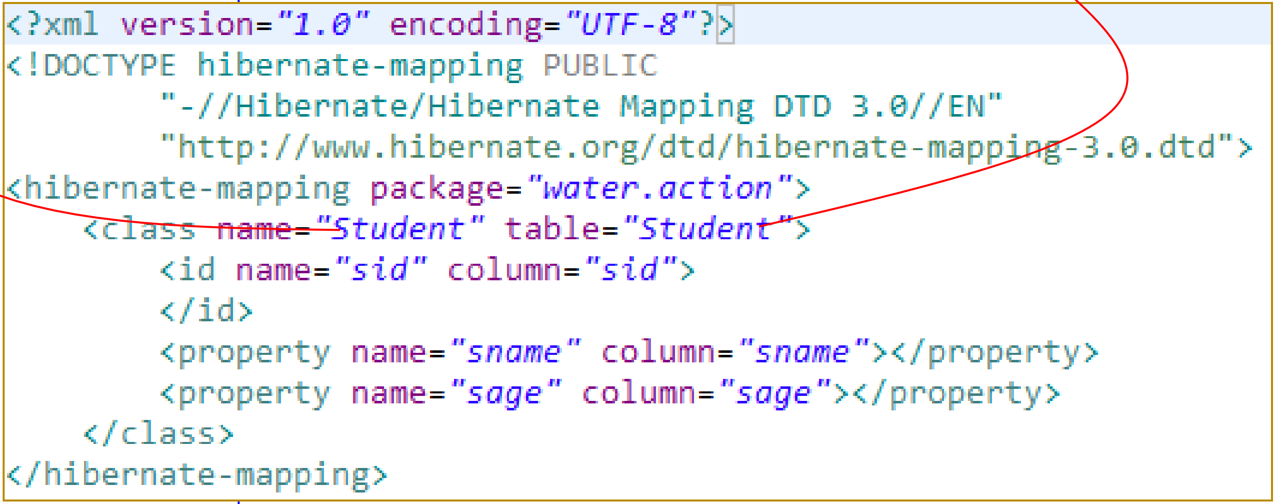
Mapping Declare by Using Hibernate Legacy XML Files



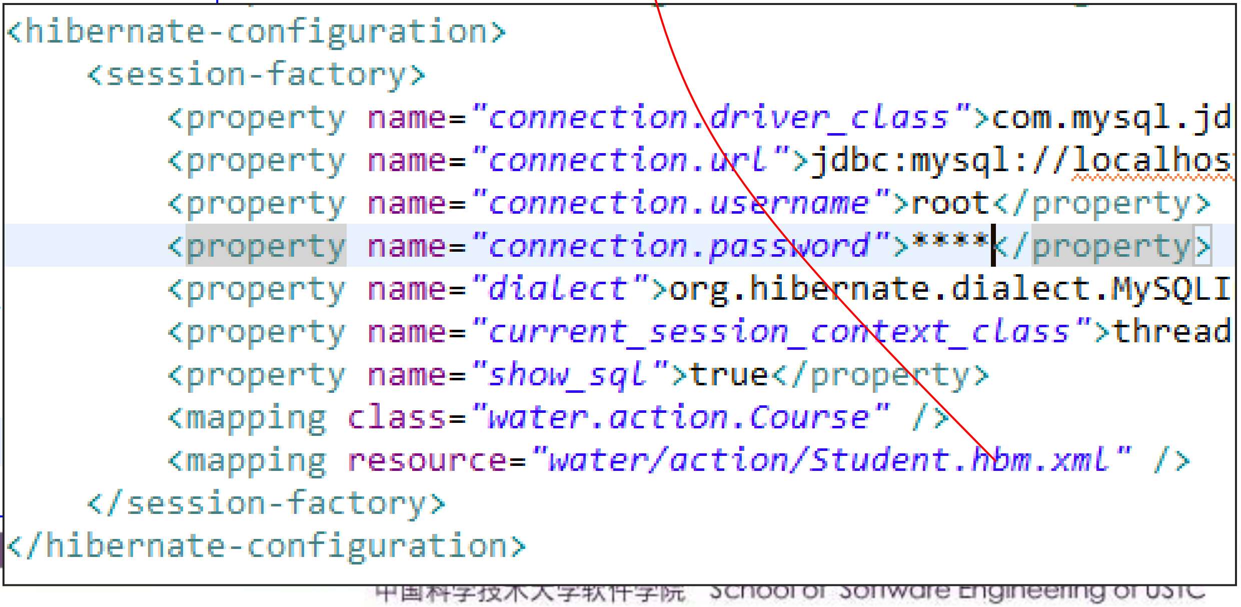
```
+------+-----------+--------+
| sid  | sname     | sage   |
+------+-----------+--------+
|    1 | Li Gang   |     40 |
+------+-----------+--------+
```



```java
public class Student {

    private int sid;
    private String sname;
    private String sage;

    public int getSid() {
        return sid;
    }

    public void setSid(int s
        this.sid = sid;
    }

    public String getSname()
        return sname;
    }

    public void setSname(Stri
        this.sname = sname;
    }

    public String getSage() {
```



```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="water.action">
    <class name="Student" table="Student">
        <id name="sid" column="sid">
        </id>
        <property name="sname" column="sname"></property>
        <property name="sage" column="sage"></property>
    </class>
</hibernate-mapping>
```

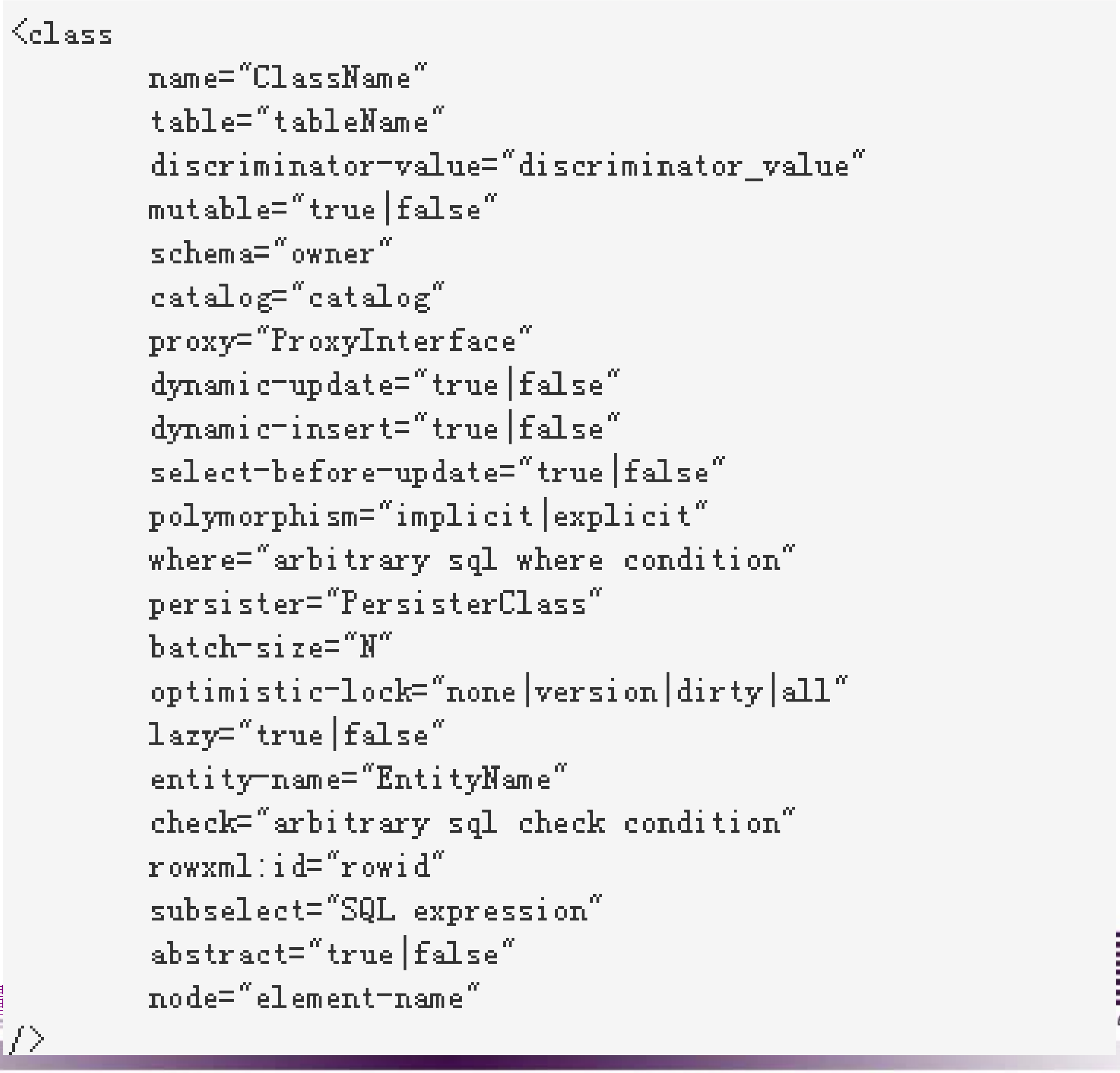


```xml
<hibernate-configuration>
    <session-factory>
        <property name="connection.driver_class">com.mysql.jd
        <property name="connection.url">jdbc:mysql://localhos
        <property name="connection.username">root</property>
        <property name="connection.password">****</property>
        <property name="dialect">org.hibernate.dialect.MySQLI
        <property name="current_session_context_class">thread
        <property name="show_sql">true</property>
        <mapping class="water.action.Course" />
        <mapping resource="water/action/Student.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

Basic O/R Mapping

- Entity Mapping
  - An entity is a regular Java object (aka POJO) which will be persisted by Hibernate
  - Using Annotation
    - @Entity
  - Using hbm.xml
    - You can declare a persistent class using the class element <class/>

Class Element in hbm.xml



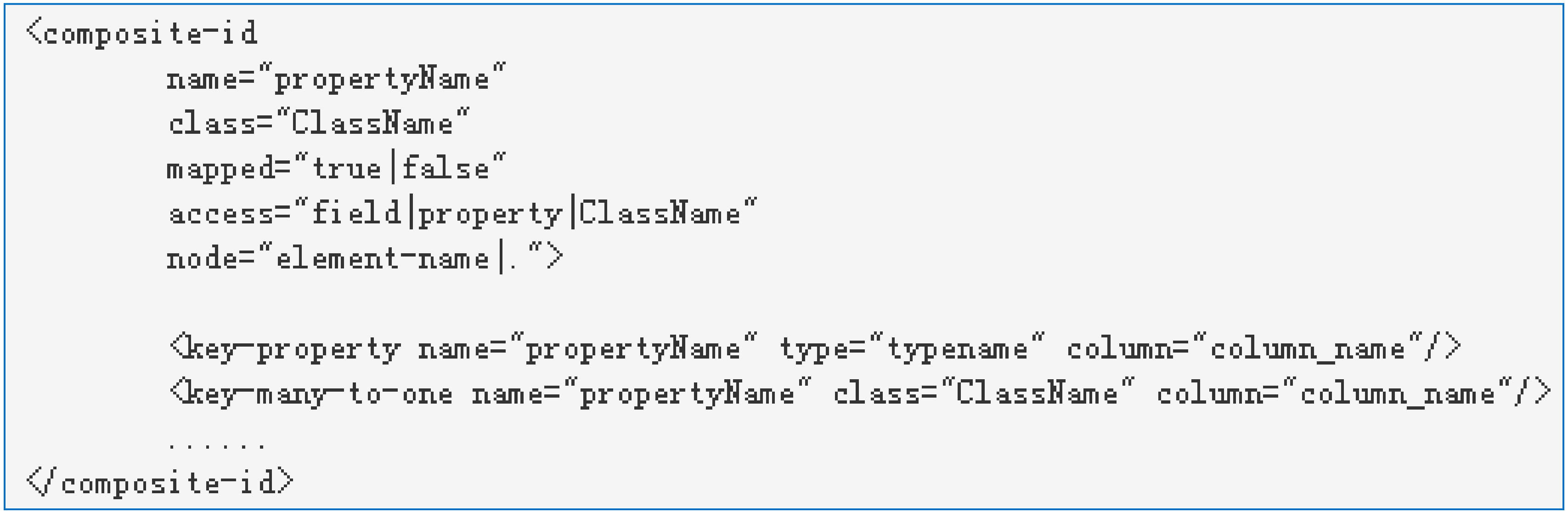
```
<class
        name=“ClassName”
        table=“tableName”
        discriminator-value=“discriminator_value”
        mutable=“true|false”
        schema=“owner”
        catalog=“catalog”
        proxy=“ProxyInterface”
        dynamic-update=“true|false”
        dynamic-insert=“true|false”
        select-before-update=“true|false”
        polymorphism=“implicit|explicit”
        where=“arbitrary sql where condition”
        persister=“PersisterClass”
        batch-size=“N”
        optimistic-lock=“none|version|dirty|all”
        lazy=“true|false”
        entity-name=“EntityName”
        check=“arbitrary sql check condition”
        rowxml:id=“rowid”
        subselect=“SQL expression”
        abstract=“true|false”
        node=“element-name”
/>
```

轻量

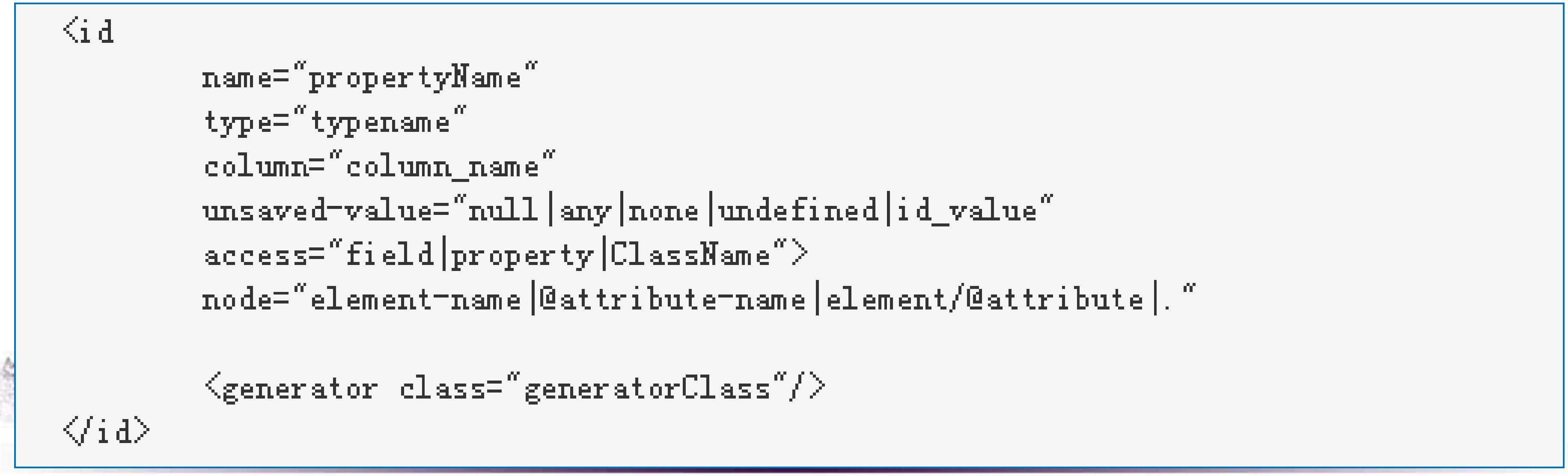
Basic O/R Mapping

- Identifiers Mapping
  - Mapped classes *must* declare the primary key column of the database table. Most classes will also have a JavaBeans-style property holding the unique identifier of an instance
  - Using Annotation
    - @Id
  - Using hbm.xml
    - <id/>
    - <composite-id>

Id and composite-id Element in hbm.xml



```
<composite-id
        name=“propertyName”
        class=“ClassName”
        mapped=“true|false”
        access=“field|property|ClassName”
        node=“element-name|.”>

        <key-property name=“propertyName” type=“typename” column=“column_name”/>
        <key-many-to-one name=“propertyName” class=“ClassName” column=“column_name”/>
        ......
</composite-id>
```

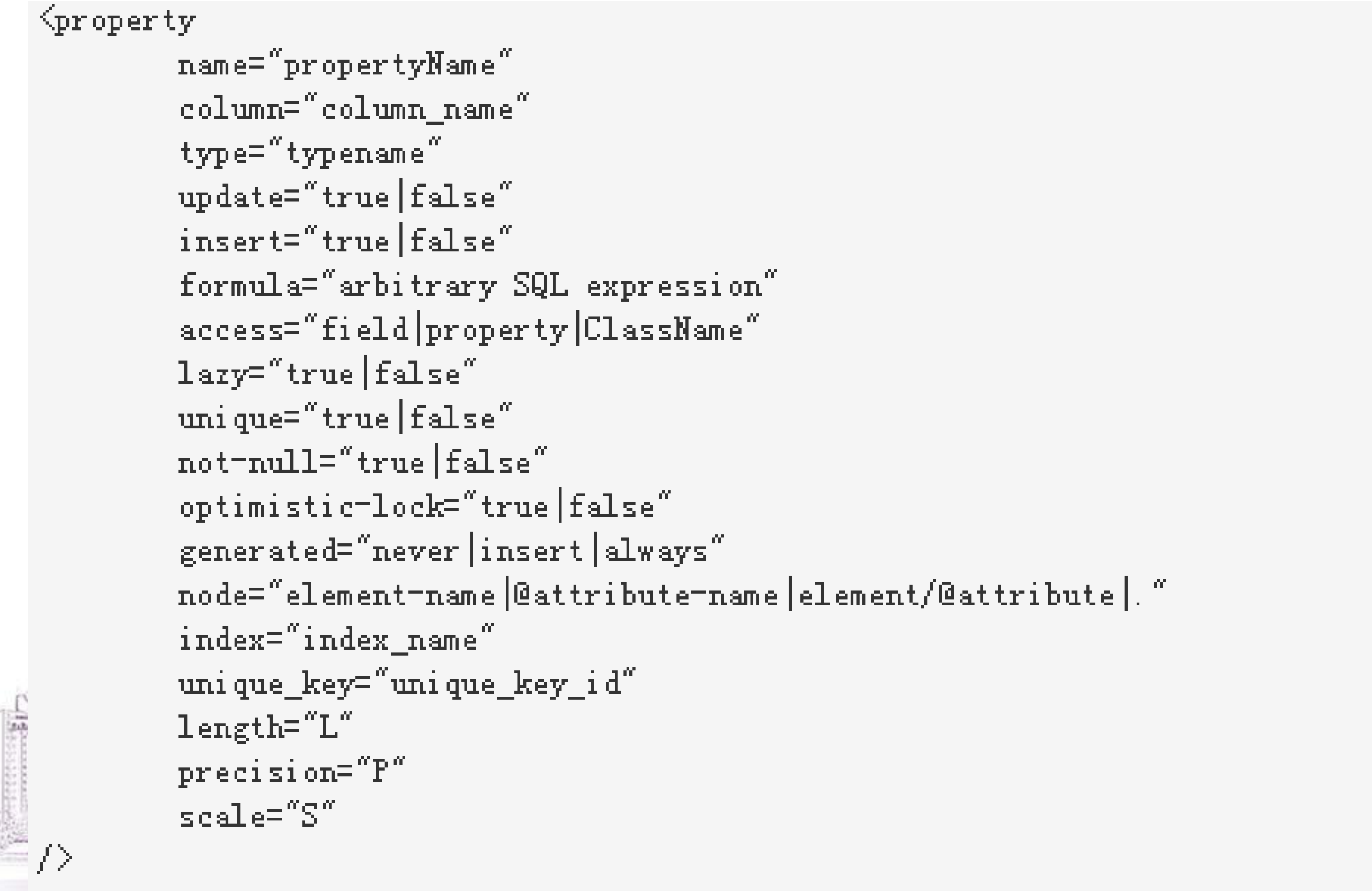


```
  <id
          name=“propertyName”
          type=“typename”
          column=“column_name”
          unsaved-value=“null|any|none|undefined|id_value”
          access=“field|property|ClassName”>
          node=“element-name|@attribute-name|element/@attribute|.”

          <generator class=“generatorClass”/>
  </id>
```

Basic O/R Mapping

- Property Mapping
  - You need to decide which property needs to be made persistent in a given entity
  - Using Annotation
    - @Basic
    - @Lob
  - Using hbm.xml
    - <property/>

Property Element in hbm.xml



```
<property
        name=“propertyName”
        column=“column_name”
        type=“typename”
        update=“true|false”
        insert=“true|false”
        formula=“arbitrary SQL expression”
        access=“field|property|ClassName”
        lazy=“true|false”
        unique=“true|false”
        not-null=“true|false”
        optimistic-lock=“true|false”
        generated=“never|insert|always”
        node=“element-name|@attribute-name|element/@attribute|.”
        index=“index_name”
        unique_key=“unique_key_id”
        length=“L”
        precision=“P”
        scale=“S”
/>
```

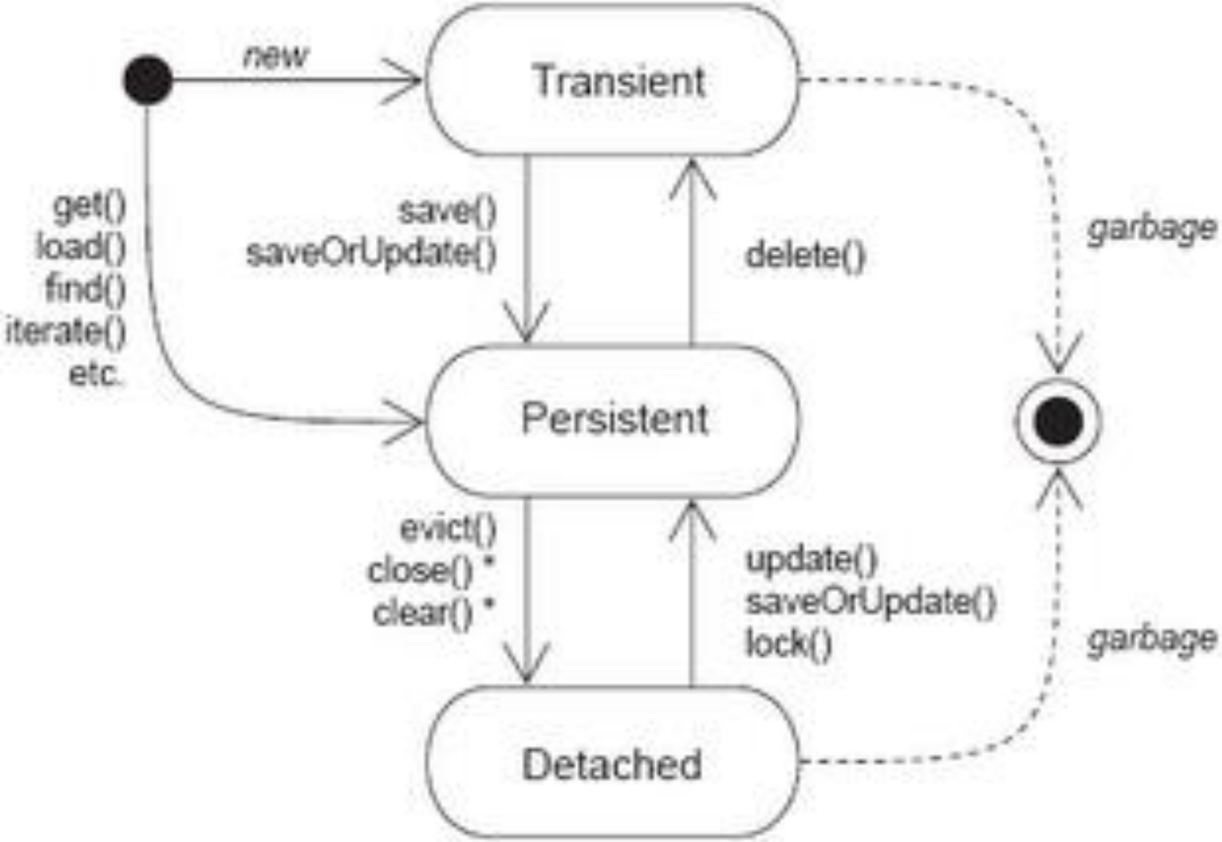
Basic O/R Mapping

- Persistence Contexts
  - Both the org.hibernate.Session API and javax.persistence.EntityManager API represent a context for dealing with persistent data
  - Persistent data has a state in relation to both a persistence context and the underlying database

Basic O/R Mapping

- Persistence Contexts (cont.)
  - Entity States
    - New or Transient
      - The entity has just been instantiated and is not associated with a persistence context
    - Managed or Persistent
    - Detached
      - a detached instance is an object that has been persistent, but its Session has been closed

Entity States

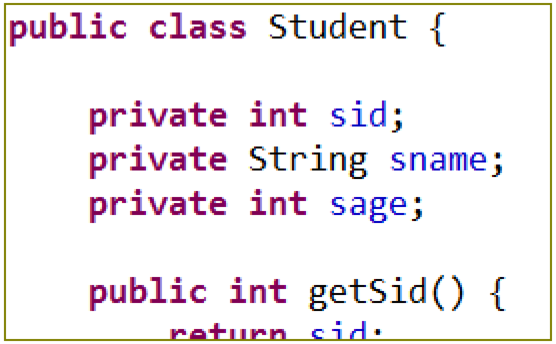


* affects all instances in a Session

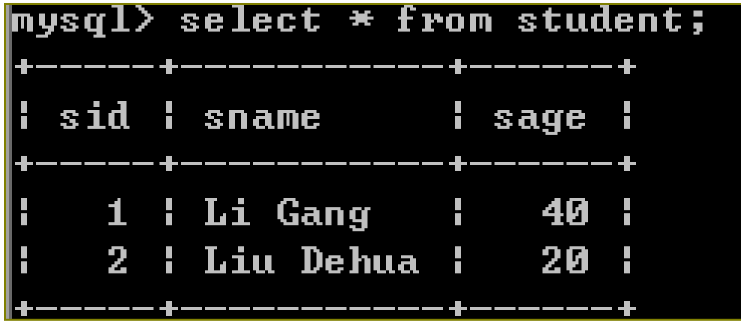
Basic O/R Mapping

- Making an Entity Persistent
  - Once you've created a new entity instance (using the standard new operator) it is in new state
  - You can make it persistent by associating it to a org.hibernate.Session
  - org.hibernate.Session has save and persist methods for the persistence

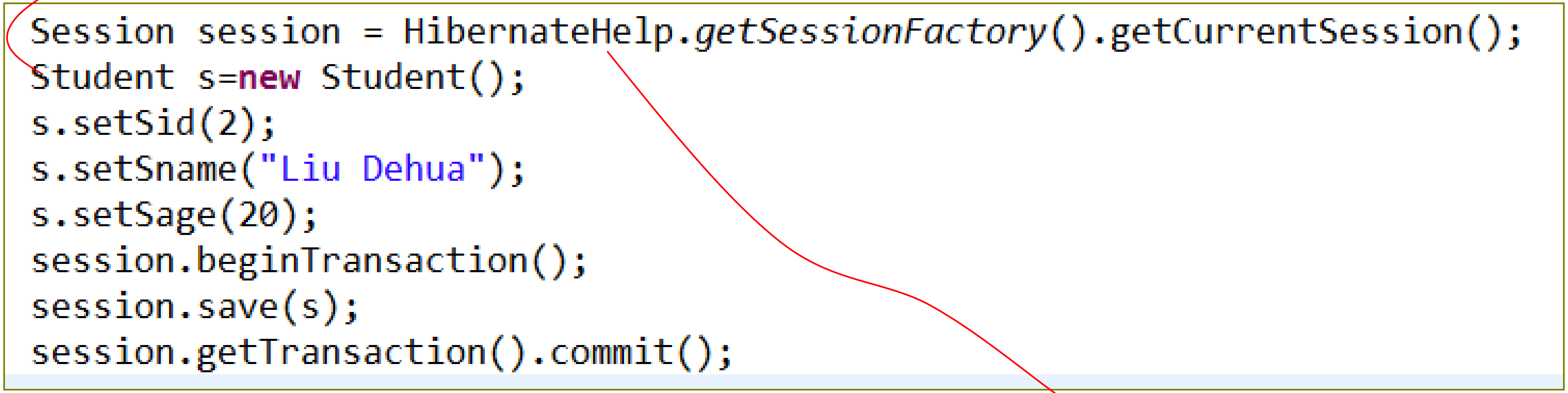
Making a New Entity Persistent Demo



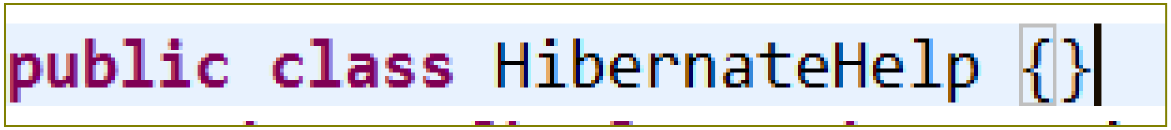
```java
public class Student {

    private int sid;
    private String sname;
    private int sage;

    public int getSid() {
        return sid;
```



```
mysql> select * from student;
+-----+-------------+------+
| sid | sname       | sage |
+-----+-------------+------+
|   1 | Li Gang     |   40 |
|   2 | Liu Dehua   |   20 |
+-----+-------------+------+
```



```java
Session session = HibernateHelp.getSessionFactory().getCurrentSession();
Student s=new Student();
s.setSid(2);
s.setSname("Liu Dehua");
s.setSage(20);
session.beginTransaction();
session.save(s);
session.getTransaction().commit();
```

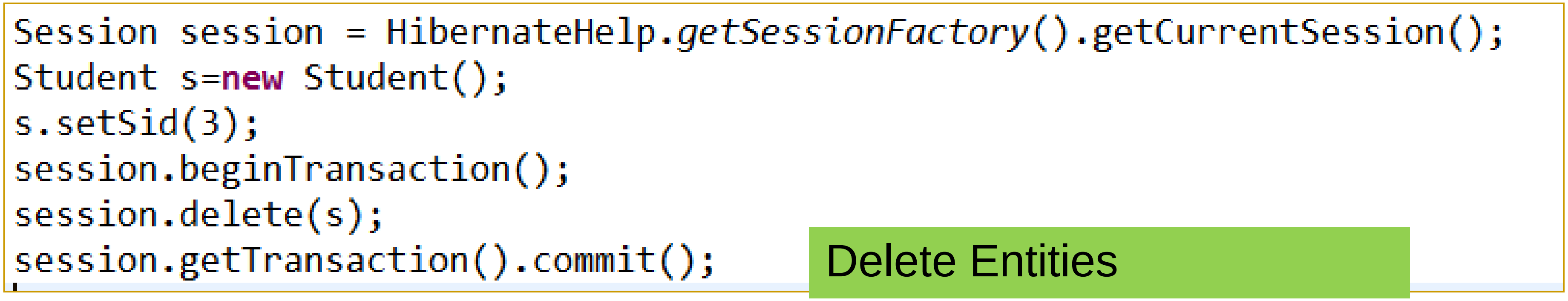


```java
public class HibernateHelp {}
```

Basic O/R Mapping

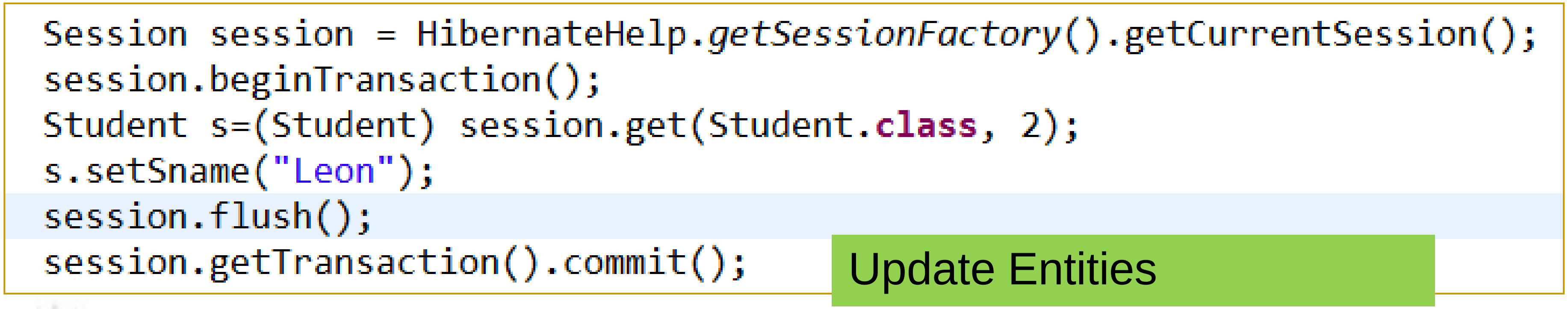
- Manage Entities
  - Delete
    - Entities can also be deleted
  - Modify
    - Entities in managed/persistent state may be manipulated by the application and any changes will be automatically detected and persisted when the persistence context is flushed
    - There is no need to call a particular method to make your modifications persistent

Manage Entities Demo



```
Session session = HibernateHelp.getSessionFactory().getCurrentSession();
Student s=new Student();
s.setSid(3);
session.beginTransaction();
session.delete(s);
session.getTransaction().commit();
```

Delete Entities



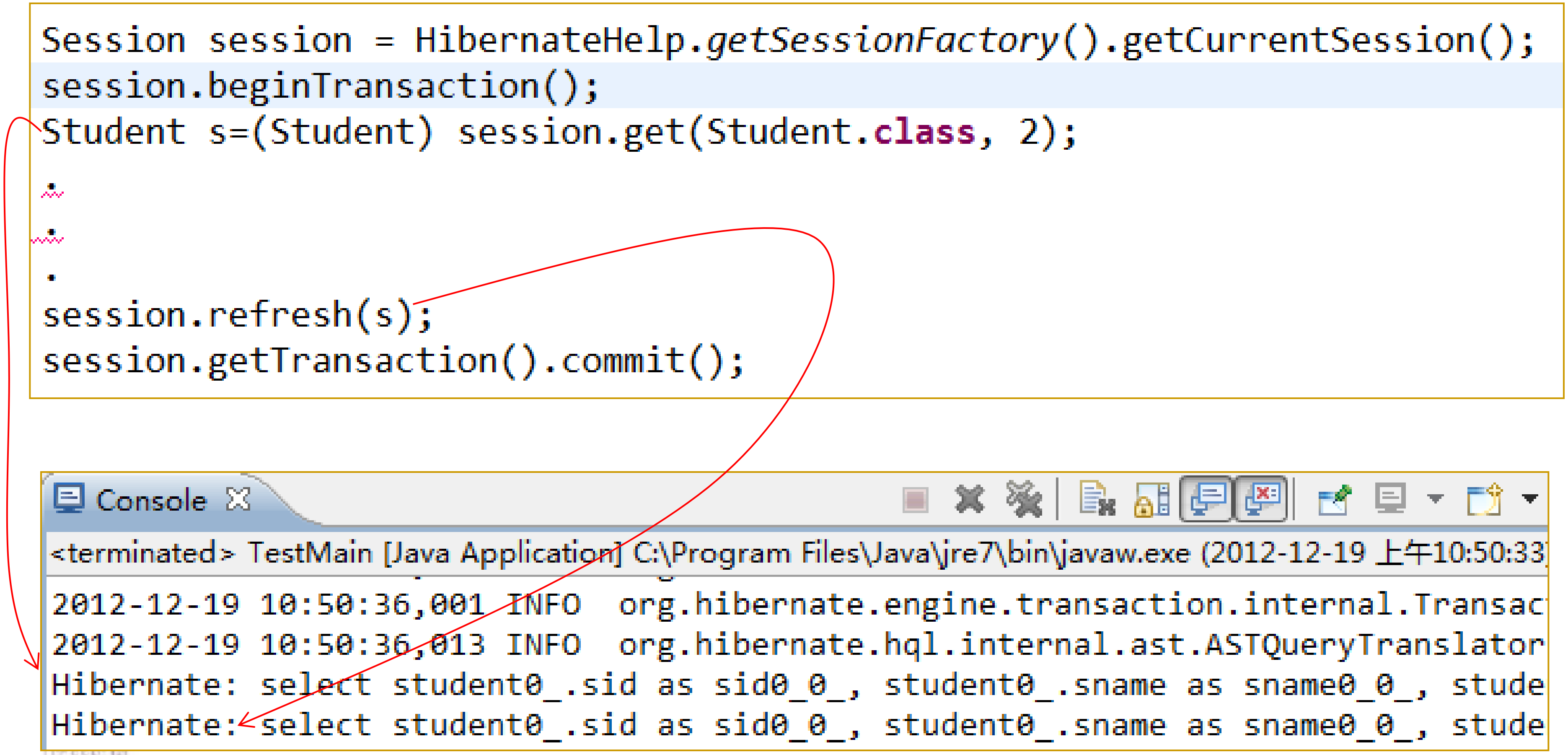
```
Session session = HibernateHelp.getSessionFactory().getCurrentSession();
session.beginTransaction();
Student s=(Student) session.get(Student.class, 2);
s.setSname("Leon");
session.flush();
session.getTransaction().commit();
```

Update Entities

Basic O/R Mapping

- Refresh Entity State
  - You can reload an entity instance and it's collections at any time
  - Refreshing allows the current database state to be pulled into the entity instance and the persistence context
  - Note that only the entity instance and its collections are refreshed unless you specify REFRESH as a cascade style of any associations

Refresh Entity State Demo



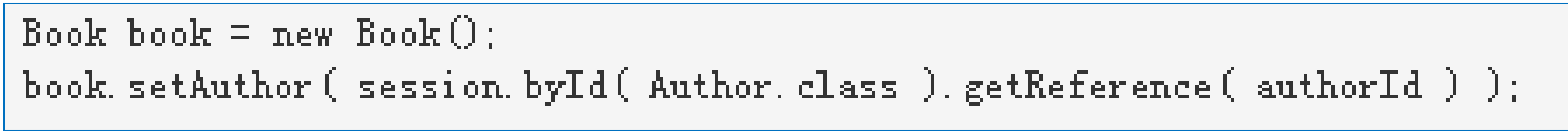
```
Session session = HibernateHelp.getSessionFactory().getCurrentSession();
session.beginTransaction();
Student s=(Student) session.get(Student.class, 2);
.
.
.
session.refresh(s);
session.getTransaction().commit();
```

```
<terminated> TestMain [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (2012-12-19 上午10:50:33)
2012-12-19 10:50:36,001 INFO  org.hibernate.engine.transaction.internal.Transac
2012-12-19 10:50:36,013 INFO  org.hibernate.hql.internal.ast.ASTQueryTranslator
Hibernate: select student0_.sid as sid0_0_, student0_.sname as sname0_0_, stude
Hibernate: select student0_.sid as sid0_0_, student0_.sname as sname0_0_, stude
```

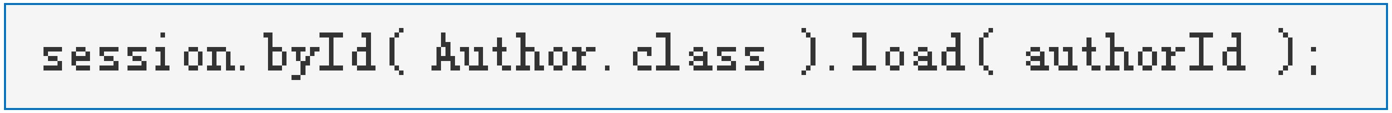
Basic O/R Mapping

- Obtain an Entity
  - Obtain an entity reference without initializing its data
    - By getReference() method
    - Should be used in cases where the identifier is assumed to exist
  - Obtain an entity with its data initialized
    - By load() method
      - does not immediately incur a call to the database
    - By get() method
      - always hit the database

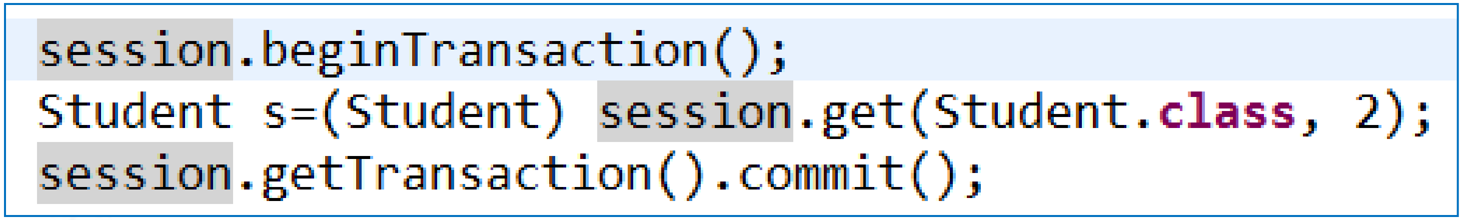
Obtain an Entity by getReference, load, or get method Demo



```java
Book book = new Book();
book.setAuthor( session.byId( Author.class ).getReference( authorId ) );
```



```java
session.byId( Author.class ).load( authorId );
```

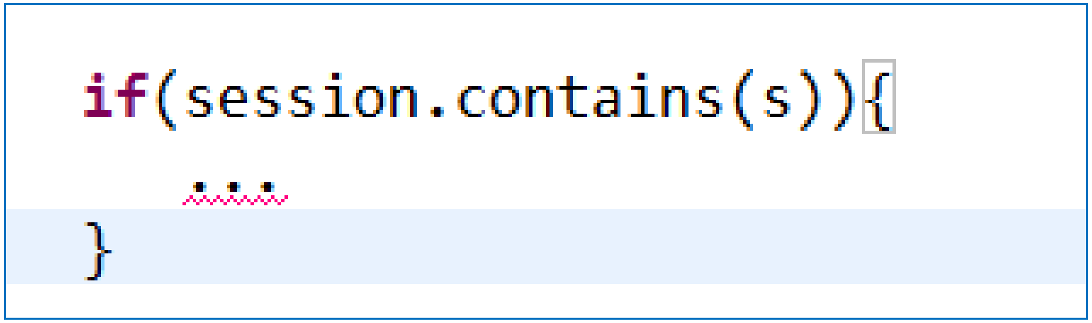


```java
session.beginTransaction();
Student s=(Student) session.get(Student.class, 2);
session.getTransaction().commit();
```

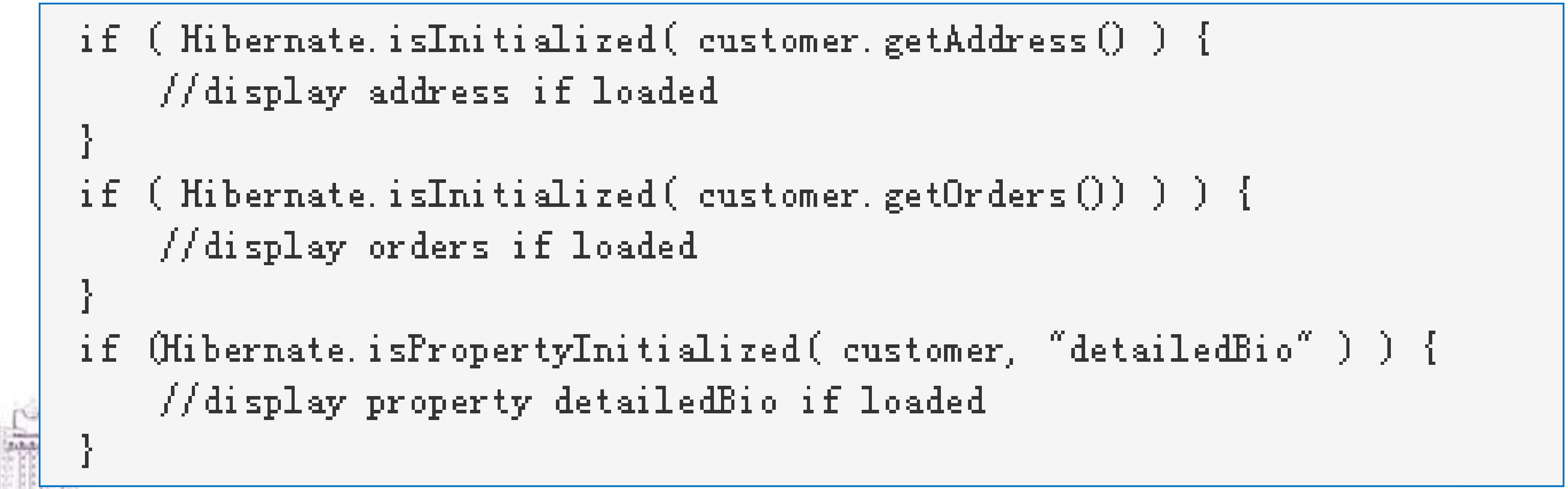
Basic O/R Mapping

- Check Persistent State
  - An application can verify the state of entities and collections in relation to the persistence context
  - Verify managed state
    - Session.contains()
  - Verify laziness
    - Hibernate.isInitialized()
    - Hibernate.isPropertyInitialized()

Check Persistent State Demo



```
if(session.contains(s)){
    ...
}
```



```
if ( Hibernate.isInitialized( customer.getAddress() ) {
    //display address if loaded
}
if ( Hibernate.isInitialized( customer.getOrders()) ) ) {
    //display orders if loaded
}
if (Hibernate.isPropertyInitialized( customer, “detailedBio” ) ) {
    //display property detailedBio if loaded
}
```

Association and Collection Mapping

- Collection Mapping
  - A Collection denotes a one-to-one or one-to-many relationship between tables of a database
  - Naturally Hibernate also allows to persist collections. These persistent collections can contain almost any other Hibernate type, including: basic types, custom types, components and references to other entities

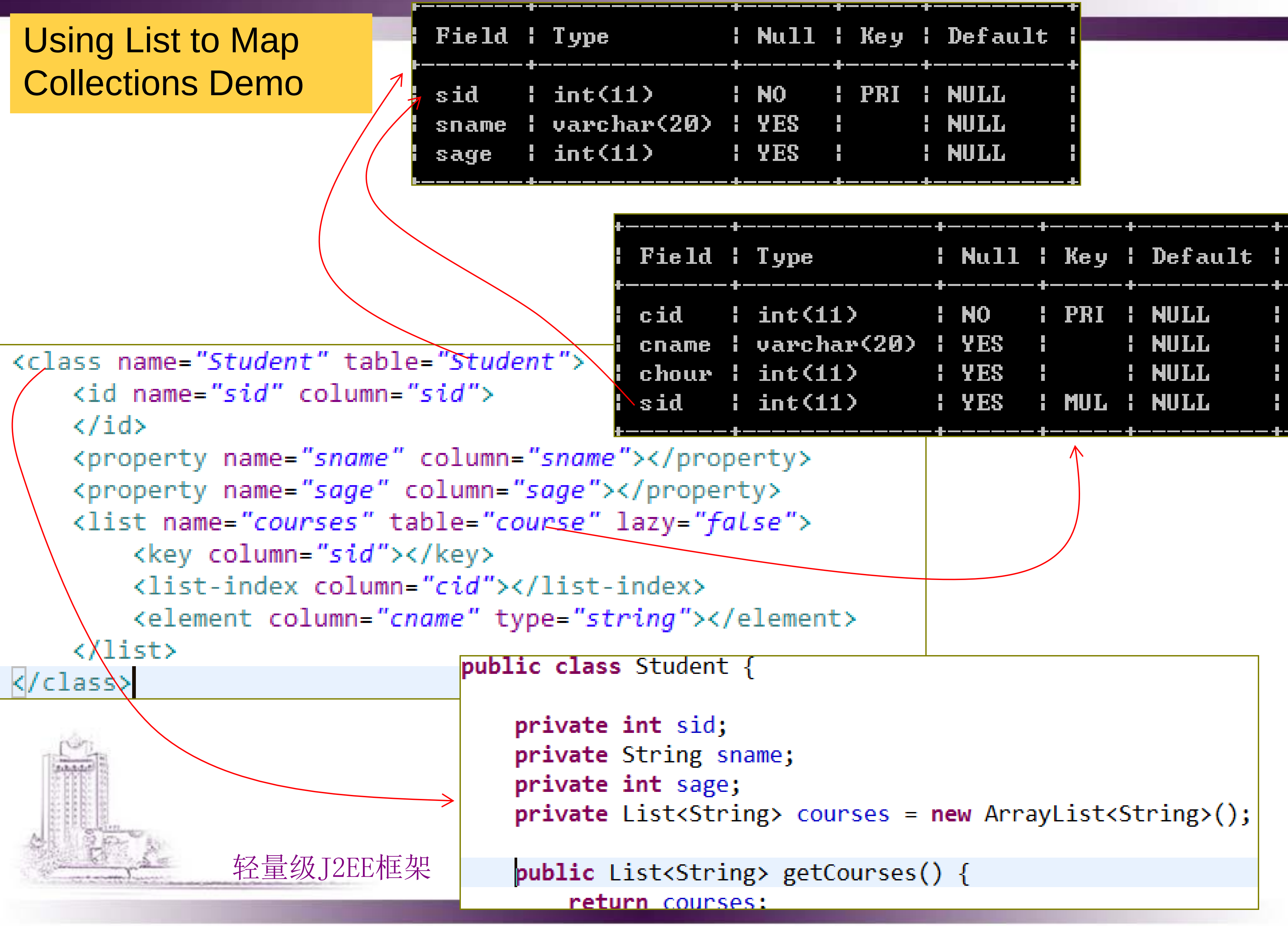
Association and Collection Mapping

- Collection Mapping (cont.)
  - As a requirement persistent collection-valued fields must be declared as an interface type
    - might be java.util.Set, java.util.Collection, java.util.List, java.util.Map, java.util.SortedSet, java.util.SortedMap or something else
  - The persistent collections injected by Hibernate behave like HashMap, HashSet, TreeMap, TreeSet or ArrayList, depending on the interface type

Association and Collection Mapping

- Collection Mapping (cont.)
  - How to Map Collections
    - Use Annotation
      - @OneToMany
      - @ManyToMany
      - @ElementCollection
    - Use hbm.xml
      - <set/>
      - <list/>
      - <map/>
      - Etc.

Using List to Map Collections Demo



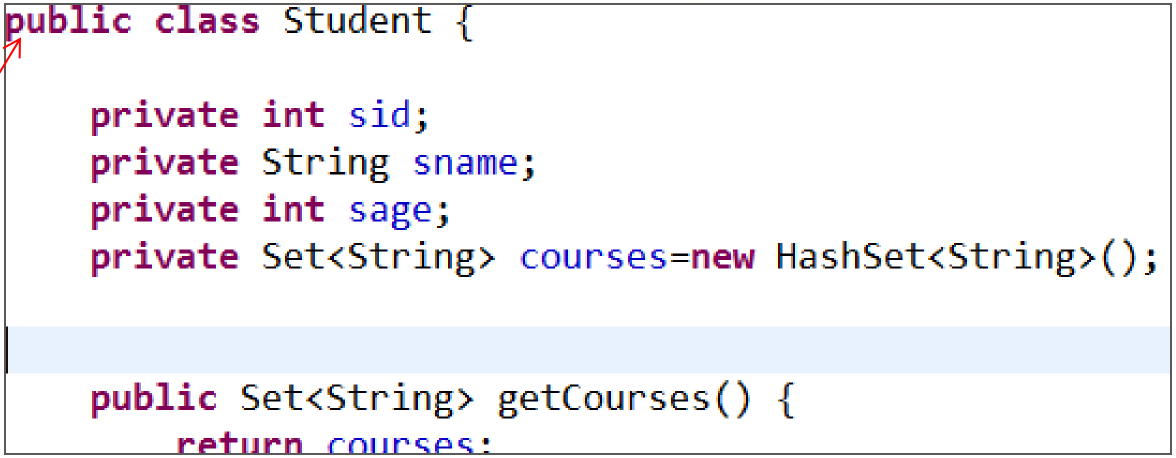
```
+--------+-------------+------+-----+---------+
| Field  | Type        | Null | Key | Default |
+--------+-------------+------+-----+---------+
| sid    | int(11)     | NO   | PRI | NULL    |
| sname  | varchar(20) | YES  |     | NULL    |
| sage   | int(11)     | YES  |     | NULL    |
+--------+-------------+------+-----+---------+
```

```
+--------+-------------+------+-----+---------+
| Field  | Type        | Null | Key | Default |
+--------+-------------+------+-----+---------+
| cid    | int(11)     | NO   | PRI | NULL    |
| cname  | varchar(20) | YES  |     | NULL    |
| chour  | int(11)     | YES  |     | NULL    |
| sid    | int(11)     | YES  | MUL | NULL    |
+--------+-------------+------+-----+---------+
```

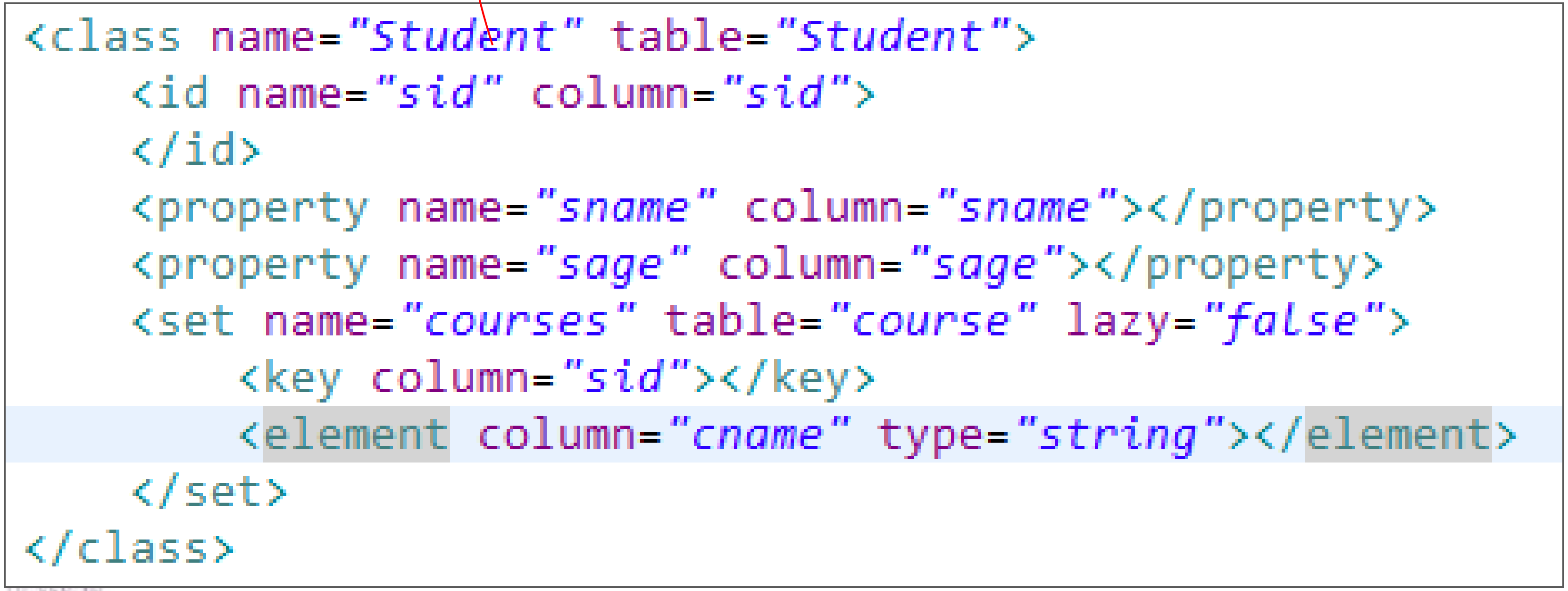
```xml
<class name="Student" table="Student">
    <id name="sid" column="sid">
    </id>
    <property name="sname" column="sname"></property>
    <property name="sage" column="sage"></property>
    <list name="courses" table="course" lazy="false">
        <key column="sid"></key>
        <list-index column="cid"></list-index>
        <element column="cname" type="string"></element>
    </list>
</class>
```

```java
public class Student {

    private int sid;
    private String sname;
    private int sage;
    private List<String> courses = new ArrayList<String>();

    public List<String> getCourses() {
        return courses;
```

Using Set to Map Collections Demo

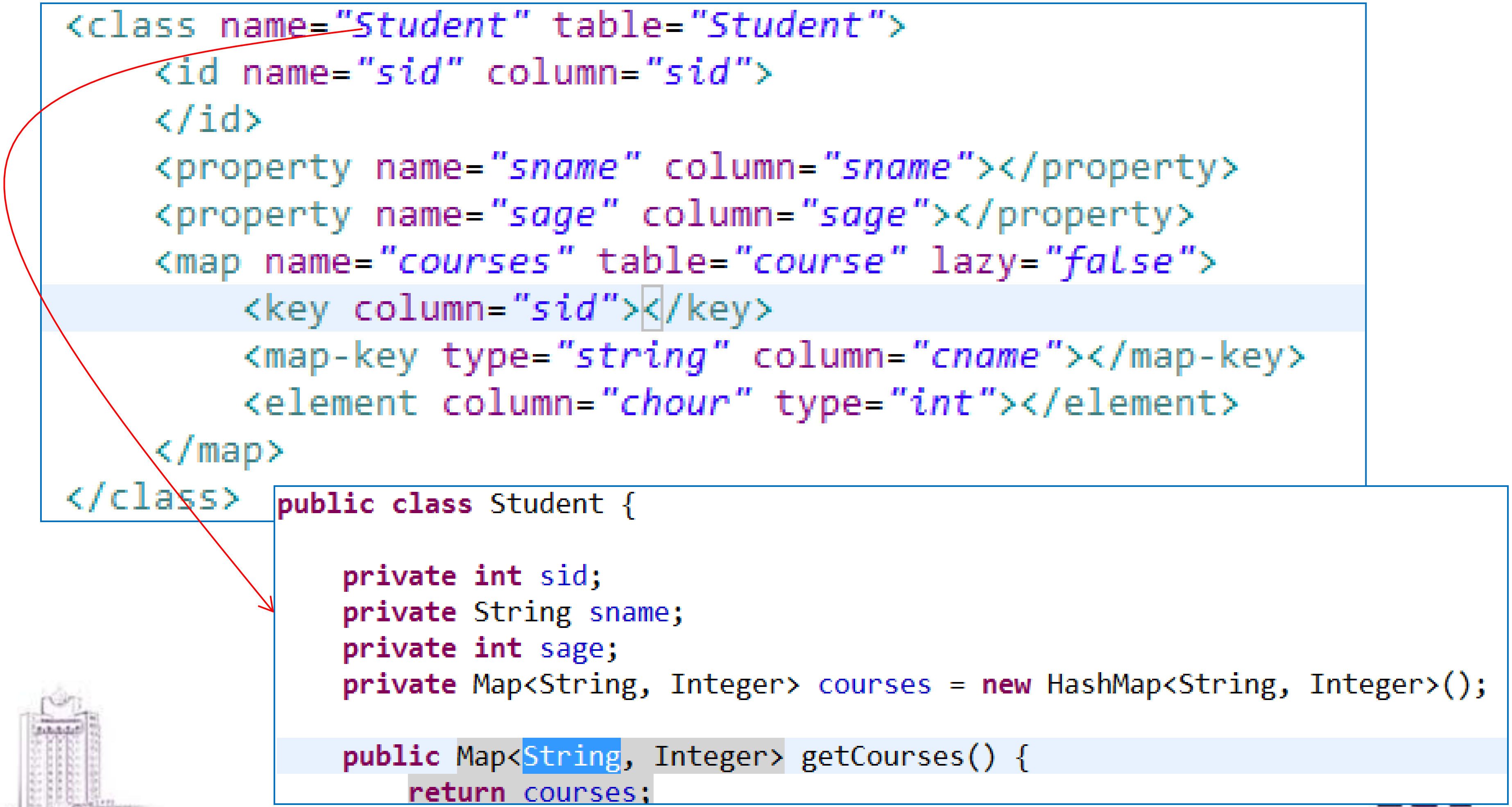


```java
public class Student {

    private int sid;
    private String sname;
    private int sage;
    private Set<String> courses=new HashSet<String>();


    public Set<String> getCourses() {
        return courses;
```



```xml
<class name="Student" table="Student">
    <id name="sid" column="sid">
    </id>
    <property name="sname" column="sname"></property>
    <property name="sage" column="sage"></property>
    <set name="courses" table="course" lazy="false">
        <key column="sid"></key>
        <element column="cname" type="string"></element>
    </set>
</class>
```

Using Map to Map Collections Demo



```
<class name="Student" table="Student">
    <id name="sid" column="sid">
    </id>
    <property name="sname" column="sname"></property>
    <property name="sage" column="sage"></property>
    <map name="courses" table="course" lazy="false">
        <key column="sid"></key>
        <map-key type="string" column="cname"></map-key>
        <element column="chour" type="int"></element>
    </map>
</class>
```

```
public class Student {

    private int sid;
    private String sname;
    private int sage;
    private Map<String, Integer> courses = new HashMap<String, Integer>();

    public Map<String, Integer> getCourses() {
        return courses;
```

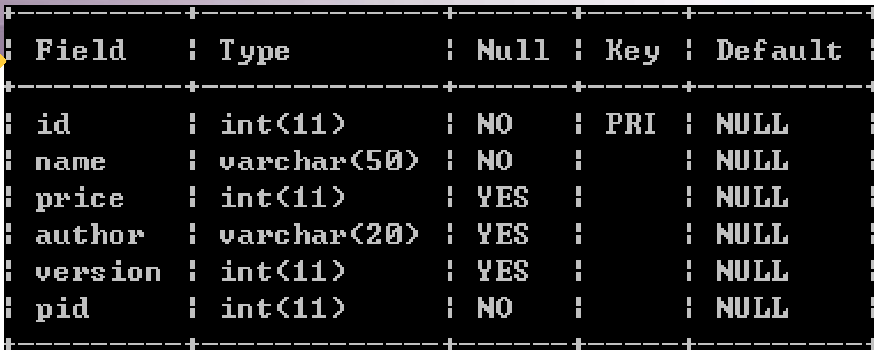
Association and Collection Mapping

- Association
  - Associations will be classified by multiplicity and whether or not they map to an intervening join table
    - unidirectional mappings
      - 1-1
      - 1-N
      - N-1
      - M-N
    - bidirectional mappings

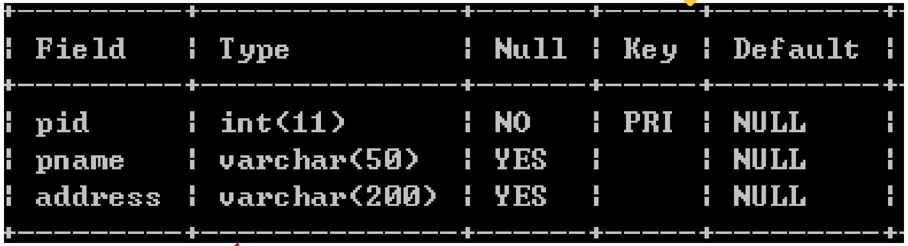
Association and Collection Mapping

- Association (cont.)
  - N-1 (Many to One)
    - Unidirectional
      - A *unidirectional many-to-one association* is the most common kind of unidirectional association
    - Bidirectional
      - A *bidirectional many-to-one association* is the most common kind of association
      - Is also a bidirectional one-to-many association

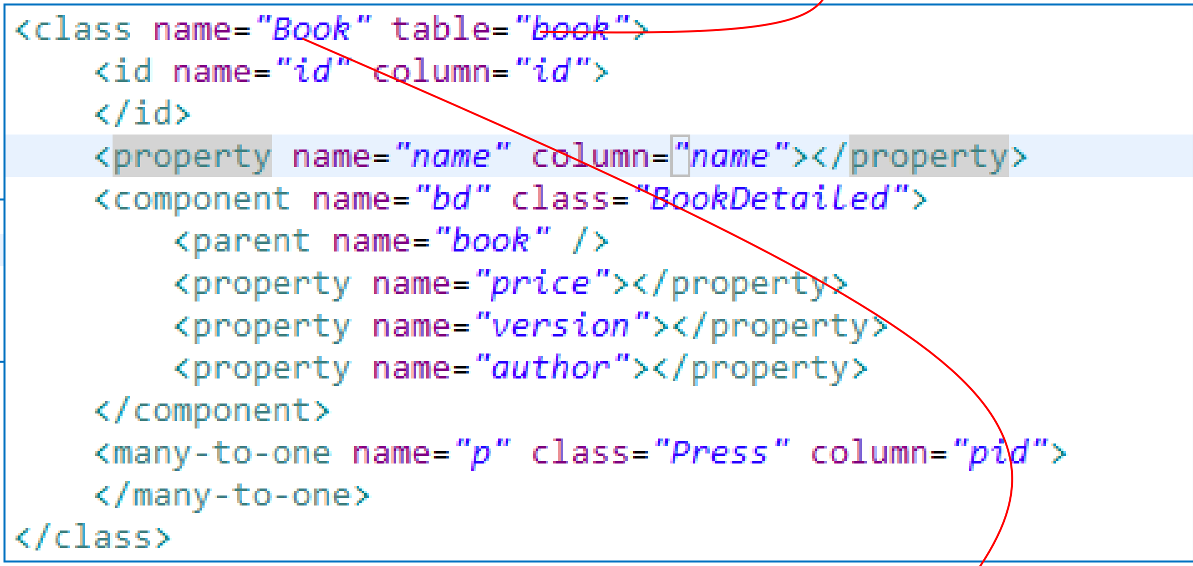
Unidirectional N-1 Demo



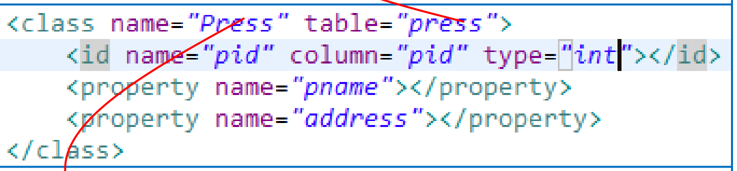
```
+---------+--------------+------+-----+---------+
| Field   | Type         | Null | Key | Default |
+---------+--------------+------+-----+---------+
| id      | int(11)      | NO   | PRI | NULL    |
| name    | varchar(50)  | NO   |     | NULL    |
| price   | int(11)      | YES  |     | NULL    |
| author  | varchar(20)  | YES  |     | NULL    |
| version | int(11)      | YES  |     | NULL    |
| pid     | int(11)      | NO   |     | NULL    |
+---------+--------------+------+-----+---------+
```



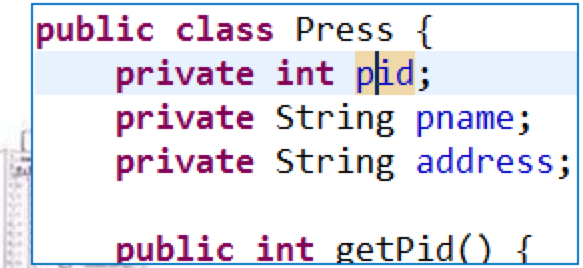
```
+---------+--------------+------+-----+---------+
| Field   | Type         | Null | Key | Default |
+---------+--------------+------+-----+---------+
| pid     | int(11)      | NO   | PRI | NULL    |
| pname   | varchar(50)  | YES  |     | NULL    |
| address | varchar(200) | YES  |     | NULL    |
+---------+--------------+------+-----+---------+
```



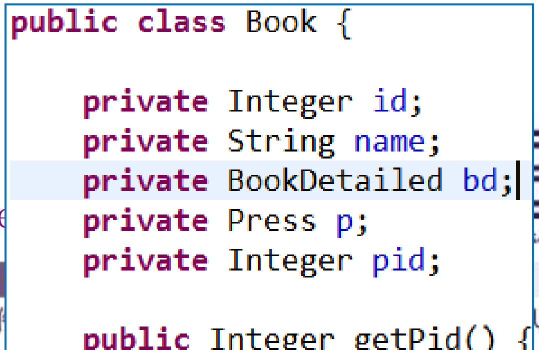
```
<class name="Book" table="book">
    <id name="id" column="id">
    </id>
    <property name="name" column="name"></property>
    <component name="bd" class="BookDetailed">
        <parent name="book" />
        <property name="price"></property>
        <property name="version"></property>
        <property name="author"></property>
    </component>
    <many-to-one name="p" class="Press" column="pid">
    </many-to-one>
</class>
```



```
<class name="Press" table="press">
    <id name="pid" column="pid" type="int"></id>
    <property name="pname"></property>
    <property name="address"></property>
</class>
```

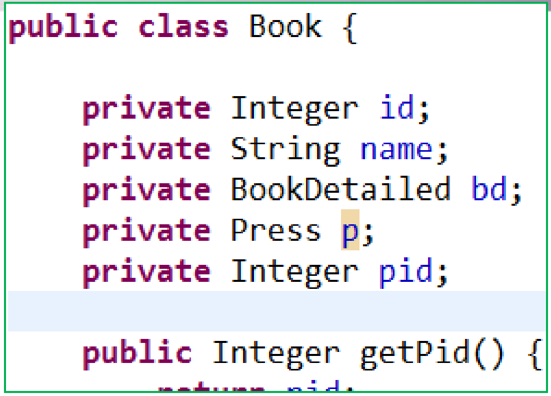


```
public class Press {
    private int pid;
    private String pname;
    private String address;

    public int getPid() {
```

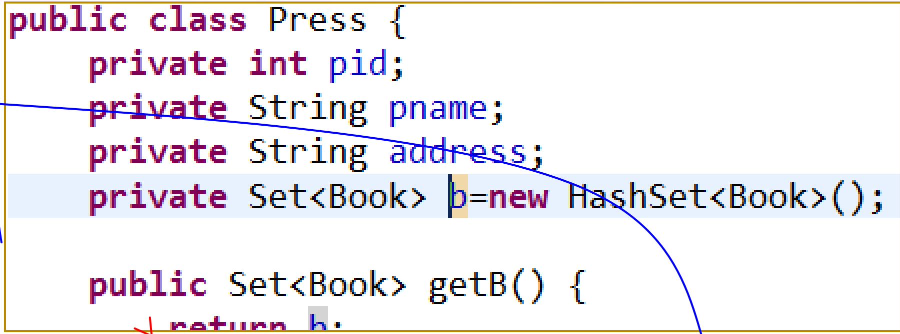


```
public class Book {

    private Integer id;
    private String name;
    private BookDetailed bd;
    private Press p;
    private Integer pid;

    public Integer getPid() {
```

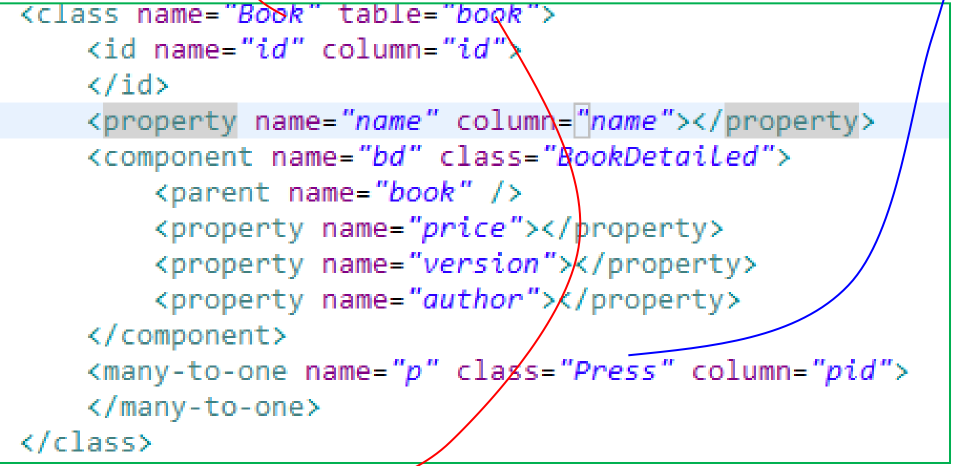
Bidirectional N-1/1-N Demo



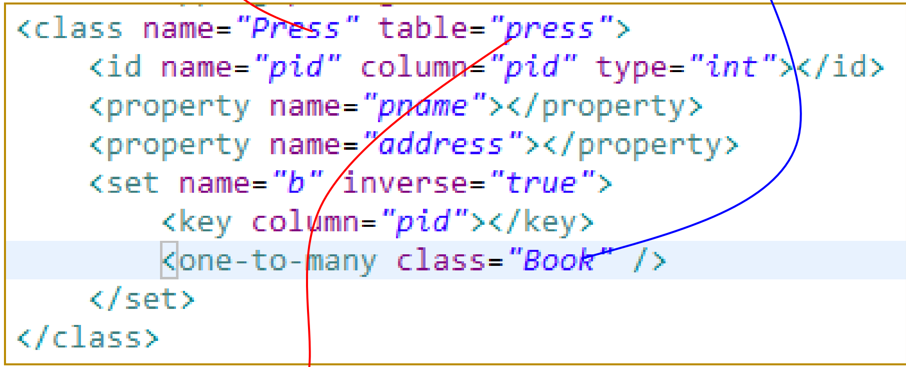
```
public class Book {

    private Integer id;
    private String name;
    private BookDetailed bd;
    private Press p;
    private Integer pid;

    public Integer getPid() {
        return pid;
```



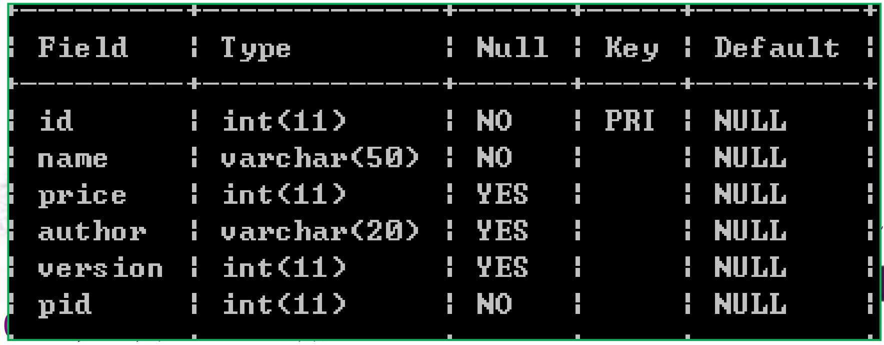
```
public class Press {
    private int pid;
    private String pname;
    private String address;
    private Set<Book> b=new HashSet<Book>();

    public Set<Book> getB() {
        return b;
```



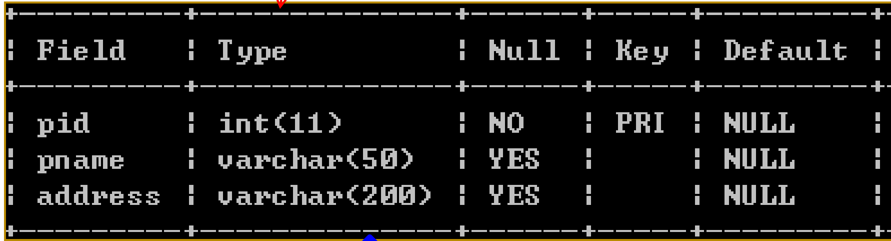
```
<class name="Book" table="book">
    <id name="id" column="id">
    </id>
    <property name="name" column="name"></property>
    <component name="bd" class="BookDetailed">
        <parent name="book" />
        <property name="price"></property>
        <property name="version"></property>
        <property name="author"></property>
    </component>
    <many-to-one name="p" class="Press" column="pid">
    </many-to-one>
</class>
```



```
<class name="Press" table="press">
    <id name="pid" column="pid" type="int"></id>
    <property name="pname"></property>
    <property name="address"></property>
    <set name="b" inverse="true">
        <key column="pid"></key>
        <one-to-many class="Book" />
    </set>
</class>
```



```
+---------+-------------+------+-----+---------+
| Field   | Type        | Null | Key | Default |
+---------+-------------+------+-----+---------+
| id      | int(11)     | NO   | PRI | NULL    |
| name    | varchar(50) | NO   |     | NULL    |
| price   | int(11)     | YES  |     | NULL    |
| author  | varchar(20) | YES  |     | NULL    |
| version | int(11)     | YES  |     | NULL    |
| pid     | int(11)     | NO   |     | NULL    |
+---------+-------------+------+-----+---------+
```

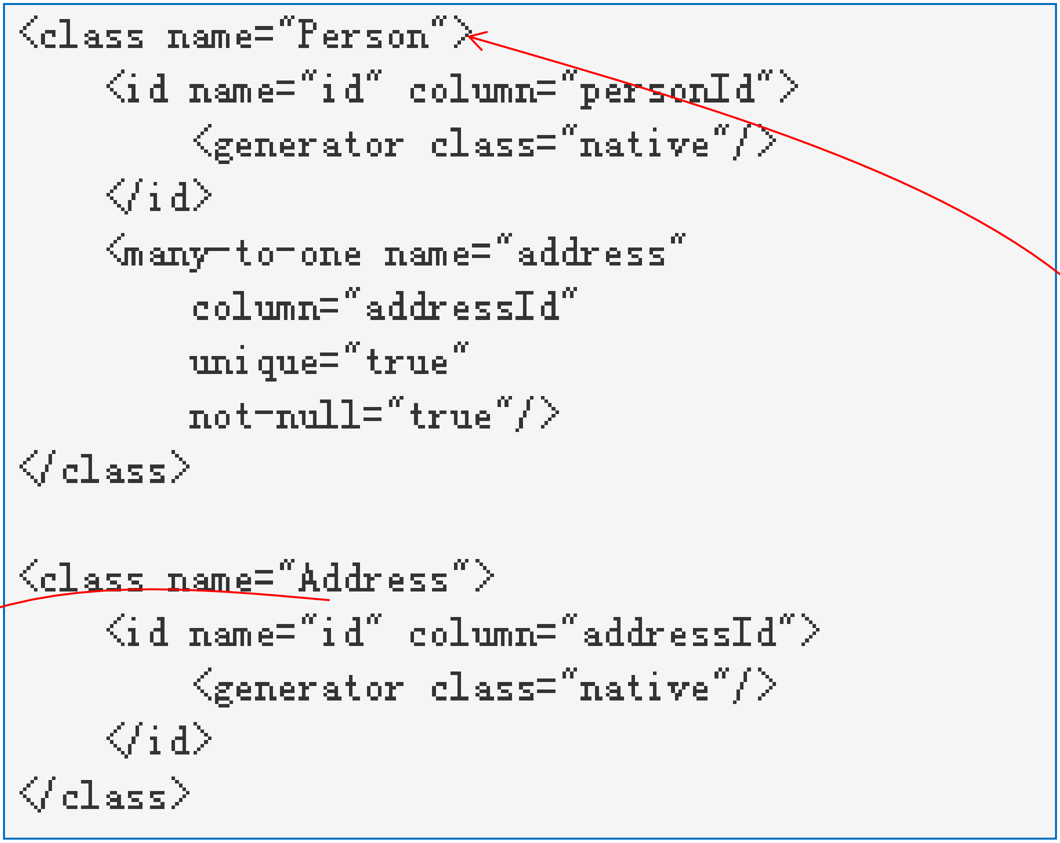


```
+---------+--------------+------+-----+---------+
| Field   | Type         | Null | Key | Default |
+---------+--------------+------+-----+---------+
| pid     | int(11)      | NO   | PRI | NULL    |
| pname   | varchar(50)  | YES  |     | NULL    |
| address | varchar(200) | YES  |     | NULL    |
+---------+--------------+------+-----+---------+
```

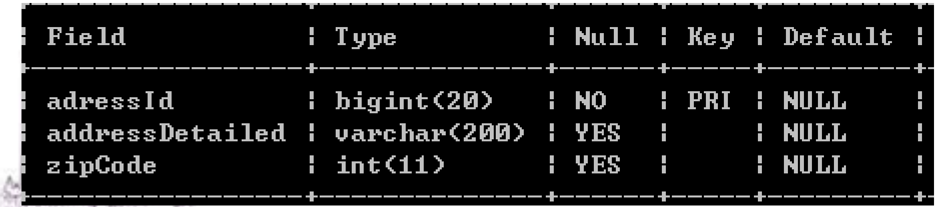
Association and Collection Mapping

- Association (cont.)
  - 1-1 (One to One)
    - Unidirectional
      - *A unidirectional one-to-one association on a foreign key* is almost identical. The only difference is the column unique constraint
      - *On a primary key* usually uses a special id generator
    - Bidirectional
      - A *bidirectional one-to-one association on a foreign key* is common
      - *On a primary key* uses the special id generator

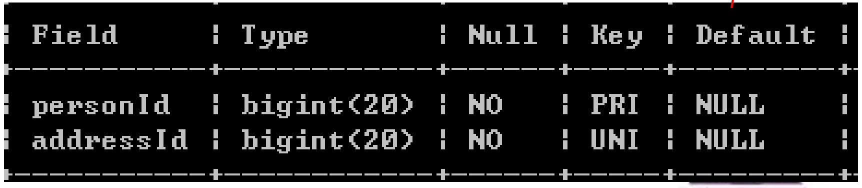
Unidirectional 1-1 on a Foreign Key Demo



```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

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

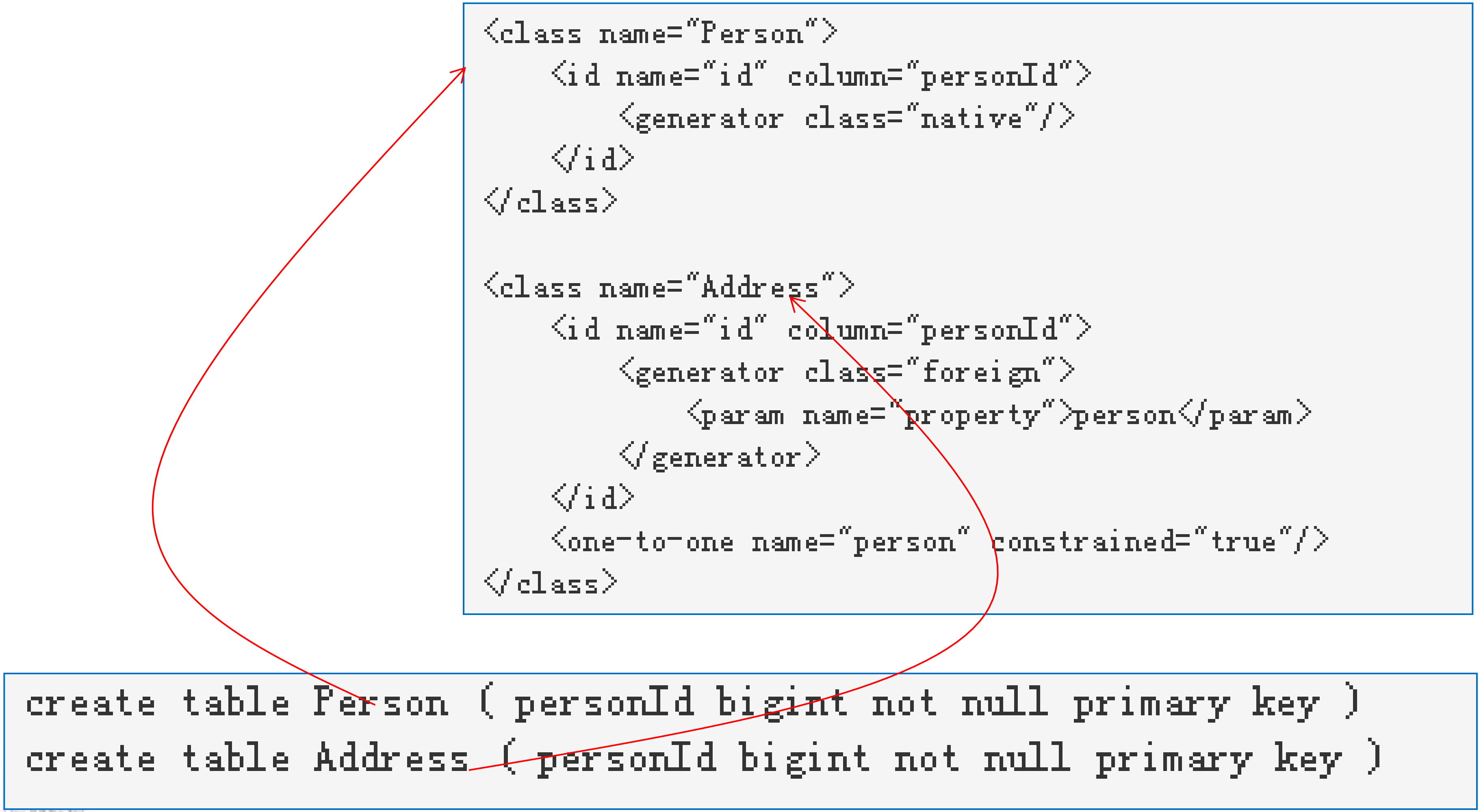


```
+----------------+--------------+------+-----+---------+
| Field          | Type         | Null | Key | Default |
+----------------+--------------+------+-----+---------+
| adressId       | bigint(20)   | NO   | PRI | NULL    |
| addressDetailed| varchar(200) | YES  |     | NULL    |
| zipCode        | int(11)      | YES  |     | NULL    |
+----------------+--------------+------+-----+---------+
```



```
+-----------+------------+------+-----+---------+
| Field     | Type       | Null | Key | Default |
+-----------+------------+------+-----+---------+
| personId  | bigint(20) | NO   | PRI | NULL    |
| addressId | bigint(20) | NO   | UNI | NULL    |
+-----------+------------+------+-----+---------+
```

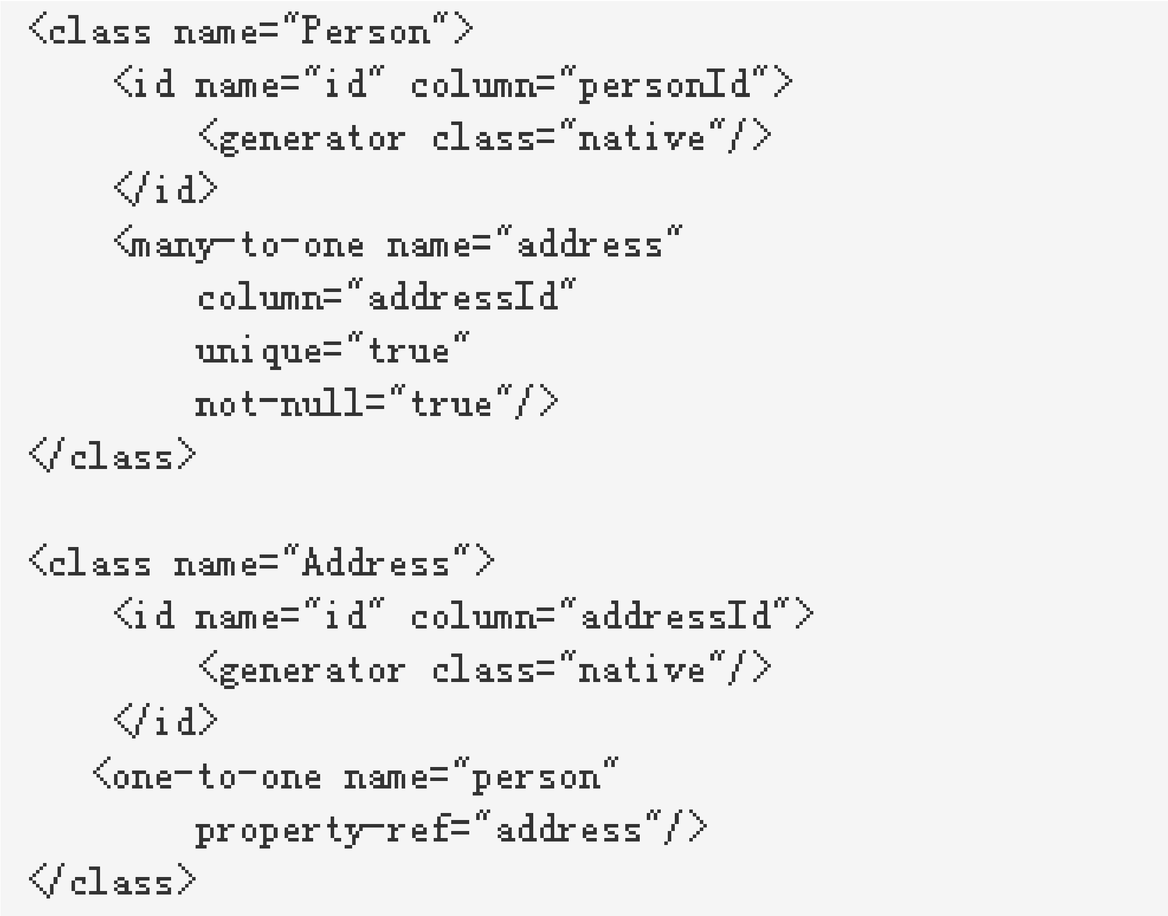
Unidirectional 1-1 on a Primary Key Demo



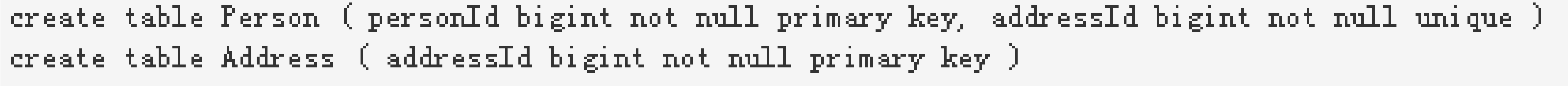
```
<class name=“Person”>
    <id name=“id” column=“personId”>
        <generator class=“native”/>
    </id>
</class>

<class name=“Address”>
    <id name=“id” column=“personId”>
        <generator class=“foreign”>
            <param name=“property”>person</param>
        </generator>
    </id>
    <one-to-one name=“person” constrained=“true”/>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

Bidirectional 1-1 on a Foreign Key Demo

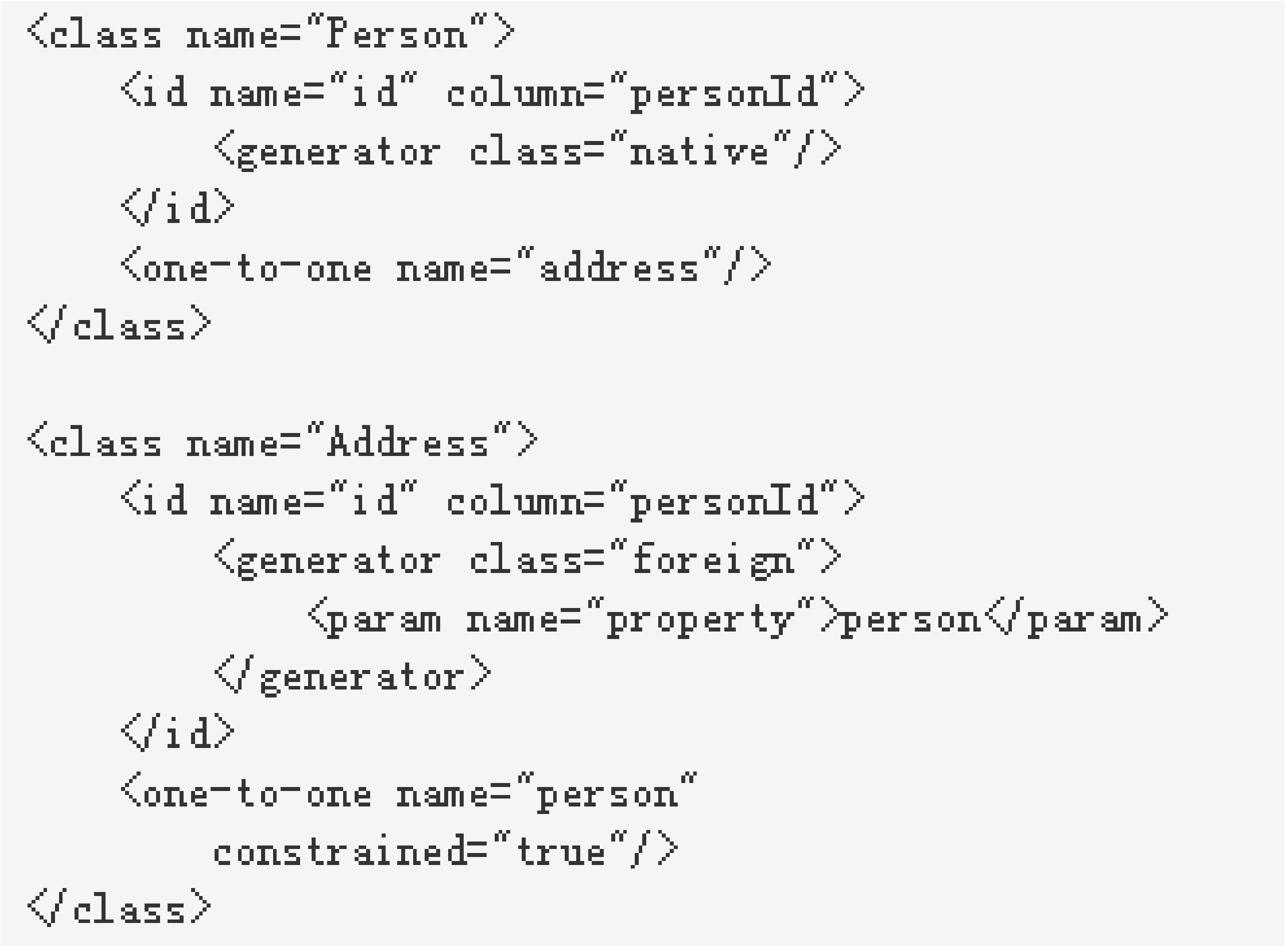


```
<class name=“Person”>
    <id name=“id” column=“personId”>
        <generator class=“native”/>
    </id>
    <many-to-one name=“address”
        column=“addressId”
        unique=“true”
        not-null=“true”/>
</class>

<class name=“Address”>
    <id name=“id” column=“addressId”>
        <generator class=“native”/>
    </id>
   <one-to-one name=“person”
        property-ref=“address”/>
</class>
```

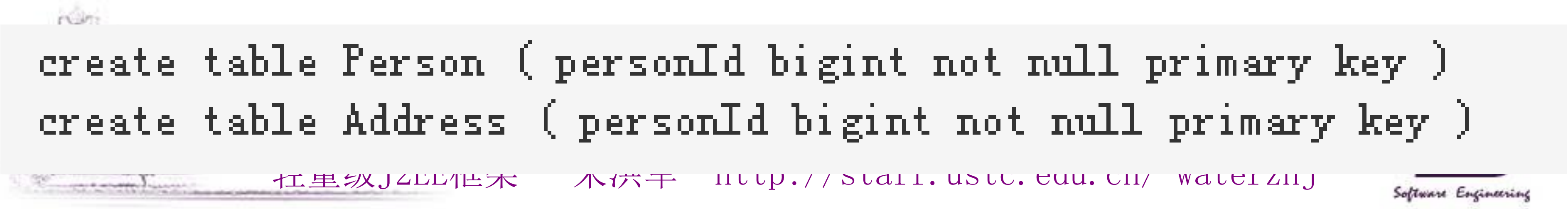


```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

Bidirectional 1-1 on a Primary Key Demo



```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <one-to-one name="address"/>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property">person</param>
        </generator>
    </id>
    <one-to-one name="person"
        constrained="true"/>
</class>
```



```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

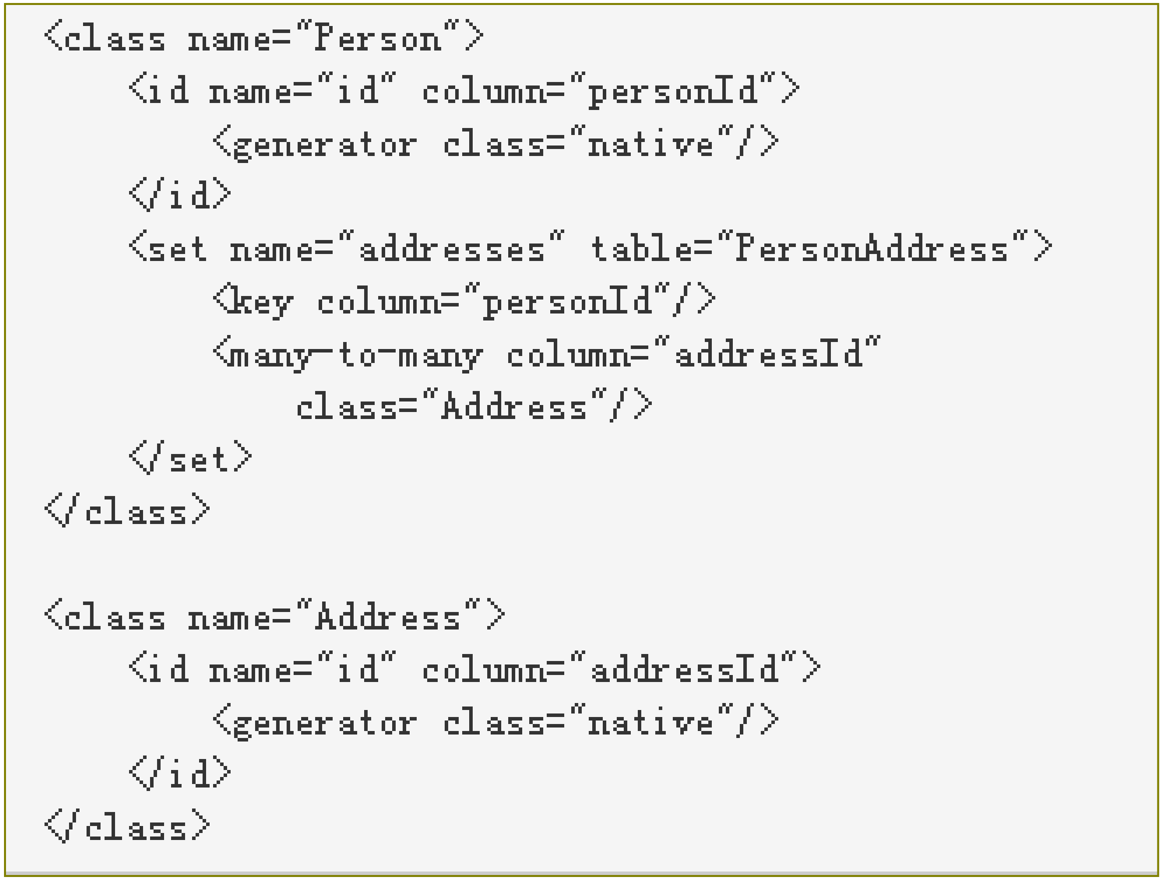
轻量级J2EE框架　宋洪军　http://staff.ustc.edu.cn/~waterzhj

Software Engineering

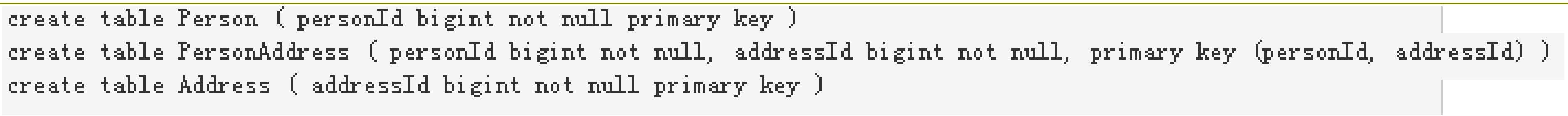
Association and Collection Mapping

- Association (cont.)
  - M-N (Many to Many)
    - Unidirectional
      - A unidirectional many to many association needs a join table
    - Bidirectional
      - A bidirectional many to many association also needs a join table

Unidirectional M-N on a Join Table Demo

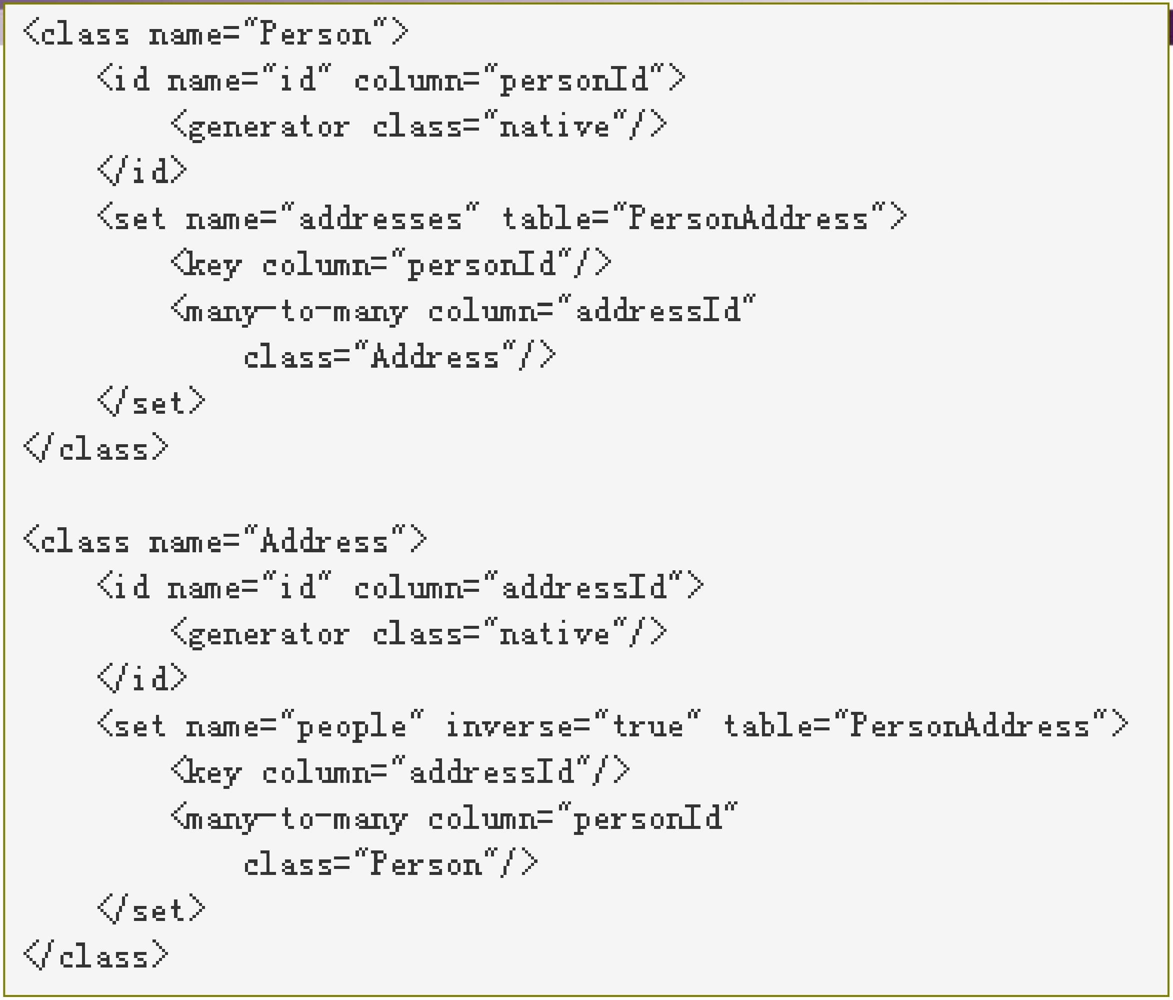


```
<class name=“Person”>
    <id name=“id” column=“personId”>
        <generator class=“native”/>
    </id>
    <set name=“addresses” table=“PersonAddress”>
        <key column=“personId”/>
        <many-to-many column=“addressId”
            class=“Address”/>
    </set>
</class>

<class name=“Address”>
    <id name=“id” column=“addressId”>
        <generator class=“native”/>
    </id>
</class>
```

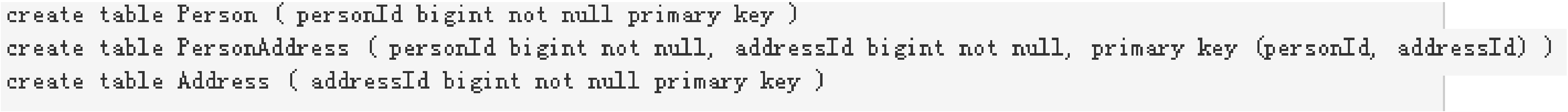


```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

Bidirectional M-N on a Join Table Demo



```
<class name=“Person”>
    <id name=“id” column=“personId”>
        <generator class=“native”/>
    </id>
    <set name=“addresses” table=“PersonAddress”>
        <key column=“personId”/>
        <many-to-many column=“addressId”
            class=“Address”/>
    </set>
</class>

<class name=“Address”>
    <id name=“id” column=“addressId”>
        <generator class=“native”/>
    </id>
    <set name=“people” inverse=“true” table=“PersonAddress”>
        <key column=“addressId”/>
        <many-to-many column=“personId”
            class=“Person”/>
    </set>
</class>
```

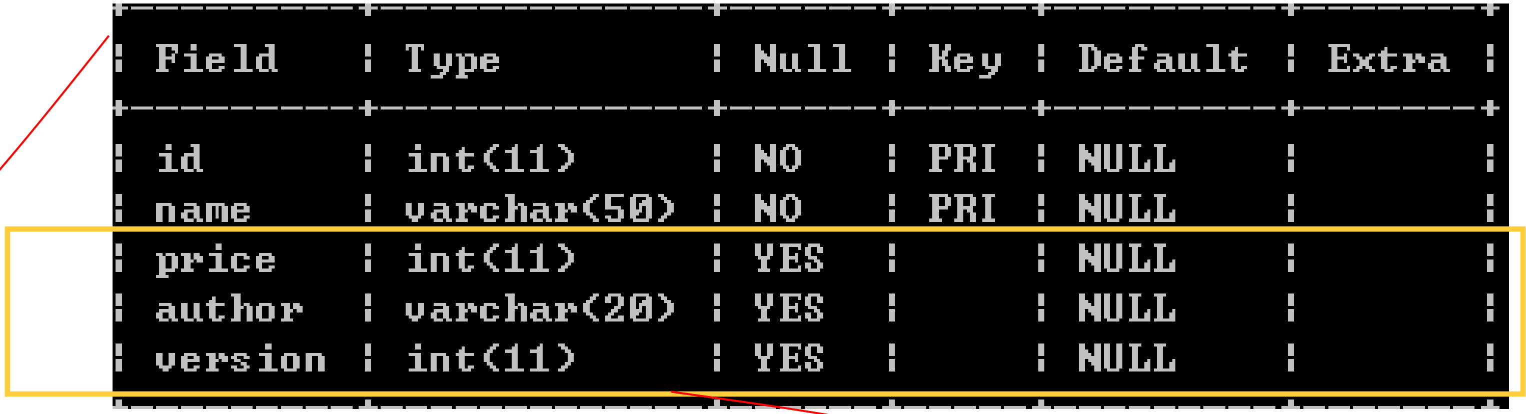


```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

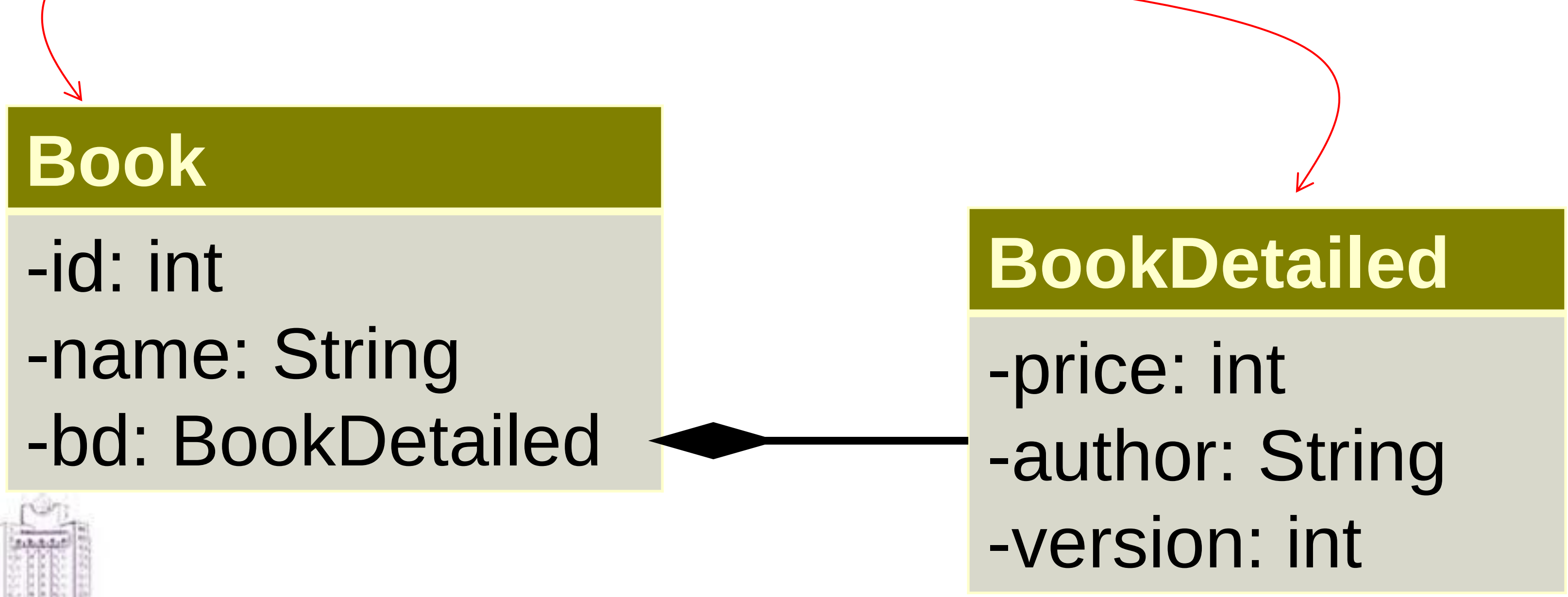
Component and Inheritance Mapping

- Component Mapping
  - Dependent Objects
    - A component is a contained object that is persisted as a value type and not an entity reference
    - Components do not support shared references
    - Using <component/>
      - The <component> element allows a <parent> subelement that maps a property of the component class as a reference back to the containing entity

Entity/Component Demo



```
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| id      | int(11)     | NO   | PRI | NULL    |       |
| name    | varchar(50) | NO   | PRI | NULL    |       |
| price   | int(11)     | YES  |     | NULL    |       |
| author  | varchar(20) | YES  |     | NULL    |       |
| version | int(11)     | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
```



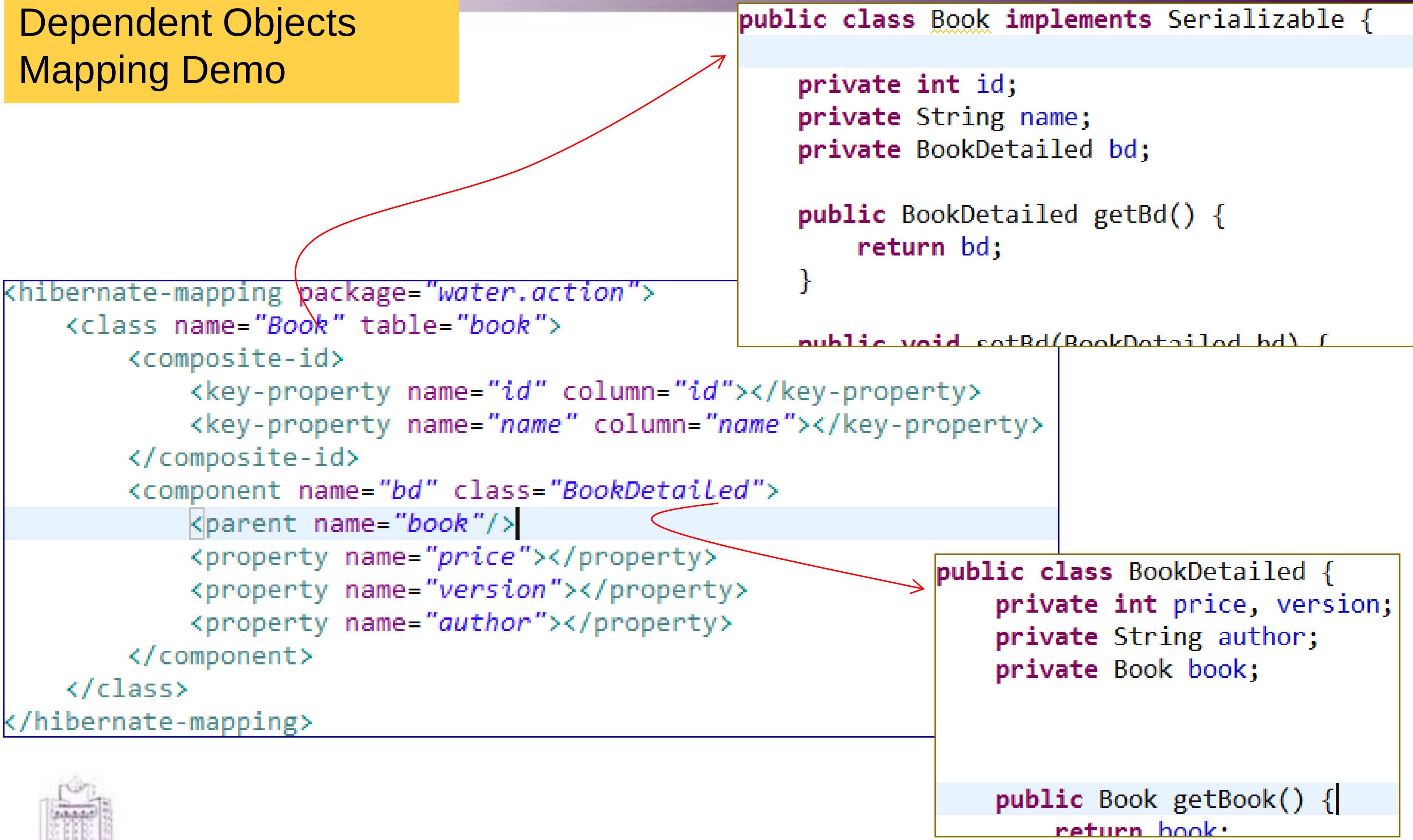
Book

-id: int
-name: String
-bd: BookDetailed

BookDetailed

-price: int
-author: String
-version: int

Dependent Objects Mapping Demo



```
public class Book implements Serializable {

    private int id;
    private String name;
    private BookDetailed bd;

    public BookDetailed getBd() {
        return bd;
    }

    public void setBd(BookDetailed bd) {
```

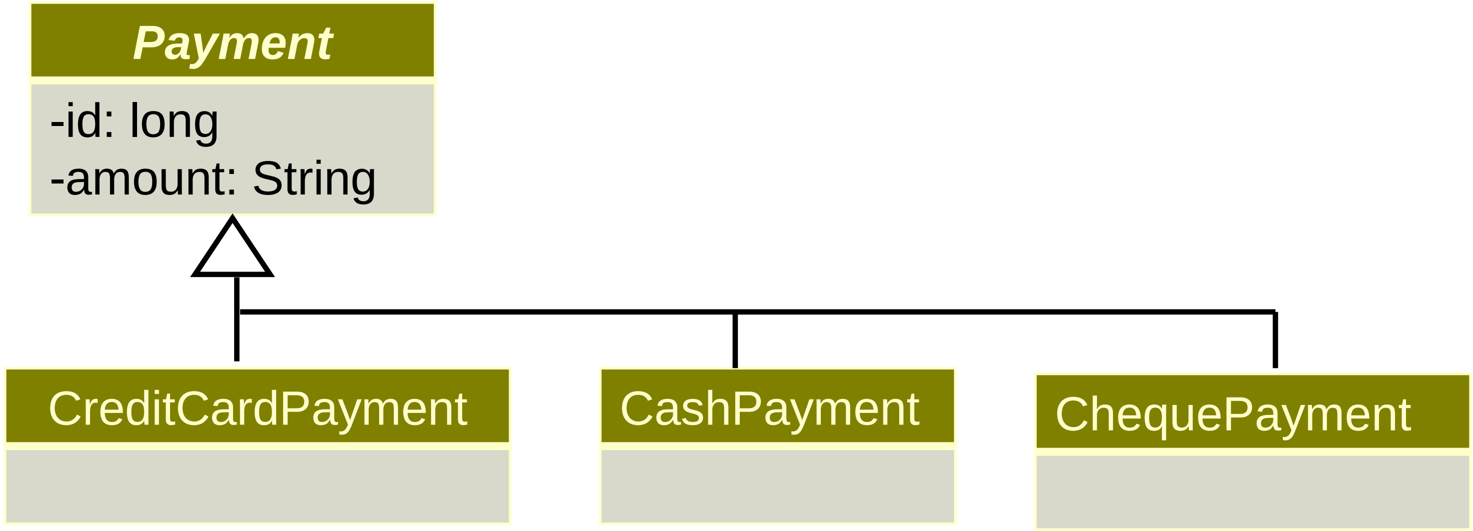
```xml
<hibernate-mapping package="water.action">
    <class name="Book" table="book">
        <composite-id>
            <key-property name="id" column="id"></key-property>
            <key-property name="name" column="name"></key-property>
        </composite-id>
        <component name="bd" class="BookDetailed">
            <parent name="book"/>
            <property name="price"></property>
            <property name="version"></property>
            <property name="author"></property>
        </component>
    </class>
</hibernate-mapping>
```

```
public class BookDetailed {
    private int price, version;
    private String author;
    private Book book;



    public Book getBook() {
        return book;
```

Component and Inheritance Mapping

- Inheritance Mapping
  - Hibernate supports the three basic inheritance mapping strategies
    - single table per class hierarchy
    - table per subclass
    - table per concrete class
  - In addition, Hibernate supports a fourth, slightly different kind of polymorphism
    - implicit polymorphism

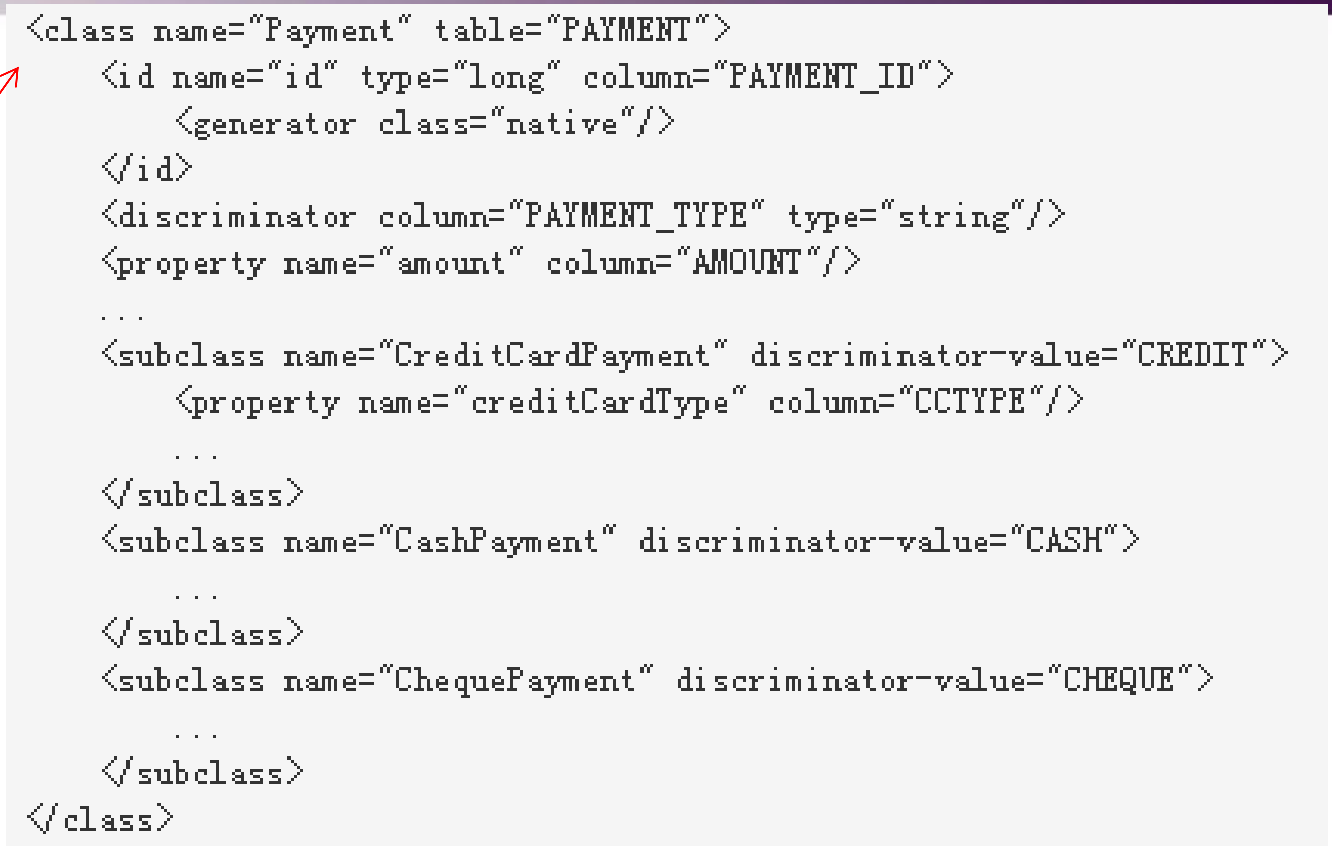
Suppose we have an Abstract class/interface Payment with the implementors CreditCardPayment, CashPayment, andChequePayment



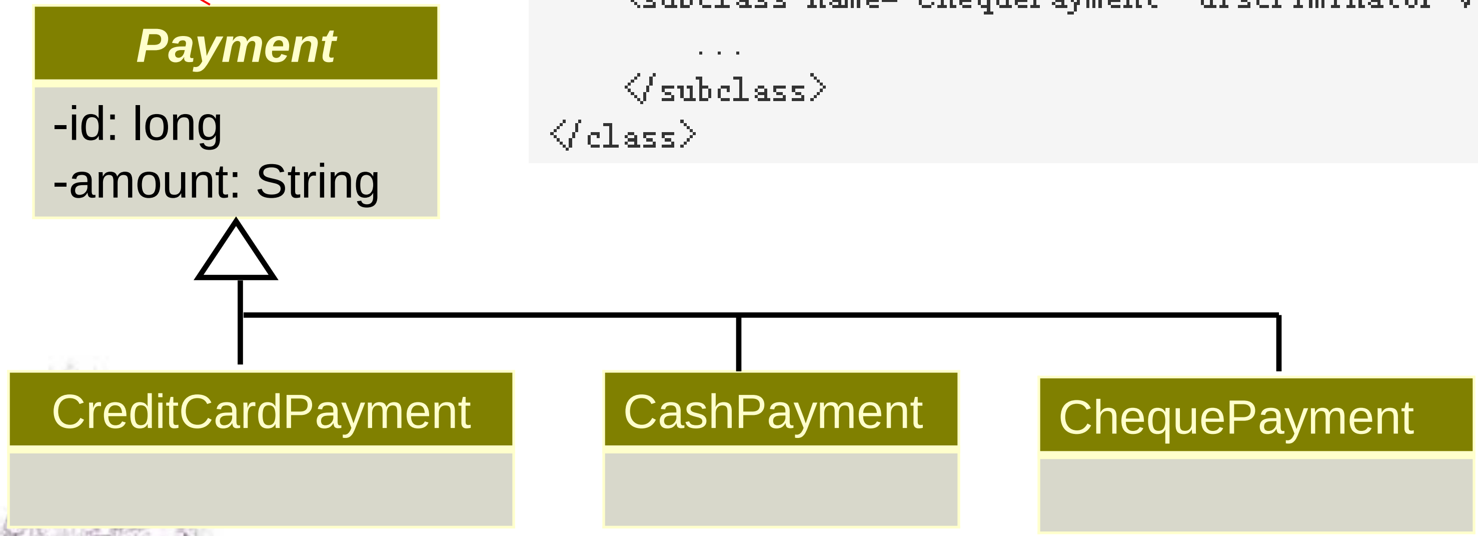
Component and Inheritance Mapping

- Single Table Per Class Hirarchy Mapping
  - If we have an abstract class/interface with the implementors. Only one table is required to map those implementors
  - There is a limitation of this mapping strategy: columns declared by the subclasses, such as CCTYPE, cannot have NOT NULL constraints
  - Use <subclass> in hbm.xml

Single Table Per Class Hirarchy Mapping Demo



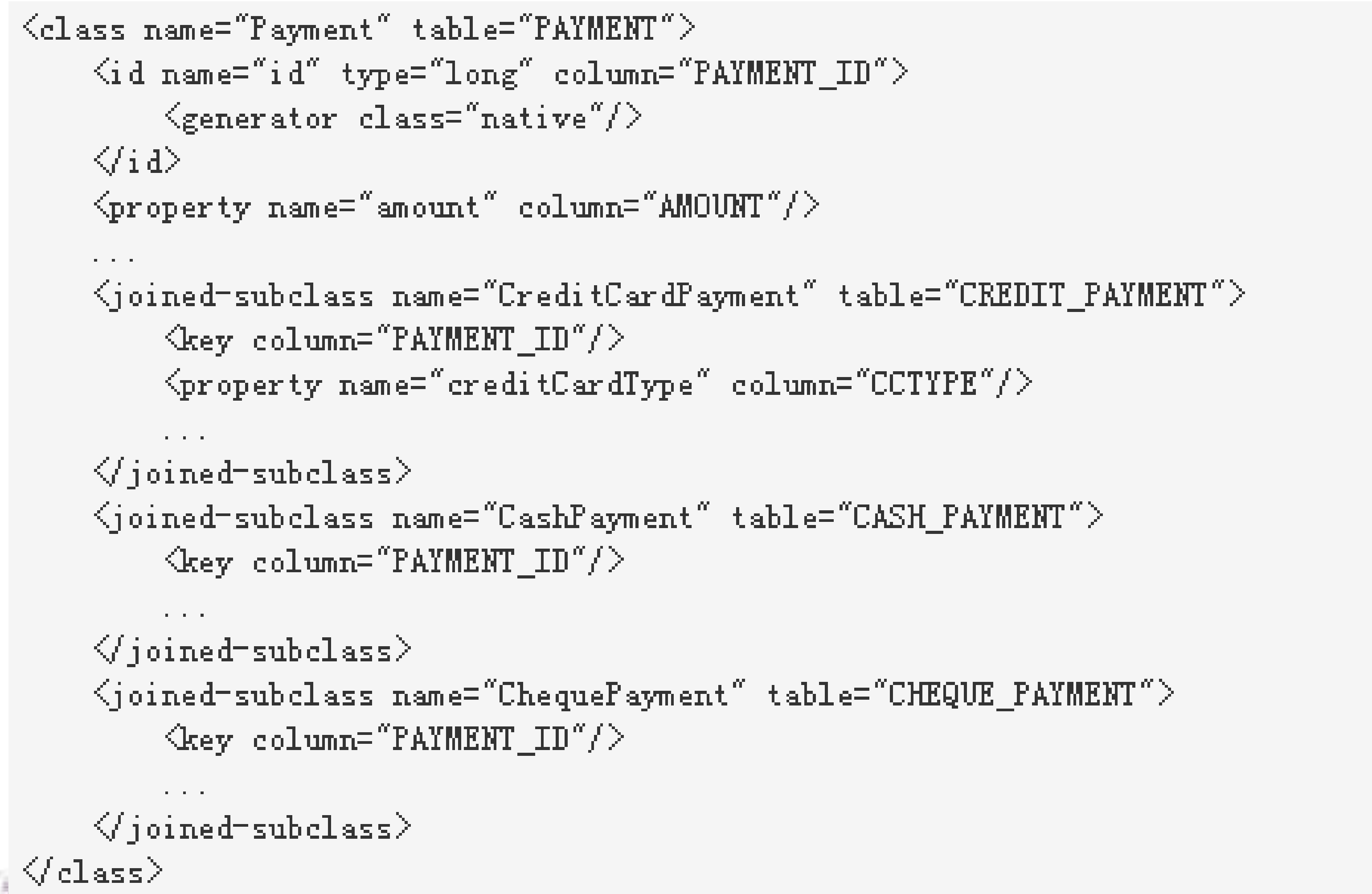
```
<class name=“Payment” table=“PAYMENT”>
    <id name=“id” type=“long” column=“PAYMENT_ID”>
        <generator class=“native”/>
    </id>
    <discriminator column=“PAYMENT_TYPE” type=“string”/>
    <property name=“amount” column=“AMOUNT”/>
    ...
    <subclass name=“CreditCardPayment” discriminator-value=“CREDIT”>
        <property name=“creditCardType” column=“CCTYPE”/>
        ...
    </subclass>
    <subclass name=“CashPayment” discriminator-value=“CASH”>
        ...
    </subclass>
    <subclass name=“ChequePayment” discriminator-value=“CHEQUE”>
        ...
    </subclass>
</class>
```



Component and Inheritance Mapping

- Table Per Subclass Mapping
  - Each class require a table
  - Subclass tables have primary key associations to the superclass table. So the relational model is actually a one-to-one association
  - Use <joined-subclass> in hbm.xml

Table Per Subclass Mapping Demo

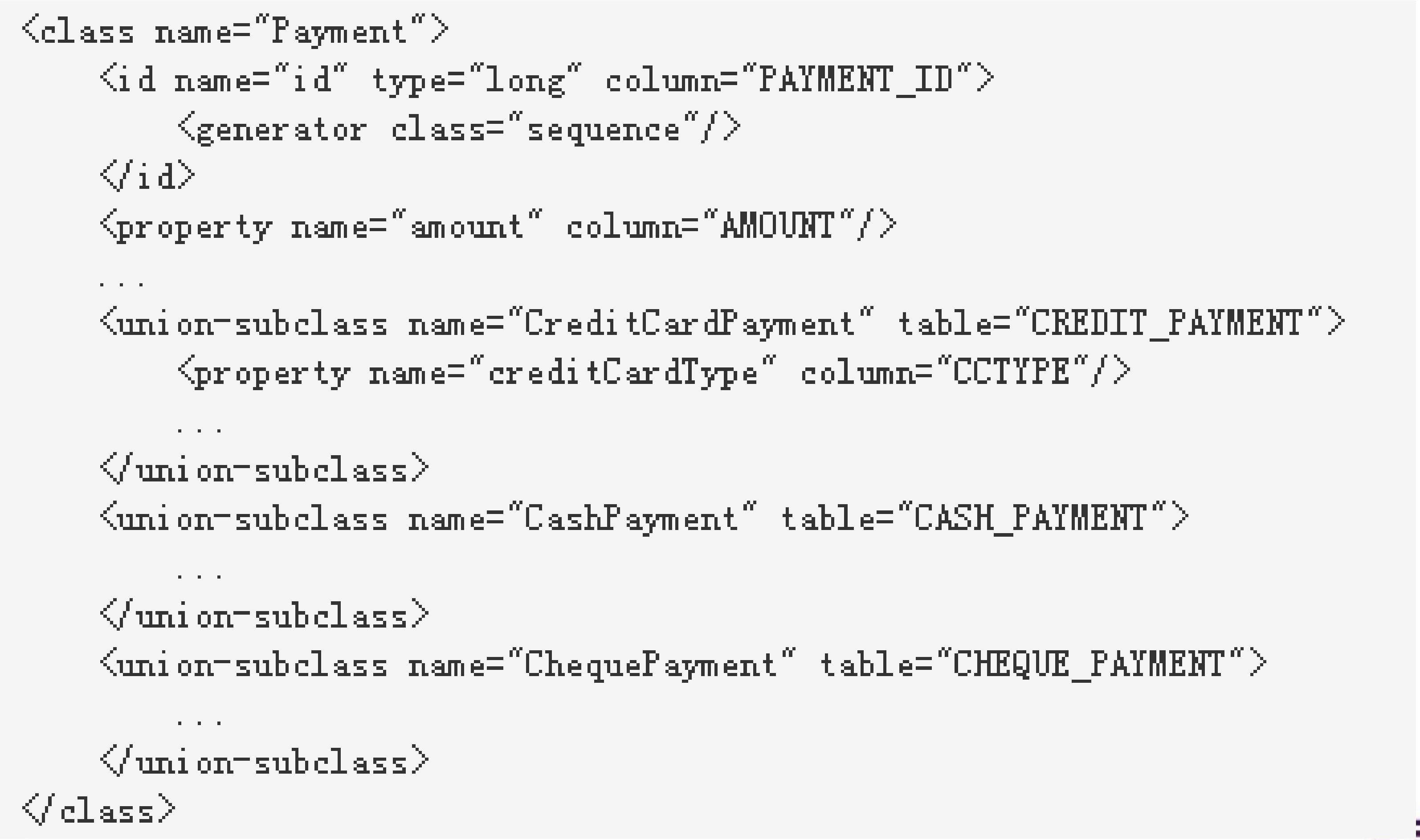


```
<class name=“Payment” table=“PAYMENT”>
    <id name=“id” type=“long” column=“PAYMENT_ID”>
        <generator class=“native”/>
    </id>
    <property name=“amount” column=“AMOUNT”/>
    ...
    <joined-subclass name=“CreditCardPayment” table=“CREDIT_PAYMENT”>
        <key column=“PAYMENT_ID”/>
        <property name=“creditCardType” column=“CCTYPE”/>
        ...
    </joined-subclass>
    <joined-subclass name=“CashPayment” table=“CASH_PAYMENT”>
        <key column=“PAYMENT_ID”/>
        ...
    </joined-subclass>
    <joined-subclass name=“ChequePayment” table=“CHEQUE_PAYMENT”>
        <key column=“PAYMENT_ID”/>
        ...
    </joined-subclass>
</class>
```

Component and Inheritance Mapping

- Table Per Concrete Class Mapping
  - There are two ways we can map the table per concrete class strategy
  - First, you can use<union-subclass>
    - Each table defines columns for all properties of the class, including inherited properties
  - An alternative approach is to make use of implicit polymorphism
    - Be deprecated

Table Per Concrete Class Mapping by <union-subclass> Demo



```
<class name=“Payment”>
    <id name=“id” type=“long” column=“PAYMENT_ID”>
        <generator class=“sequence”/>
    </id>
    <property name=“amount” column=“AMOUNT”/>
    ...
    <union-subclass name=“CreditCardPayment” table=“CREDIT_PAYMENT”>
        <property name=“creditCardType” column=“CCTYPE”/>
        ...
    </union-subclass>
    <union-subclass name=“CashPayment” table=“CASH_PAYMENT”>
        ...
    </union-subclass>
    <union-subclass name=“ChequePayment” table=“CHEQUE_PAYMENT”>
        ...
    </union-subclass>
</class>
```

Conclusions

- Refresher in Enterprise Application Architectures
- Traditional Persistence and Hibernate
- Basic O/R Mapping
- Association and Collection Mapping
- Component and Inheritance Mapping