Hibernate Fundamentals

Lam Tang
Principle Software Engineer

Jun 2016

Course Objectives

- At the end of the course, you will have acquired sufficient knowledge to:

  You can understand and working on the hibernate.

Agenda

Course Audience and Prerequisite

- Basic understanding of Java
- Prior exposure to SQL can be helpful, but its not a pre-requisite for the course

Assessment Disciplines

- Class Participation: at least 100% of course time
- Final Exam: 70%

Duration and Course Timetable

- Course Duration: 6 hrs

Course Administration

- In order to complete the course you must:
  - Sign in the Class Attendance List
  - Participate in the course
  - Provide your feedback in the End of Course Evaluation

Introduction to Hibernate

What and why is Hibernate?

- Hibernate ORM (object-relational mapping) is an object-relational mapping framework for the Java language. It provides a framework for mapping an object-oriented domain model to a relational database. Hibernate solves object-relational impedance mismatch problems by replacing direct, persistent database accesses with high-level object handling functions.
- Hibernate is database independent
- Object-relational mapping, you will map a database table with java object called "Entity".
- Caching mechanism
- Supports **Lazy loading**

Programming relate to relational database

- What do relational DBs do well?
  - Work with large amount of data
    - Searching, sorting
  - Work with sets of data
    - Joining, aggregating
  - Sharing
    - Concurrency (Transactions)
  - Integrity
    - Constrains

Programming relate to relational database

- What do relational DBs do badly?
  - Modeling
    - No polymorphism/inheritance
    - No support for automatic conversion to objects
  - Business logic
    - There ‘s stored procedures, but:
      - Very database specific
      - Very coupled with data, really belongs in the application domain

Java Persistence

- Storing Java objects to relational databases
- Persistent classes:
  - Persistent classes are classes in an application that implement the entities of the business problem (ex: Customer, Product…)

The Hibernate solution

- Hibernate is framework for mapping an object-oriented domain model to a relational database
- Easy to develop:
  - Persistent classes (entities) are POJOs
    - Easy to write and refactor
    - Can be serialized
    - Can execute outside the container (Junit)
  - POJO programming model
    - Persistent properties are not abstract
    - Can instantiate POJOs using new()
  - Session Interface
    - Session interface is provided for persistence operations

The Hibernate solution

- Convenience:
  - Truly object-oriented
    - Inheritance
    - Polymorphism
    - Association
    - Collections API for “many” relationships
- Reduce application code.
- Improve performance

Hibernate development history

- Hibernate was started in 2001 by Gavin King with colleagues from Cirrus Technologies as an alternative to using EJB2-style entity beans..
- In early 2003, the Hibernate development team began Hibernate2 releases, which offered many significant improvements over the first release.
- JBoss, Inc. (now part of Red Hat) later hired the lead Hibernate developers in order to further its development.
- In 2005, Hibernate version 3.0 was released. Key features included a new Interceptor/Callback architecture, user defined filters, and JDK 5.0 Annotations (Java's metadata feature). As of 2010, Hibernate 3 (version 3.5.0 and up) was a certified implementation of the Java Persistence API 2.0 specification via a wrapper for the Core module which provides conformity with the JSR 317 standard.[2]

Hibernate development history

- In Dec 2011, Hibernate Core 4.0.0 Final was released. This includes new features such as multi-tenancy support, introduction of ServiceRegistry (a major change in how Hibernate builds and manages "services"), better session opening from SessionFactory, improved integration via *org.hibernate.integrator.spi.Integrator* and auto discovery, internationalization support, message codes in logging, and a more distinction between the API, SPI or implementation classes.[3]
- In Dec 2012, Hibernate ORM 4.1.9 Final was released.[4]
- In Mar 2013, Hibernate ORM 4.2 Final was released.[5]
- In Dec 2013, Hibernate ORM 4.3.0 Final was released.[6] It features Java Persistence API 2.1.[7]
- In Sep 2015, Hibernate ORM 5.0.2 Final was released. It has improved bootstrapping, hibernate-java8, hibernate-spatial, Karaf support.

Points to remember

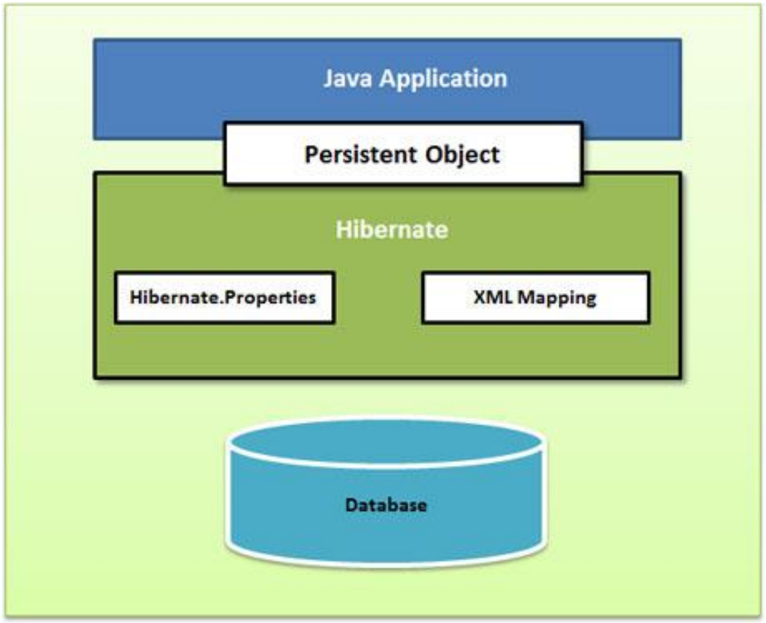
Hibernate architecture

Hibernate architecture

- Hibernate architecture overview
- Hibernate main classes and interface API
- Working with Session interface

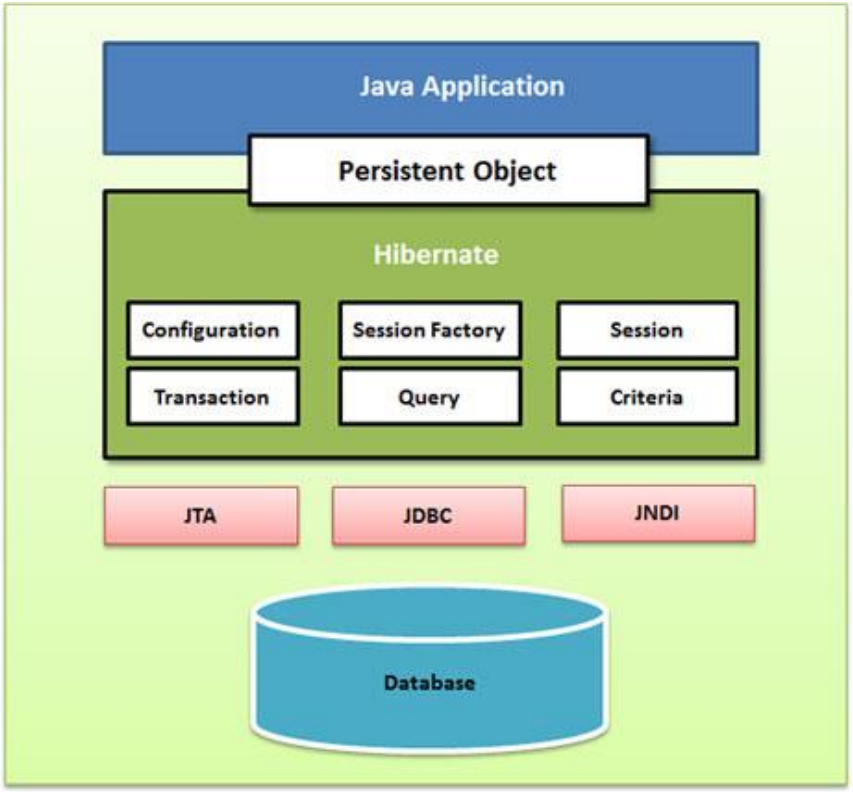
Hibernate architecture overview

- High-level view of the Hibernate architecture



Hibernate architecture overview

– Hibernate abstracts the application away from the underlying JDBC/JTA APIs and allows Hibernate to manage the details



Hibernate architecture overview

- Hibernate architecture has three main components:
  - **Connection Management**
    - Provides efficient management of the database connections.
  - **Transaction management**
    - Provides the ability to the user to execute more than one database statements at a time.
  - **Object relational mapping**
    - Is a technique of mapping the data representation from an object model to a relational data model.

Hibernate main classes and interface API

- The main Hibernate API are given below:

- org.hibernate.Hibernate	- org.hibernate.Criteria
- org.hibernate.cfg.Configuration	- org.hibernate.ScrollableResults
- org.hibernate.SessionFactory	- org.hibernate.expression.Expression
- org.hibernate.Session	- org.hibernate.Query
- org.hibernate.Transaction	- org.hibernate.expression.Order

Hibernate main classes and interface API

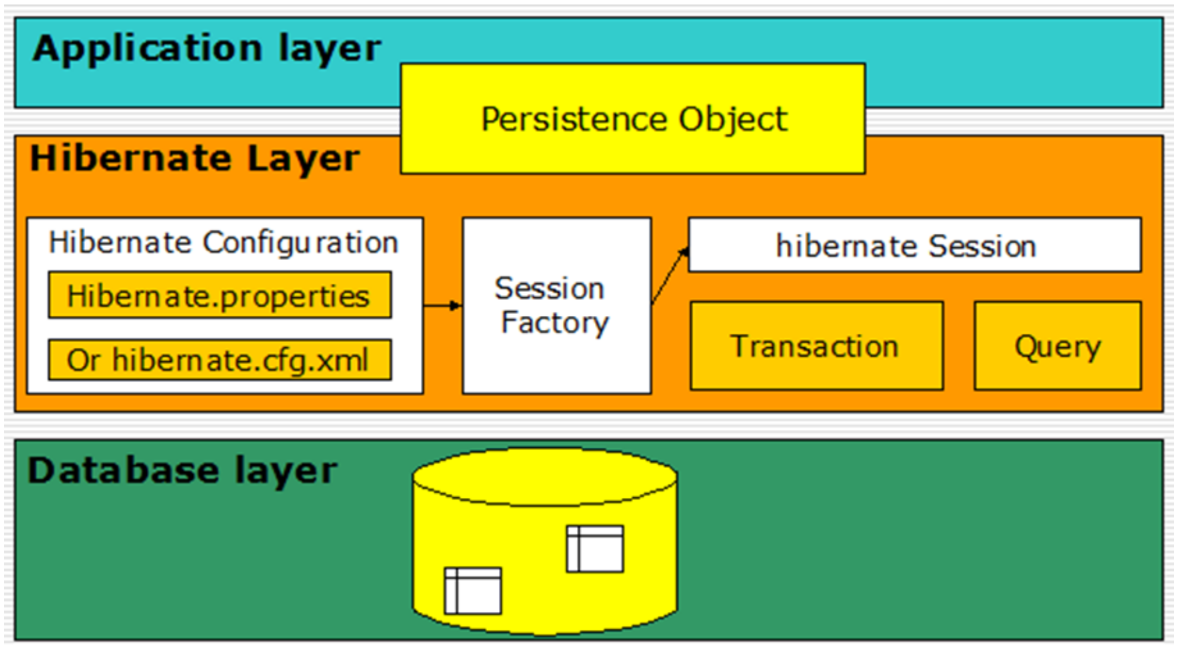
- Configuration
  - Reads and establishes the properties that Hibernate uses to connect to a database and configure itself for work.
    - Database Connection
    - Class Mapping Setup
  - Used to create a SessionFactory and then is typically discarded.
- SessionFactory
  - The SessionFactory object is a factory for Session objects and is an expensive object to create.
  - It is usually created once during application start up like the Configuration object and retained for later use
- Session
  - Provides the main interface to accomplish work with the database.
  - A Session object is lightweight and inexpensive to create
  - A Session object establishes a physical connection to the database.

Hibernate main classes and interface API

- Transaction
  - The Hibernate Transaction object absolves the developer from having to deal with the underlying transaction manager/transaction.
- Query and Criteria
  - Query and Criteria objects are used to retrieve (and recreate) persistent objects

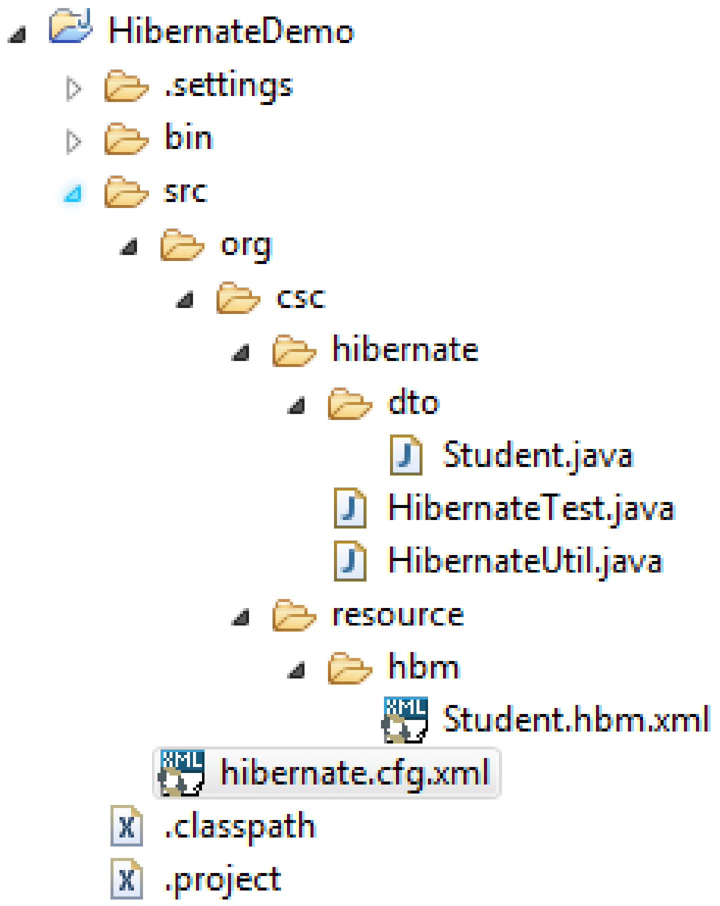
Hibernate main classes and interface API

- Classes and interfaces illustrations



Hibernate main classes and interface API

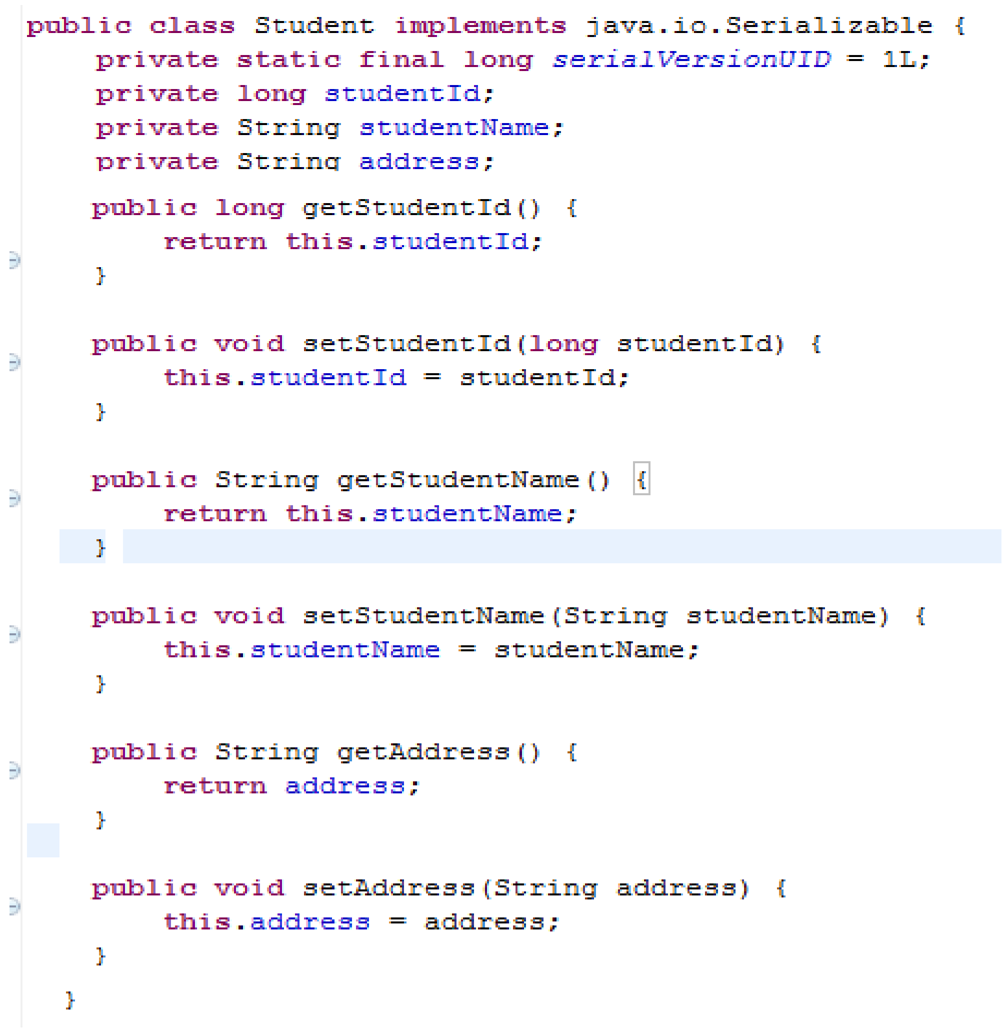
- Structure:



```
HibernateDemo
    .settings
    bin
    src
        org
            csc
                hibernate
                    dto
                        Student.java
                    HibernateTest.java
                    HibernateUtil.java
                resource
                    hbm
                        Student.hbm.xml
        hibernate.cfg.xml
    .classpath
    .project
```

Hibernate main classes and interface API

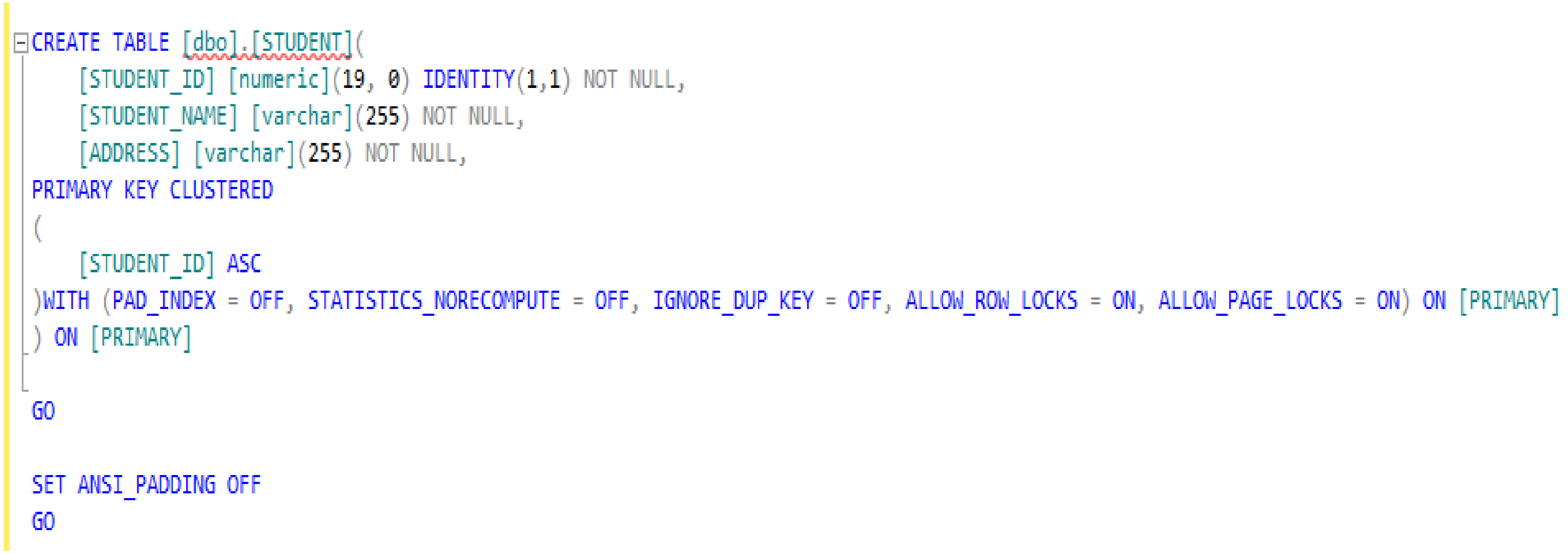
- Step 1: Create a POJO with the name Student



```
public class Student implements java.io.Serializable {
    private static final long serialVersionUID = 1L;
    private long studentId;
    private String studentName;
    private String address;

    public long getStudentId() {
        return this.studentId;
    }

    public void setStudentId(long studentId) {
        this.studentId = studentId;
    }

    public String getStudentName() {
        return this.studentName;
    }

    public void setStudentName(String studentName) {
        this.studentName = studentName;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
}
```

Hibernate main classes and interface API

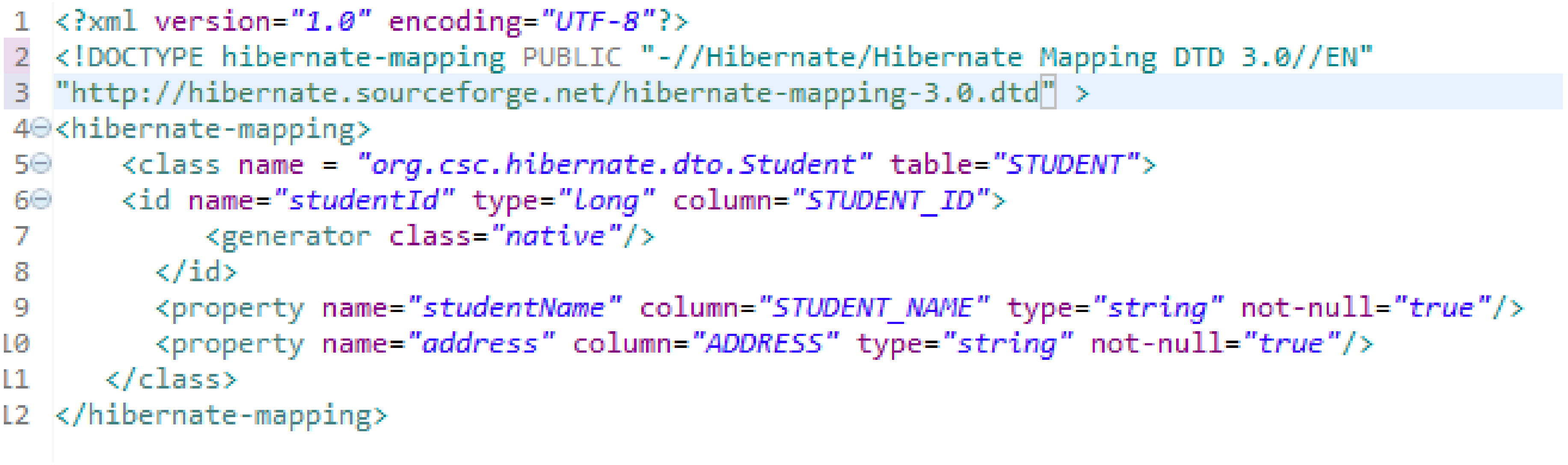
- Step 2: Create a table in SQL database with the name Student



```
CREATE TABLE [dbo].[STUDENT](
    [STUDENT_ID] [numeric](19, 0) IDENTITY(1,1) NOT NULL,
    [STUDENT_NAME] [varchar](255) NOT NULL,
    [ADDRESS] [varchar](255) NOT NULL,
PRIMARY KEY CLUSTERED
(
    [STUDENT_ID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]

GO

SET ANSI_PADDING OFF
GO
```

Hibernate main classes and interface API

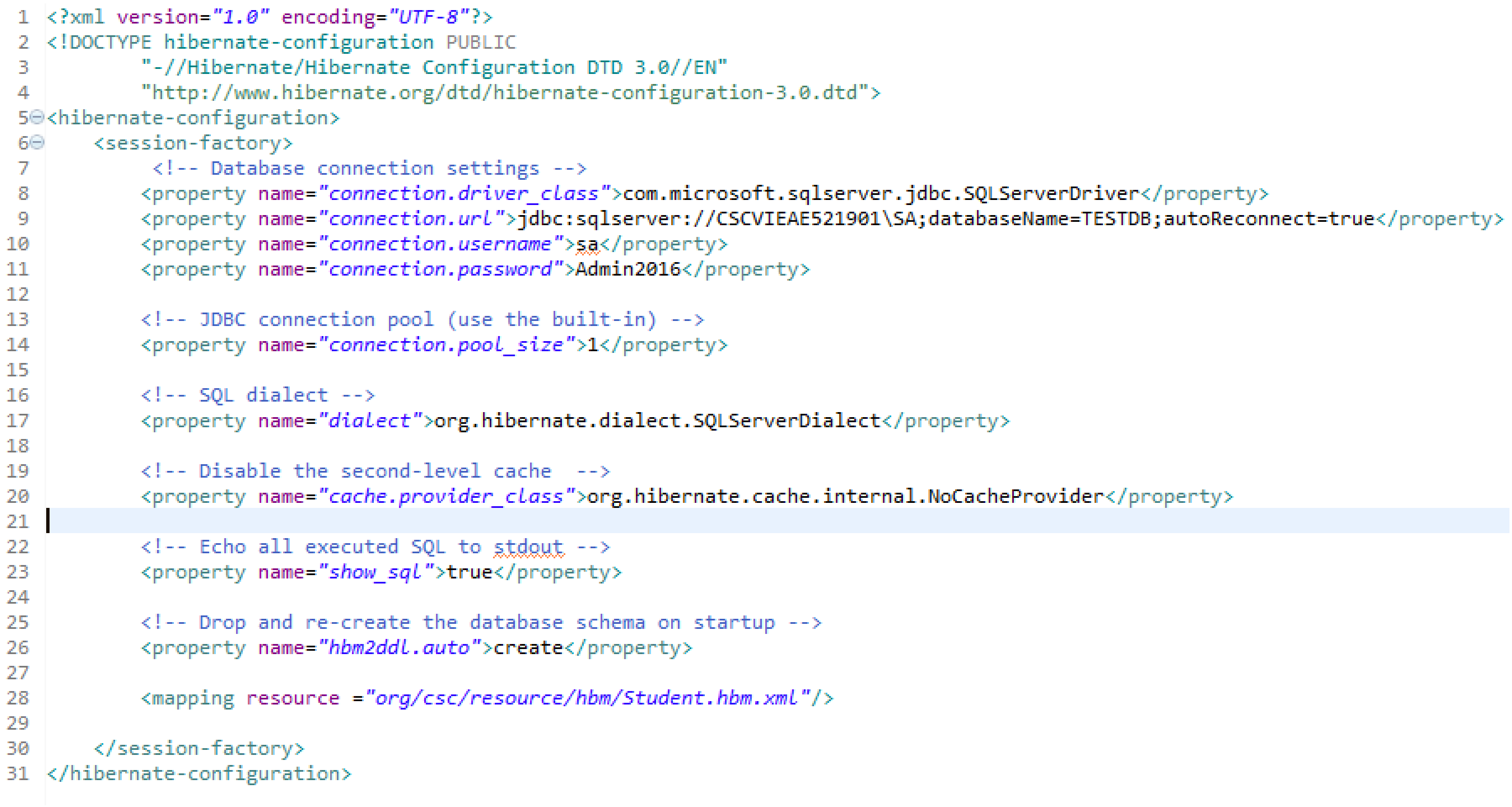
- Step 3: Map the Student object to the database Student table by creating Student.hbm.xml file



```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <class name = "org.csc.hibernate.dto.Student" table="STUDENT">
    <id name="studentId" type="long" column="STUDENT_ID">
         <generator class="native"/>
      </id>
      <property name="studentName" column="STUDENT_NAME" type="string" not-null="true"/>
      <property name="address" column="ADDRESS" type="string" not-null="true"/>
   </class>
</hibernate-mapping>
```

Hibernate main classes and interface API

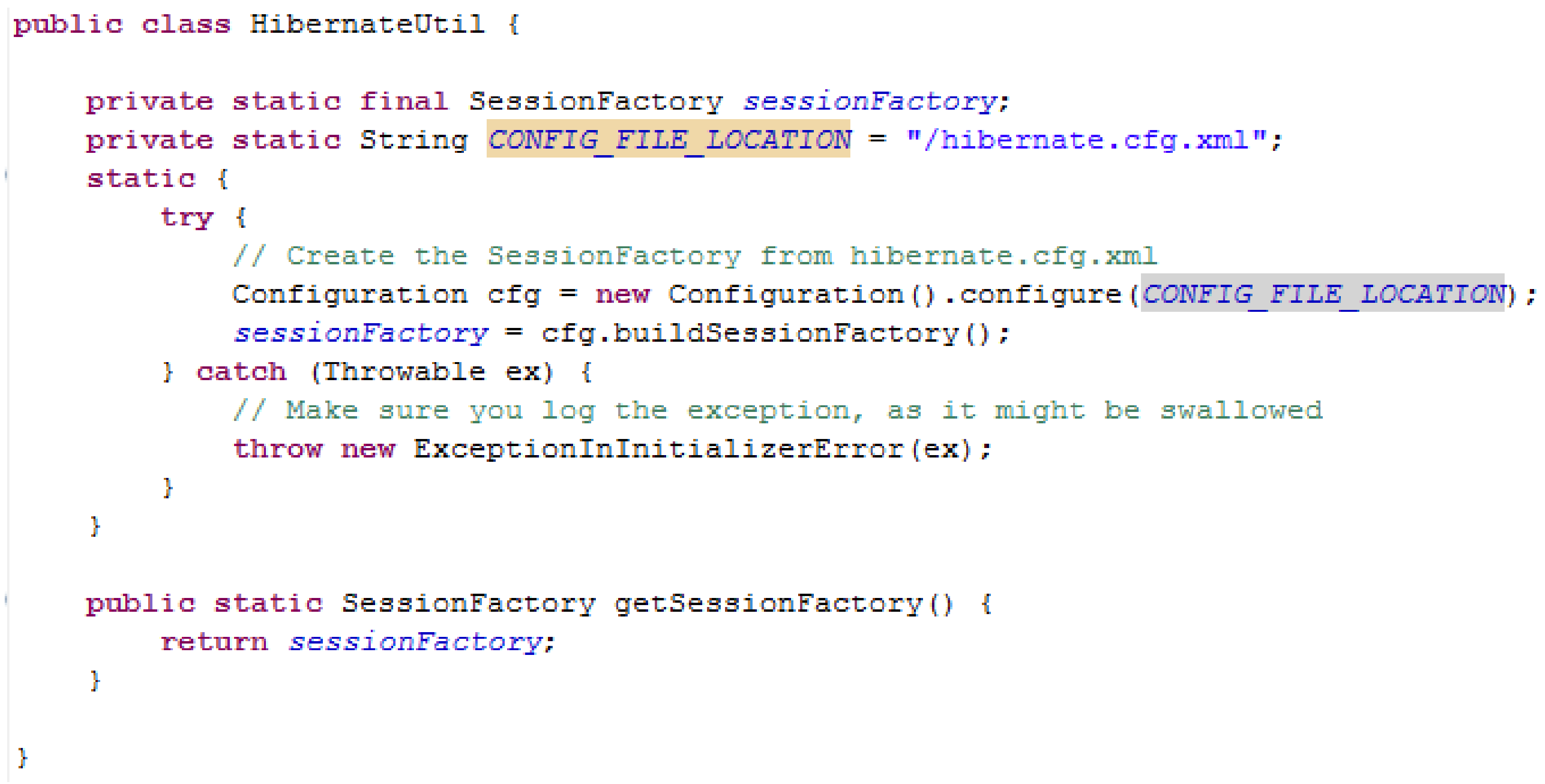
- Step 4: Create Hibernate configuration file (hibernate.cfg.xml)



```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
         <!-- Database connection settings -->
        <property name="connection.driver_class">com.microsoft.sqlserver.jdbc.SQLServerDriver</property>
        <property name="connection.url">jdbc:sqlserver://CSCVIEAE521901\SA;databaseName=TESTDB;autoReconnect=true</property>
        <property name="connection.username">sa</property>
        <property name="connection.password">Admin2016</property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

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

        <!-- Disable the second-level cache  -->
        <property name="cache.provider_class">org.hibernate.cache.internal.NoCacheProvider</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create</property>

        <mapping resource ="org/csc/resource/hbm/Student.hbm.xml"/>

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

Hibernate main classes and interface API

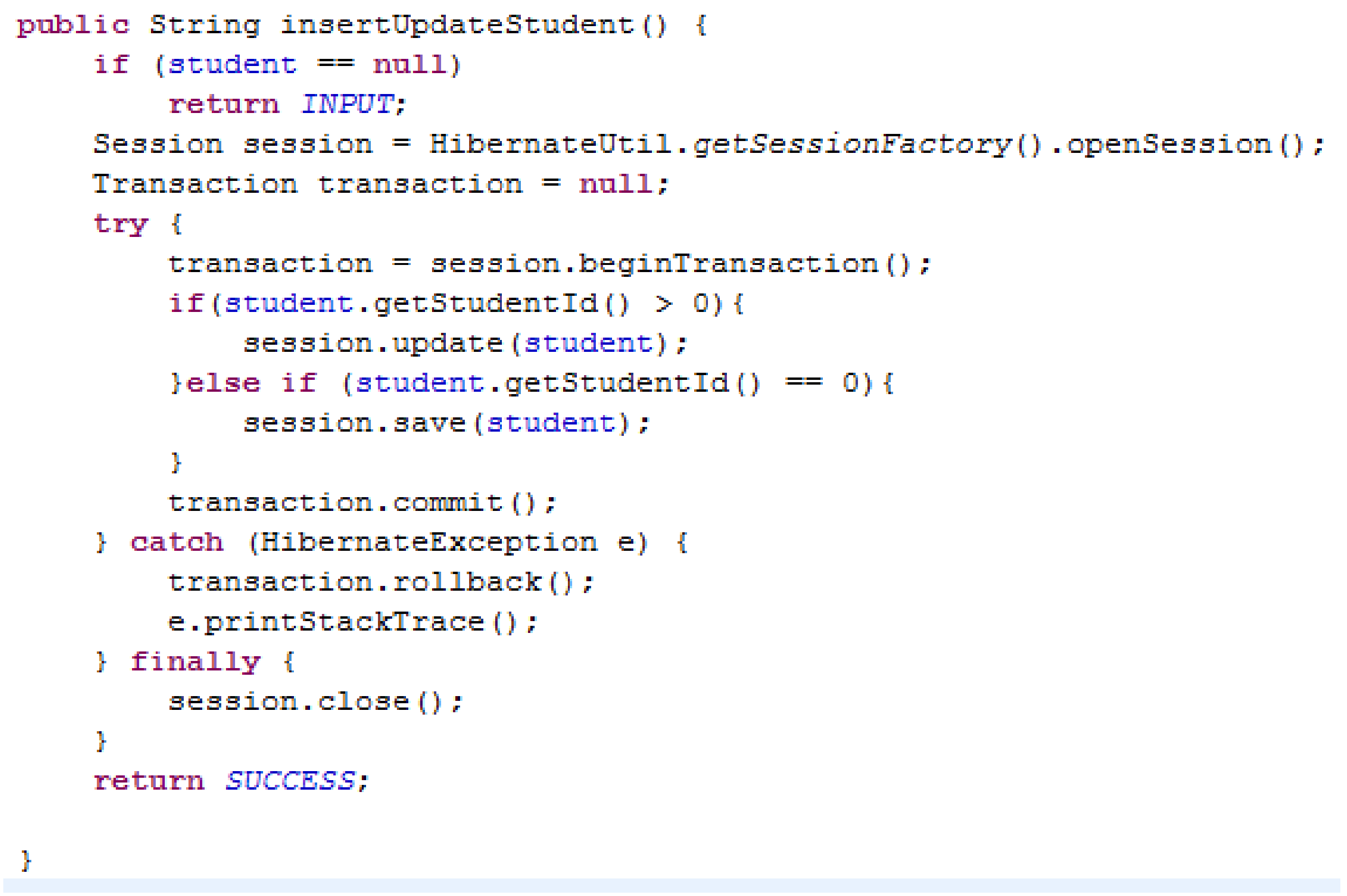
- Step 5: Develop a Hibernate Util class



```
public class HibernateUtil {

    private static final SessionFactory sessionFactory;
    private static String CONFIG_FILE_LOCATION = "/hibernate.cfg.xml";
    static {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            Configuration cfg = new Configuration().configure(CONFIG_FILE_LOCATION);
            sessionFactory = cfg.buildSessionFactory();
        } catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

}
```

Hibernate main classes and interface API

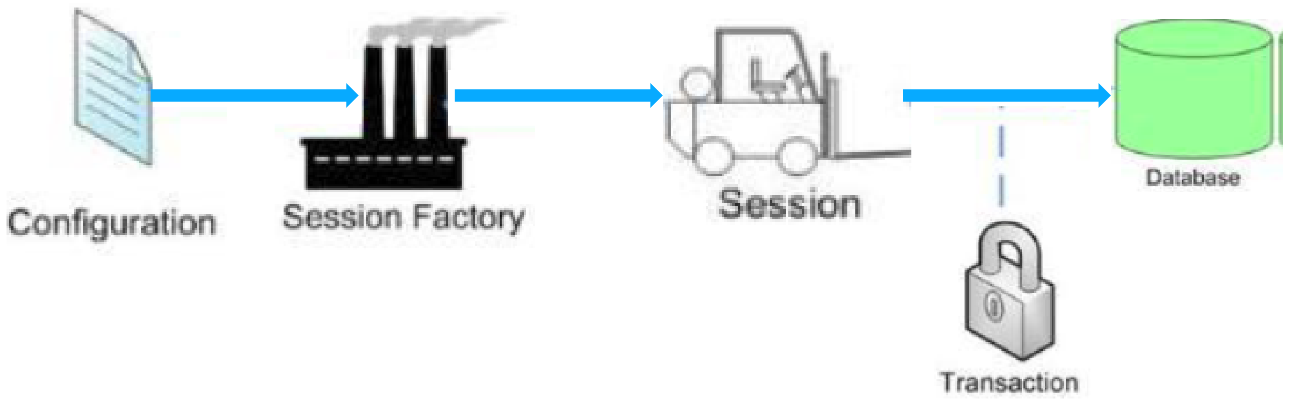
- Step 6: Develop the Insert or update method



```
public String insertUpdateStudent() {
    if (student == null)
        return INPUT;
    Session session = HibernateUtil.getSessionFactory().openSession();
    Transaction transaction = null;
    try {
        transaction = session.beginTransaction();
        if(student.getStudentId() > 0){
            session.update(student);
        }else if (student.getStudentId() == 0){
            session.save(student);
        }
        transaction.commit();
    } catch (HibernateException e) {
        transaction.rollback();
        e.printStackTrace();
    } finally {
        session.close();
    }
    return SUCCESS;

}
```

Hibernate main classes and interface API

- Flow of Hibernate application



Annotation mappings

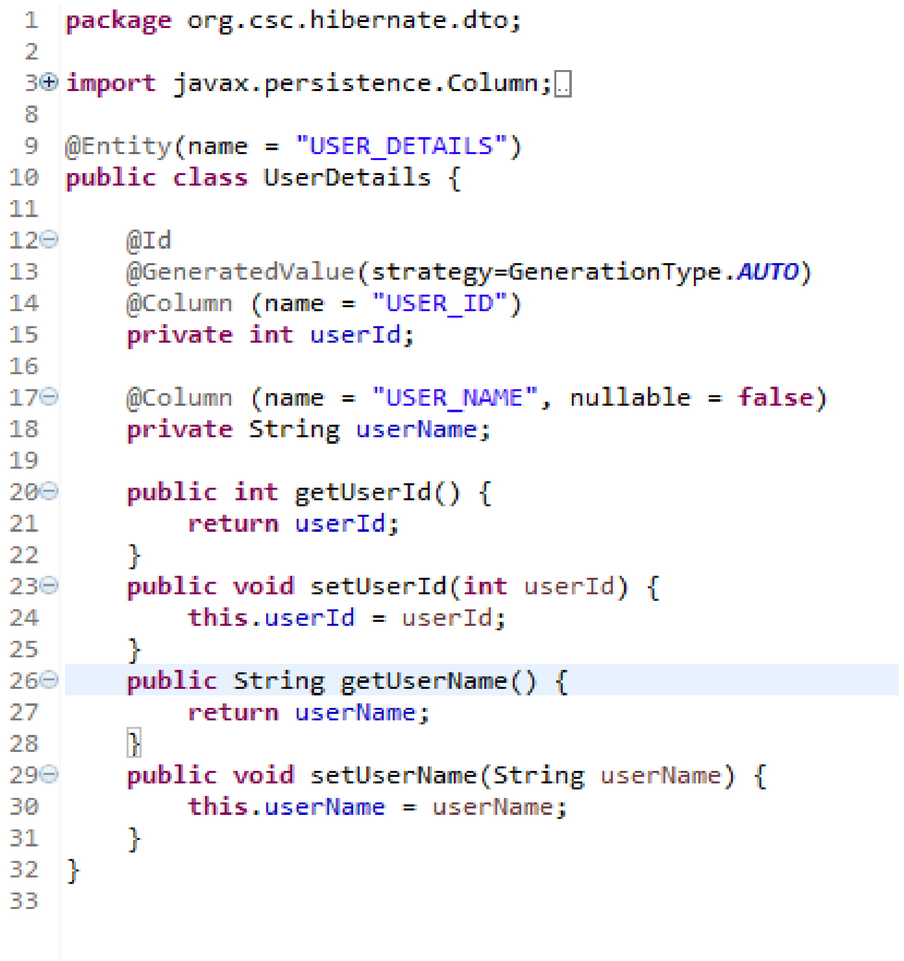
- Make sure you have Java 5.0 or higher version is installed
- Hibernate Core 3.2.0 and above
- In addition to the already existing jar files you need to add the following jar files to the classpath
  - hibernate-commons-annotations.jar
  - ejb3-persistence.jar
  - hibernate-annotations.jar

Annotation mappings

- Some common annotation attributes
  - **@Entity**: Identifies an entity and allows attributes, such as its name, to be overridden from the defaults
  - **@Table** : Allows the default details of an entity’s primary table to be overridden.
  - **@Column**: Associates a field or property of the class with a column in the mapped table.
  - **@Id**: Identifies the primary key of the entity. Placement of the @Id attribute also determines whether the default access mode for the entity class is field or property access.
  - **@GeneratedValue**: Allows generation strategies to be specified for the marked entity’s primary key value(s).
  - **@Transient** : Allows a field or property to be marked so that it will not be persisted.
  - **@OneToOne** : Allows a one-to-one association to be defined between entities.
  - **@OneToMany** : Allows a one-to-many association to be defined between entities.
  - **@ManyToMany** : Allows a many-to-many association to be defined between entities.

Annotation mappings

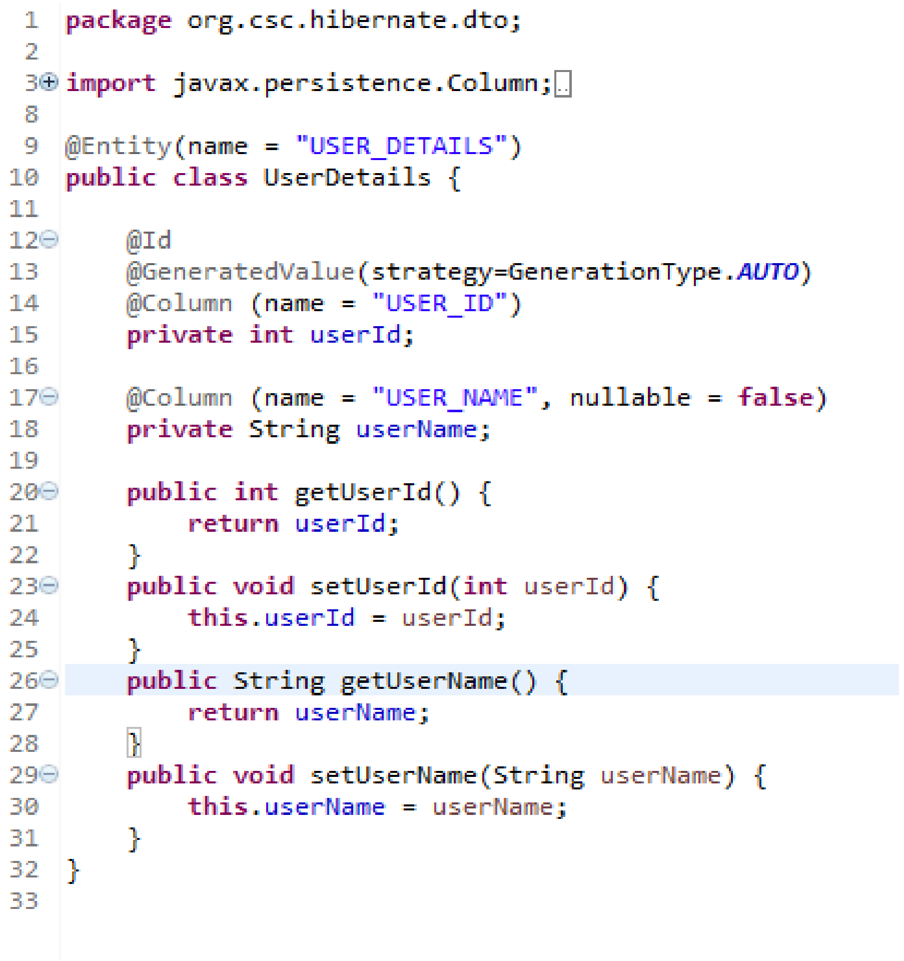
– Creating the *User* class with annotations



```java
package org.csc.hibernate.dto;

import javax.persistence.Column;

@Entity(name = "USER_DETAILS")
public class UserDetails {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column (name = "USER_ID")
    private int userId;

    @Column (name = "USER_NAME", nullable = false)
    private String userName;

    public int getUserId() {
        return userId;
    }
    public void setUserId(int userId) {
        this.userId = userId;
    }
    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
}

```

Annotation mappings

– Creating the *User* class with annotations



```java
package org.csc.hibernate.dto;

import javax.persistence.Column;

@Entity(name = "USER_DETAILS")
public class UserDetails {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column (name = "USER_ID")
    private int userId;

    @Column (name = "USER_NAME", nullable = false)
    private String userName;

    public int getUserId() {
        return userId;
    }
    public void setUserId(int userId) {
        this.userId = userId;
    }
    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
}

```

Working with Session interface

- Session is the main runtime interface between a Java application and Hibernate
- The main functions of the Session are:
  - Load instances
  - Create instances
  - Update instances
  - Delete instances

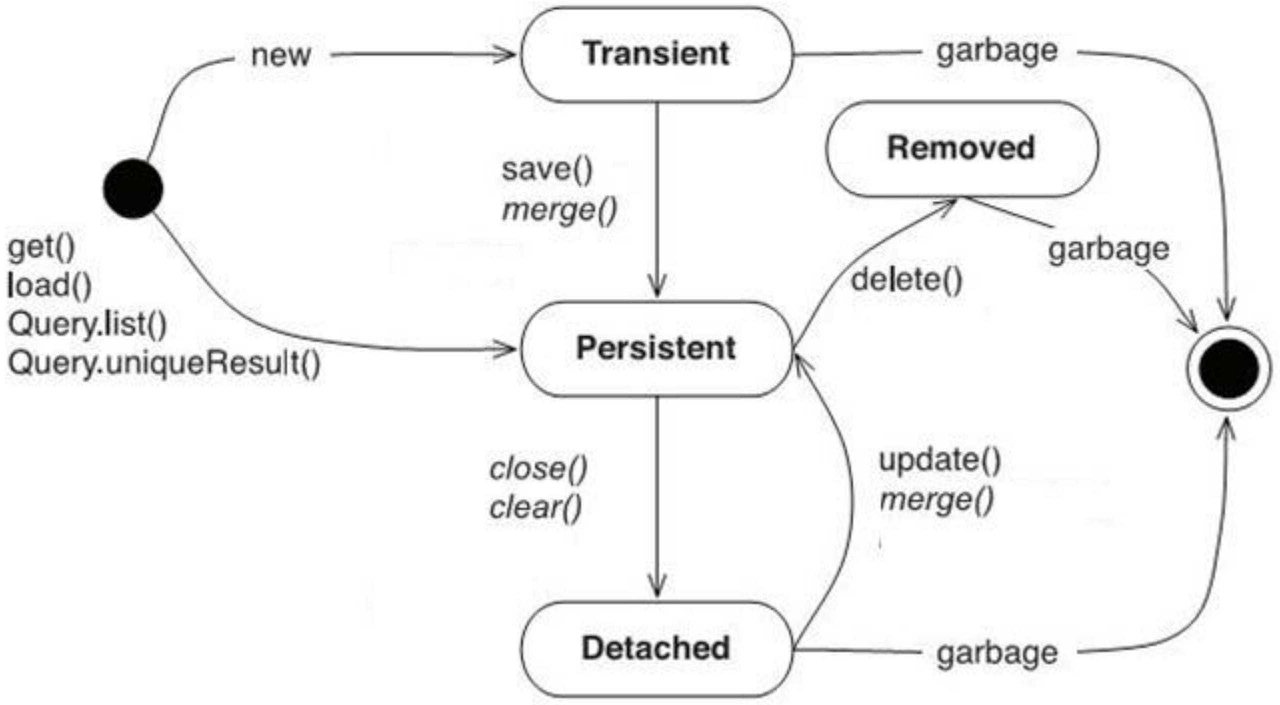
Working with Session interface

- Some important methods of Session interface:

Method	Description
connection()	Get the JDBC connection of this Session
contains(Object object)	Check if the instance is associated with the Session
merge(Object object)	Copy the state of the given object to the persistent object with the same identifier
save(Object object)	Persist the give transient instance, first assigning generated identifier
update(Object object)	Update the persistent instance with the given detached instance
delete(Object object)	Remove the persistent instance from data store
createSQLQuery(String query)	Create a new instance of SQLQuery for the given SQL query string
disconnect()	Disconnect the Session from the current JDBC connection
getTransaction()	Get the Transaction instance associated with this Session

Working with Session interface

- Instances may exist in one of three states:
  - *Transient: never persistent, not associated with any Session*
  - *Persistent: associated with a unique Session*
  - *Detached: previously persistent, not associated with any Session*



Points to remember

- Flow of Hibernate application
- Hibernate API
- Session interface
- Instance states

Hibernate Object Relational mapping

Hibernate Object Relational mapping

- Why Object Relational Mapping?
- Ways to map
- Types of mappings
- Inheritance
- Annotation mappings
- Hibernate data types

Why Object Relational Mapping?

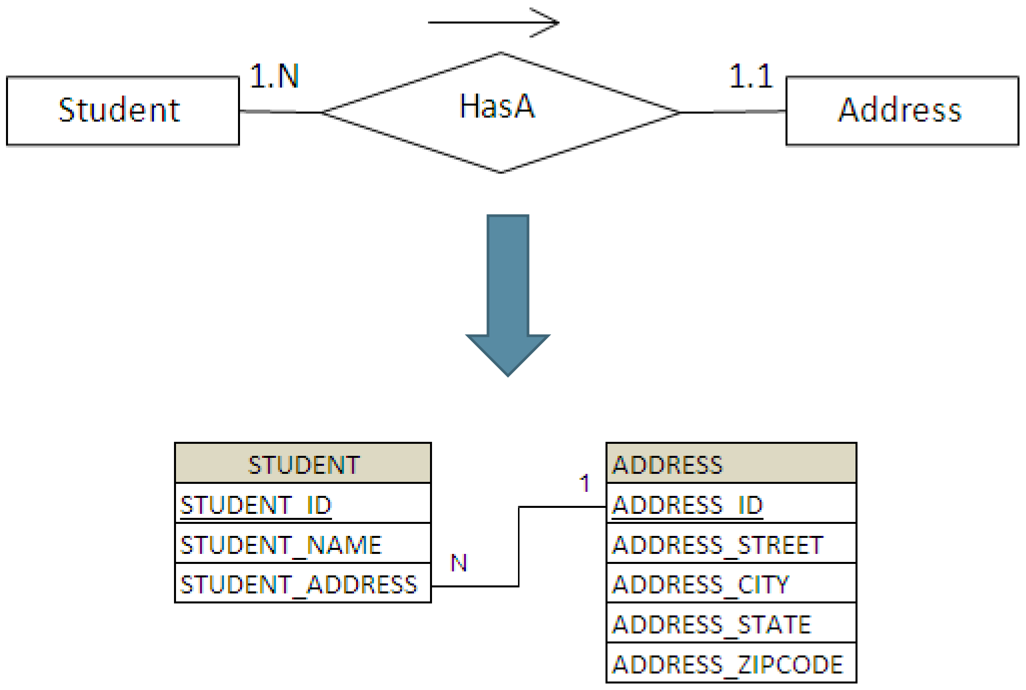
- Structural mapping more robust
- Less error-prone code
- Optimized performance all the time
- Vendor independence
- The mapping document is designed to be readable and hand-editable.

Ways to map

- XML Mappings
  - The technique that has been available the longest is the use of XML mapping files.
  - These files can be created directly with a text editor or with the help of various tools created by the Hibernate team and others
- Annotation
  - Hibernate now also supports the Annotations feature introduced in Java 5. This permits the use of a special syntax to include metadata directly in the source code for the application

Types of association mappings

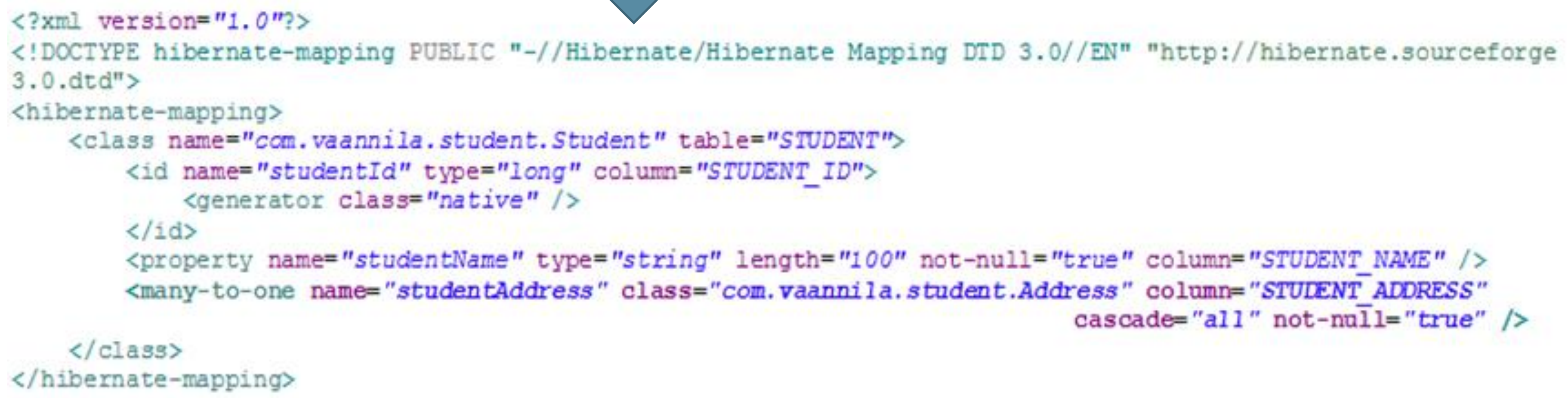
- Many-to-One Association
  – Consider the following relationship between Student and Address entity.



Types of association mappings

- Many-to-One Association
  - Student Java and mapping files:

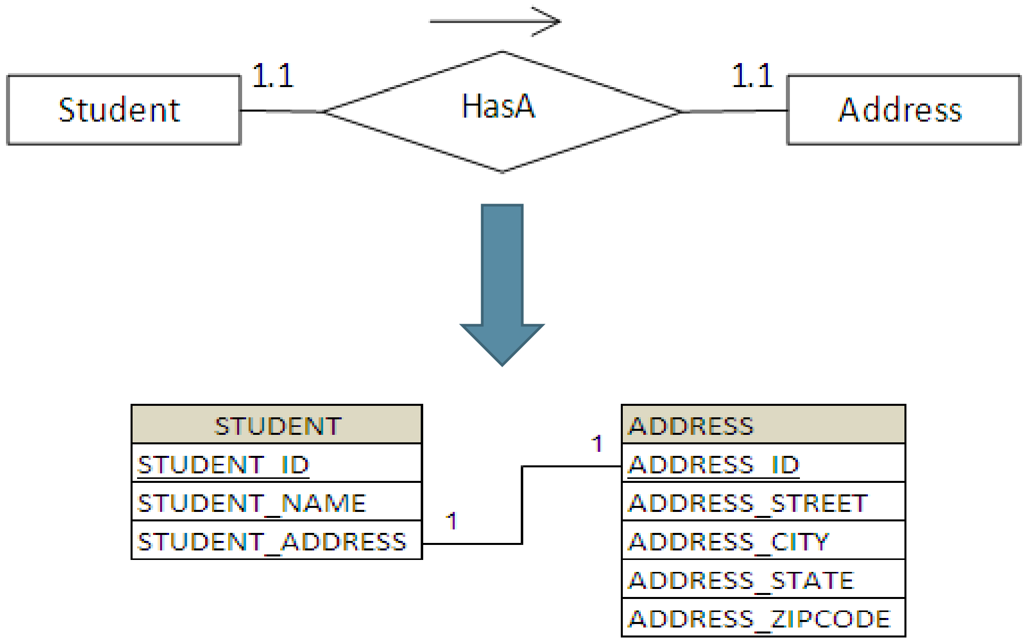
```
public class Student implements java.io.Serializable {

    private long studentId;
    private String studentName;
    private Address studentAddress;

    …
}
```



```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge
3.0.dtd">
<hibernate-mapping>
    <class name="com.vaannila.student.Student" table="STUDENT">
        <id name="studentId" type="long" column="STUDENT_ID">
            <generator class="native" />
        </id>
        <property name="studentName" type="string" length="100" not-null="true" column="STUDENT_NAME" />
        <many-to-one name="studentAddress" class="com.vaannila.student.Address" column="STUDENT_ADDRESS"
                                                                          cascade="all" not-null="true" />
    </class>
</hibernate-mapping>
```

Types of association mappings

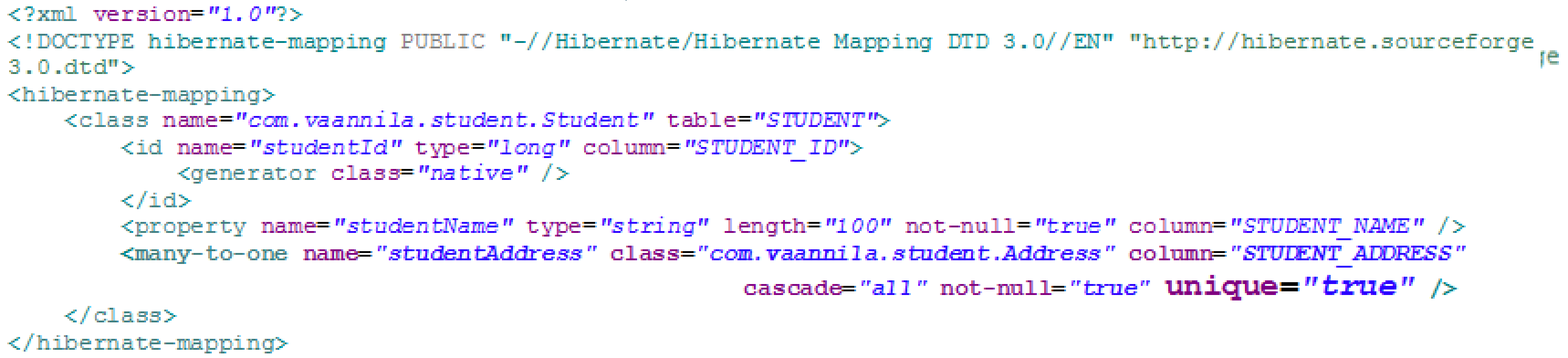
- One-to-One Association
  – Consider the following relationship between Student and Address entity



Types of association mappings

- One-to-One Association
  - Student Java and mapping files:

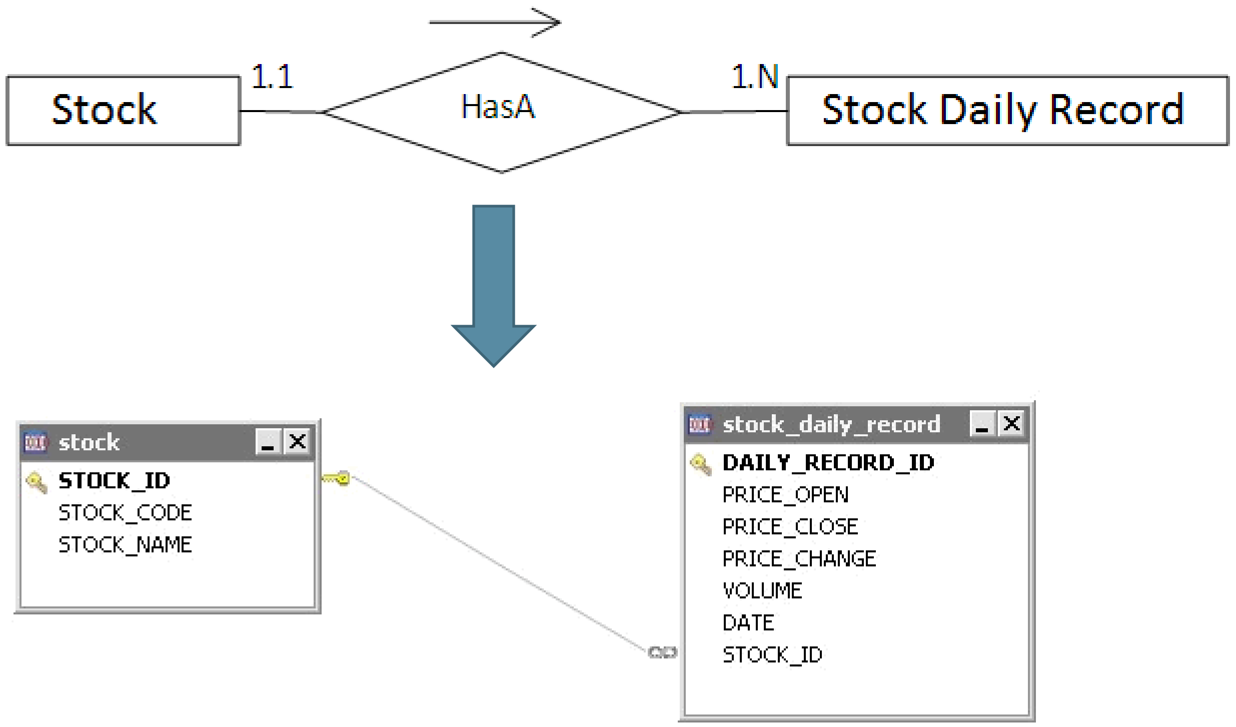
```
public class Student implements java.io.Serializable {

    private long studentId;
    private String studentName;
    private Address studentAddress;

    …
}
```



```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge
3.0.dtd">
<hibernate-mapping>
    <class name="com.vaannila.student.Student" table="STUDENT">
        <id name="studentId" type="long" column="STUDENT_ID">
            <generator class="native" />
        </id>
        <property name="studentName" type="string" length="100" not-null="true" column="STUDENT_NAME" />
        <many-to-one name="studentAddress" class="com.vaannila.student.Address" column="STUDENT_ADDRESS"
                                                   cascade="all" not-null="true" unique="true" />
    </class>
</hibernate-mapping>
```

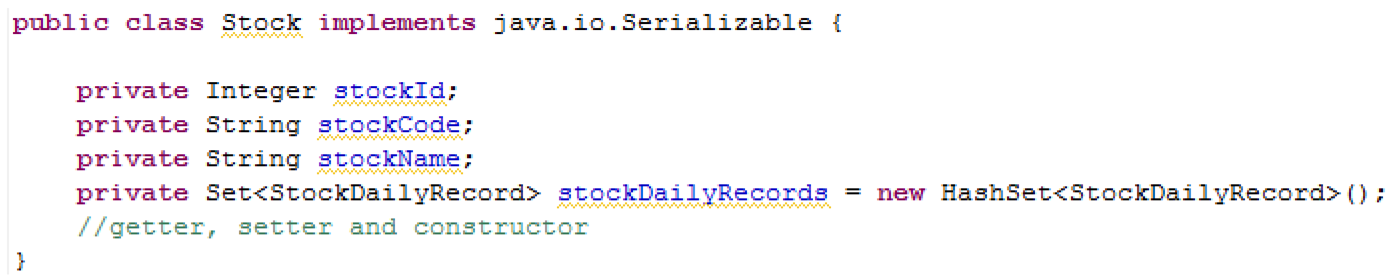
Types of association mappings

- One-to-Many Association
  - Consider the following relationship between *Student* and *Phone* entity

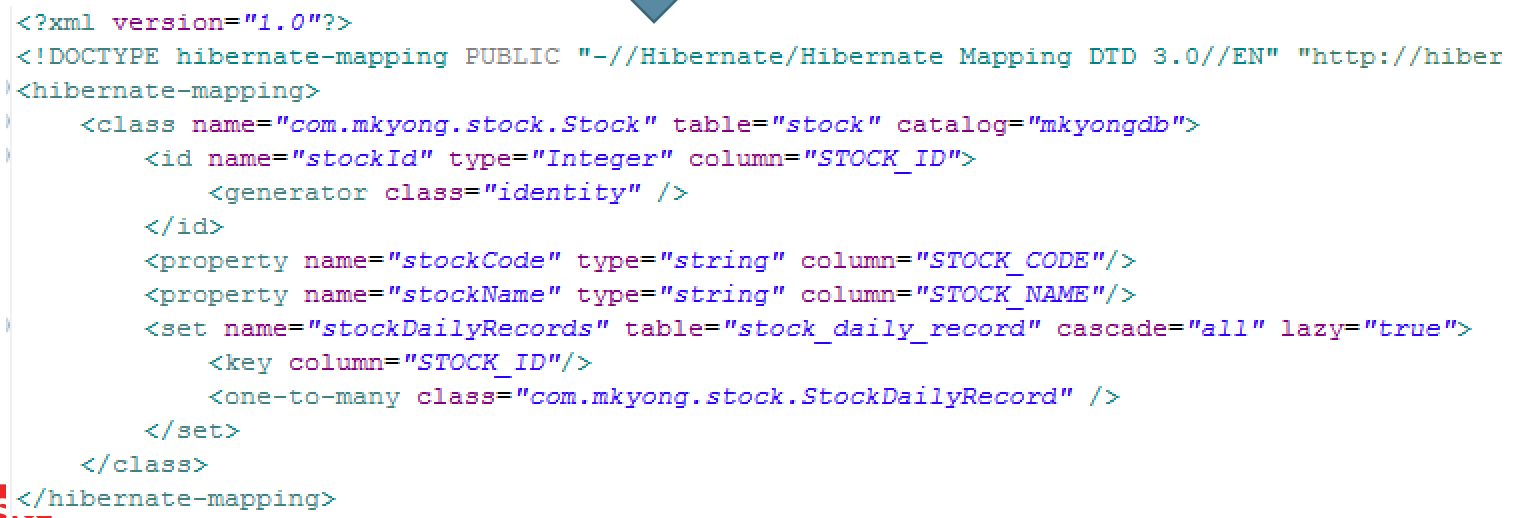


Types of association mappings

- One-to-Many Association
  - Student and Phone Java files:



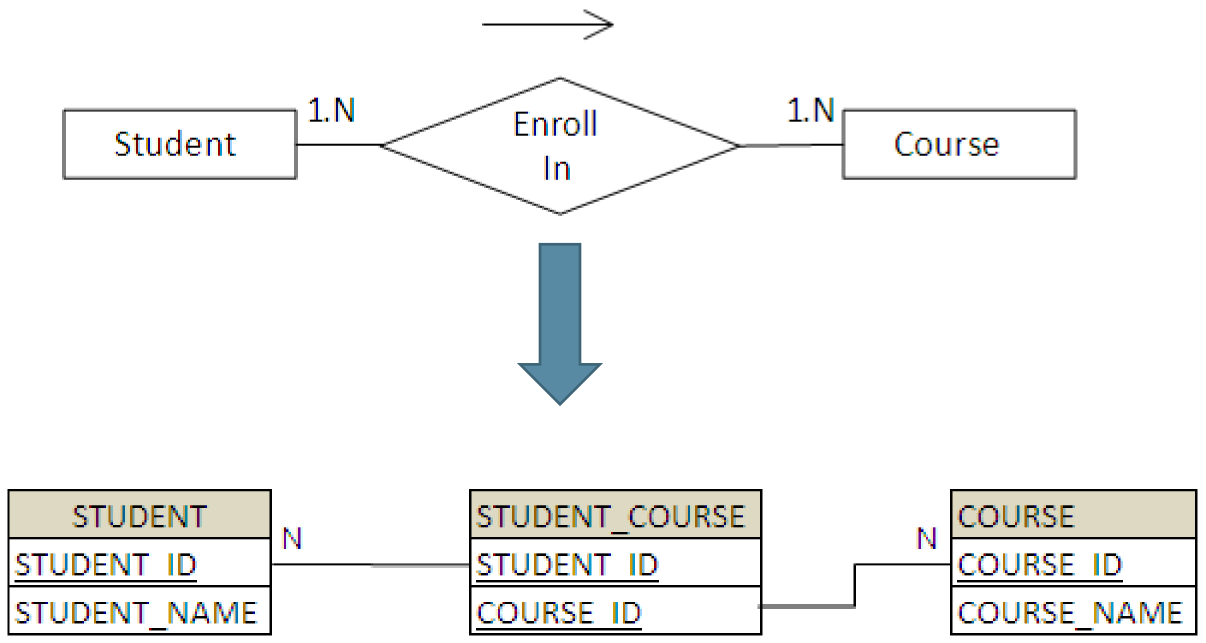
```
public class Stock implements java.io.Serializable {

    private Integer stockId;
    private String stockCode;
    private String stockName;
    private Set<StockDailyRecord> stockDailyRecords = new HashSet<StockDailyRecord>();
    //getter, setter and constructor
}
```



```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hiber
<hibernate-mapping>
    <class name="com.mkyong.stock.Stock" table="stock" catalog="mkyongdb">
        <id name="stockId" type="Integer" column="STOCK_ID">
            <generator class="identity" />
        </id>
        <property name="stockCode" type="string" column="STOCK_CODE"/>
        <property name="stockName" type="string" column="STOCK_NAME"/>
        <set name="stockDailyRecords" table="stock_daily_record" cascade="all" lazy="true">
            <key column="STOCK_ID"/>
            <one-to-many class="com.mkyong.stock.StockDailyRecord" />
        </set>
    </class>
</hibernate-mapping>
```

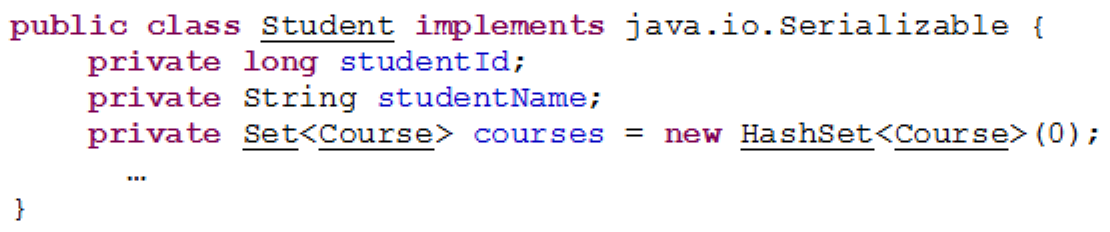
Types of association mappings

- Many-to-Many Association
  - Consider the following relationship between ***Student*** and ***Course*** entity

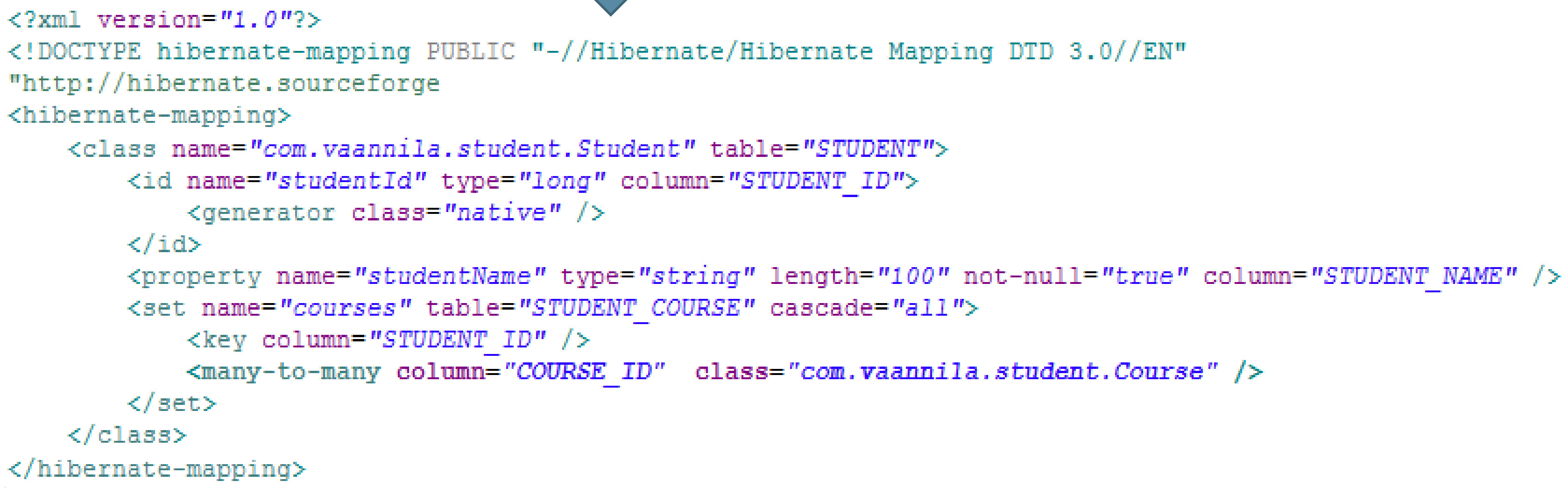


Types of association mappings

- Many-to-Many Association
  - Student and Course Java files:



```
public class Student implements java.io.Serializable {
    private long studentId;
    private String studentName;
    private Set<Course> courses = new HashSet<Course>(0);
      …
}
```



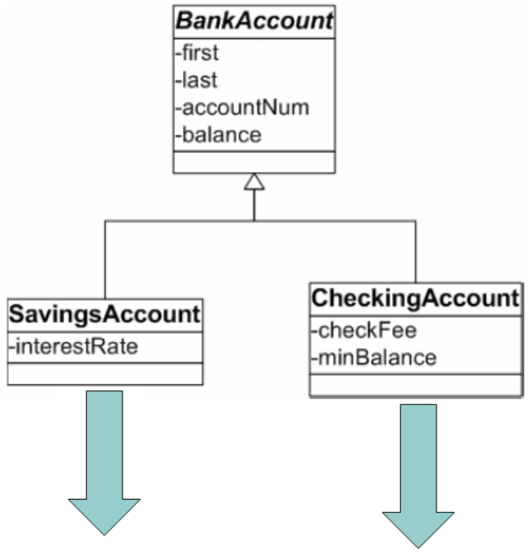
```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge
<hibernate-mapping>
    <class name="com.vaannila.student.Student" table="STUDENT">
        <id name="studentId" type="long" column="STUDENT_ID">
            <generator class="native" />
        </id>
        <property name="studentName" type="string" length="100" not-null="true" column="STUDENT_NAME" />
        <set name="courses" table="STUDENT_COURSE" cascade="all">
            <key column="STUDENT_ID" />
            <many-to-many column="COURSE_ID"  class="com.vaannila.student.Course" />
        </set>
    </class>
</hibernate-mapping>
```

Inheritance

- Hibernate also handles inheritance associations
- Three strategies with regard to inheritance mapping
  - Table per concrete class
  - Table per subclass
  - Table per class hierarchy

Inheritance

- Table per concrete class



Savings_Accounts

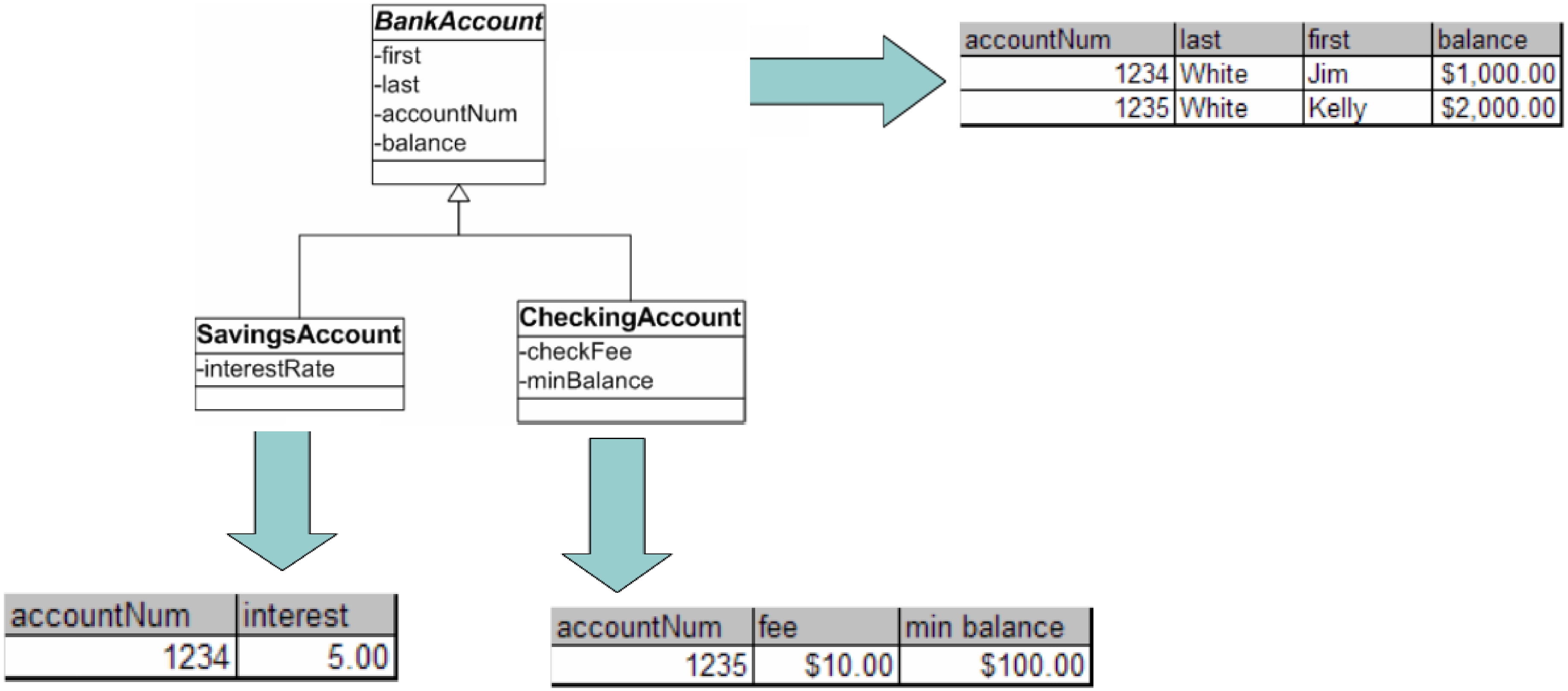
accountNum	last	first	balance	interest
1234	White	Jim	$1,000.00	5.00

Checking_Accounts

accountNum	last	first	balance	fee	min balance
1235	White	Jim	$1,000.00	$10.00	$100.00

Inheritance

- Table per subclass



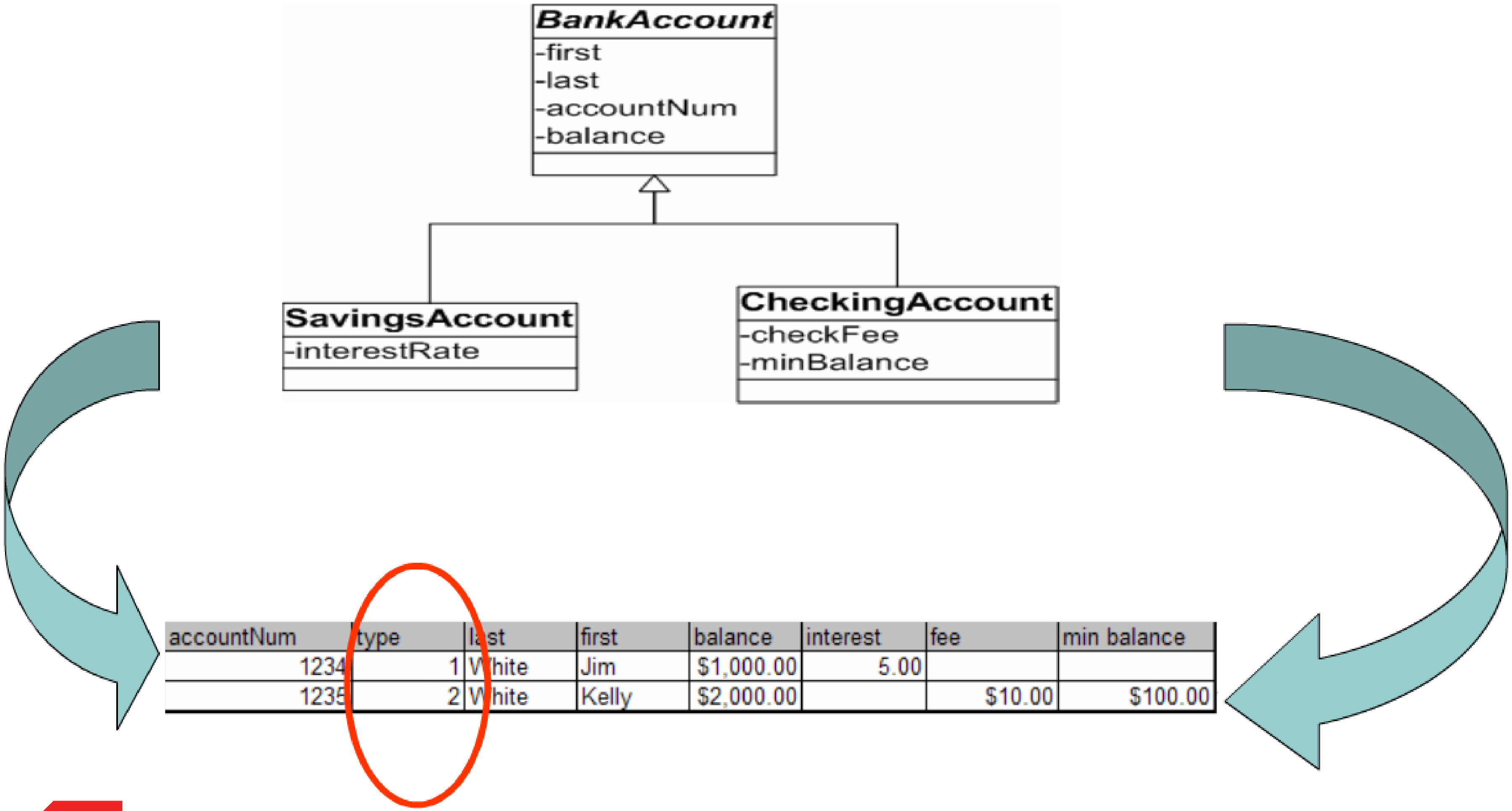
accountNum	last	first	balance
1234	White	Jim	$1,000.00
1235	White	Kelly	$2,000.00

accountNum	interest
1234	5.00

accountNum	fee	min balance
1235	$10.00	$100.00

Inheritance

- Table per class hierarchy

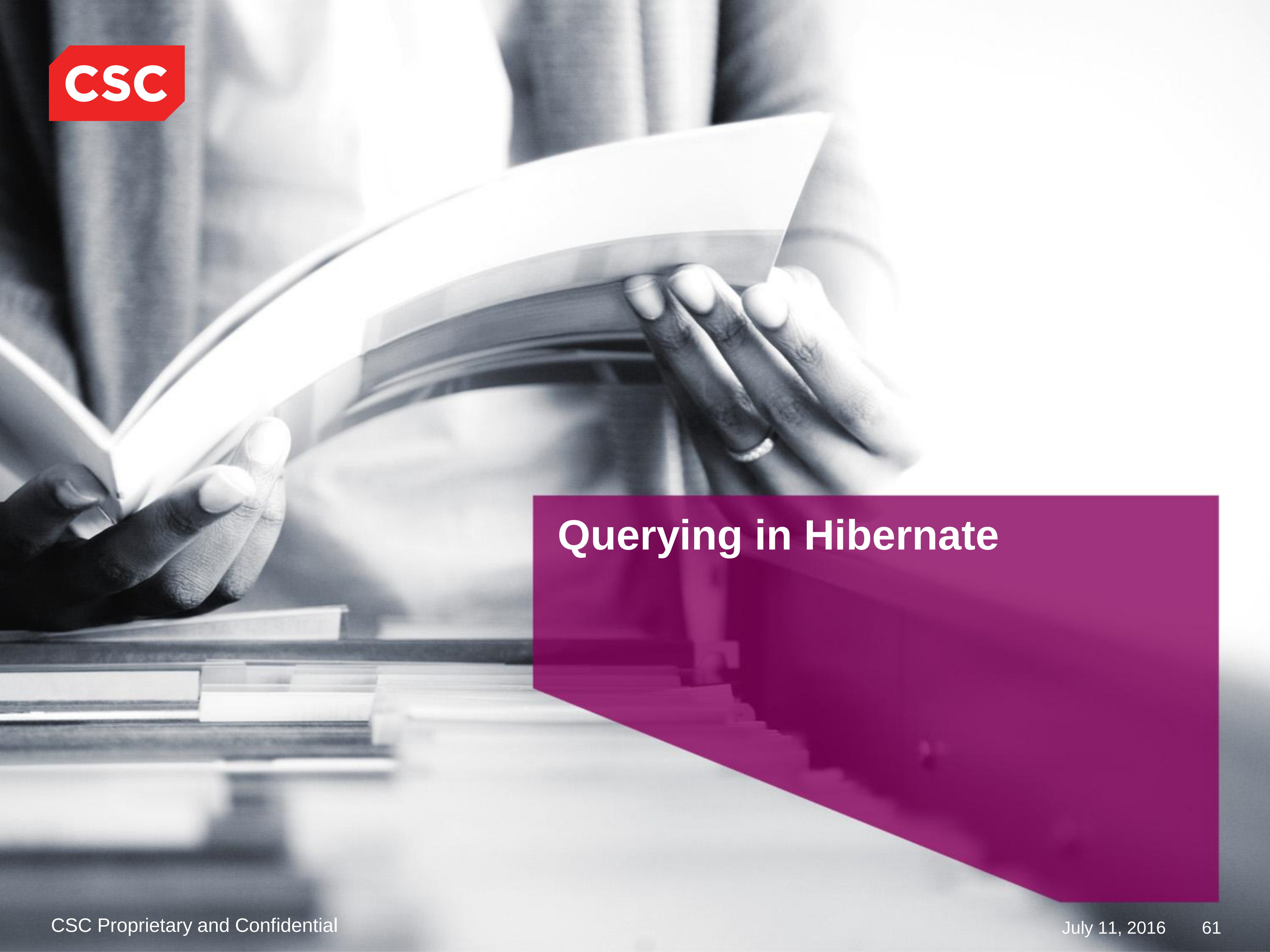


accountNum	type	last	first	balance	interest	fee	min balance
1234	1	White	Jim	$1,000.00	5.00		
1235	2	White	Kelly	$2,000.00		$10.00	$100.00

Hibernate data types

- Support all Java primitives and many JDK classes.
- Hibernate supports user-defined custom types.
- Example:
  - int, long, String, java.io.Serialize, java.util.Calendar,….. (Java Types)
  - Collection, List, ArrayList,…. (Java Collection)
  - Personel, PlayerInfo,…. (Custom Types)

mappings
- Types of
mappings



Querying in Hibernate

Querying in Hibernate

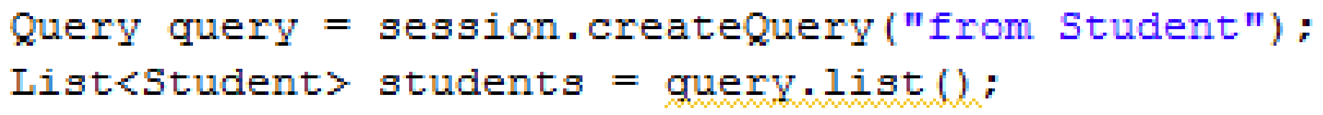
- Hibernate Query Language
- The Criteria Query API
- Native SQL

Hibernate Query Language

- Make SQL be object oriented
  - Classes and properties instead of tables and columns
  - Polymorphism
  - Associations
  - Much less verbose than SQL
- Full support for relational operations
  - Inner/outer/full joins
  - Projection
  - Aggregation (max, avg) and grouping
  - Ordering
  - Subqueries
  - SQL functions calls
- Database independent
  - Queries written in HQL are database independent

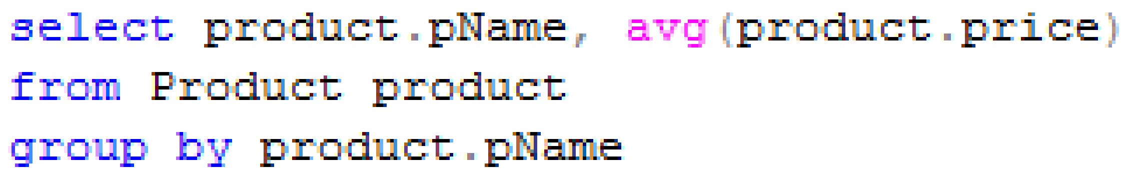
Hibernate Query Language

- Support: Select, From, Where, Order by, Group by and Having clause
- Simple query

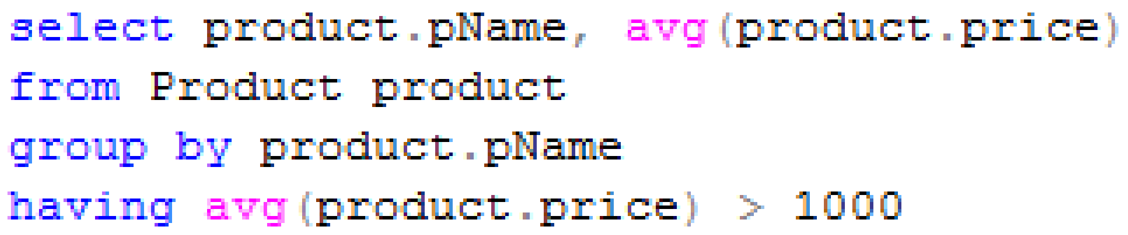


```
Query query = session.createQuery("from Student");
List<Student> students = query.list();
```

- Aggregation query



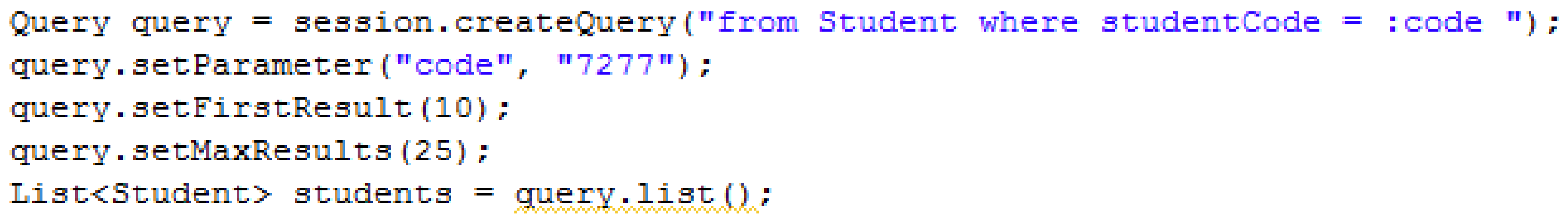
```
select product.pName, avg(product.price)
from Product product
group by product.pName
```



```
select product.pName, avg(product.price)
from Product product
group by product.pName
having avg(product.price) > 1000
```

Hibernate Query Language

- Paging Through the Result Set
  - setFirstResult(): set the first row to retrieve
  - setMaxResults(): set the maximum number of rows to retrieve



```
Query query = session.createQuery("from Student where studentCode = :code ");
query.setParameter("code", "7277");
query.setFirstResult(10);
query.setMaxResults(25);
List<Student> students = query.list();
```

Hibernate Query Language

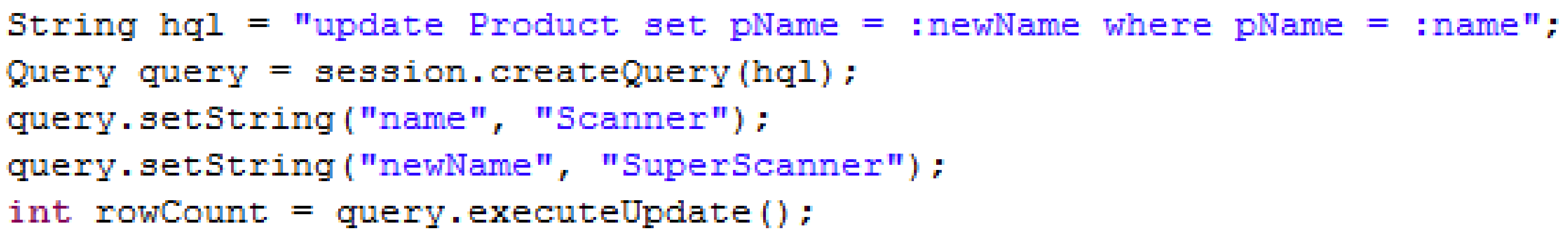
- Associations and joins
  - Allows to use more than one class in HQL query
    - inner join
    - cross join
    - left outer join
    - right outer join
    - Full join (not usually useful)
- Aggregate functions
  - They work the same way in HQL as they do in SQL
    - avg(…)
    - sum(…)
    - min(…)
    - max(…)
    - count(*)
    - count(…)
    - count(distinct…)
    - count(all…)

Hibernate Query Language

- Expressions
  - Allowed in the where clause include most of the operations that you could perform in SQL:
    - Mathematical operators +, -, *, /
    - Binary comparison operators =, >=, <=, <>, !=, like
    - Logical operations and, or, not
    - String concatenation ||
    - SQL scalar functions such as upper() and lower()
    - Parentheses ( ) indicate grouping
    - in, between, is null
    - JDBC IN parameters ?
    - named parameters :name, :start_date, :x1

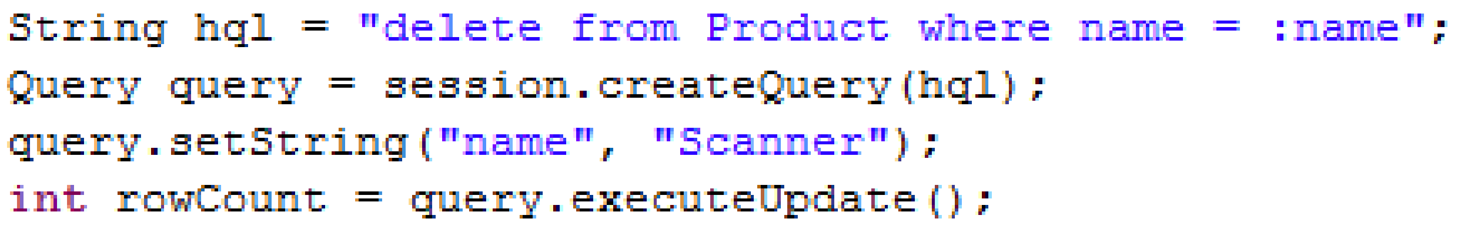
Hibernate Query Language

- Bulk Updates and Deletes:
  - Bulk Updates:



```
String hql = "update Product set pName = :newName where pName = :name";
Query query = session.createQuery(hql);
query.setString("name", "Scanner");
query.setString("newName", "SuperScanner");
int rowCount = query.executeUpdate();
```

  - Bulk Deletes:

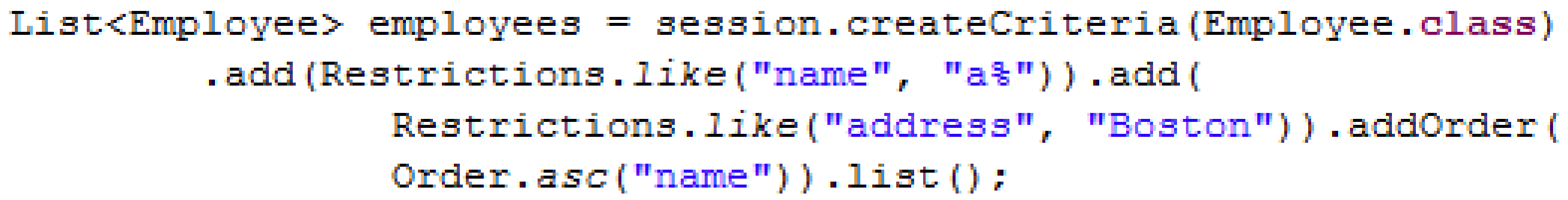


```
String hql = "delete from Product where name = :name";
Query query = session.createQuery(hql);
query.setString("name", "Scanner");
int rowCount = query.executeUpdate();
```

The Criteria Query API

- Provides way of generating queries through method calls
- Provide a compile-time syntax-checking that is not possible with a query language
- Lets you build nested, structured query expressions in Java
- The Criteria API also includes *query by example (QBE) functionality*

Criteria Query example:



```
List<Employee> employees = session.createCriteria(Employee.class)
        .add(Restrictions.like("name", "a%")).add(
                Restrictions.like("address", "Boston")).addOrder(
                Order.asc("name")).list();
```

The Criteria Query API

- Using the Restrictions

  eq(…): Apply an "equal" constraint to the named property

  ge(…) : Apply a "greater than or equal" constraint to the named property

  gt(…): Apply a "greater than" constraint to the named property

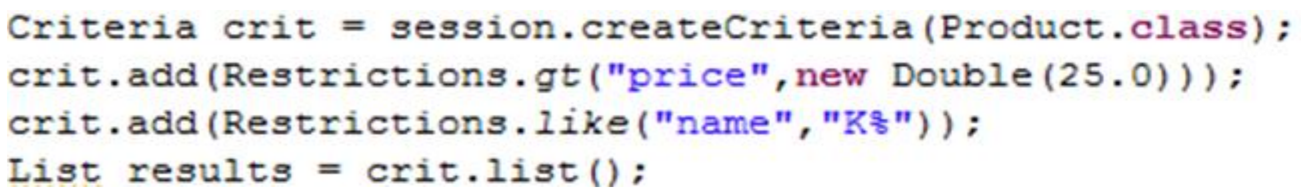
  like(…): Apply a "like" constraint to the named property

  ne(…): Apply a "not equal" constraint to the named property

  and(…): Return the conjunction of two expressions

  or(…): Return the disjunction of two expressions

  – **Criteria AND:**

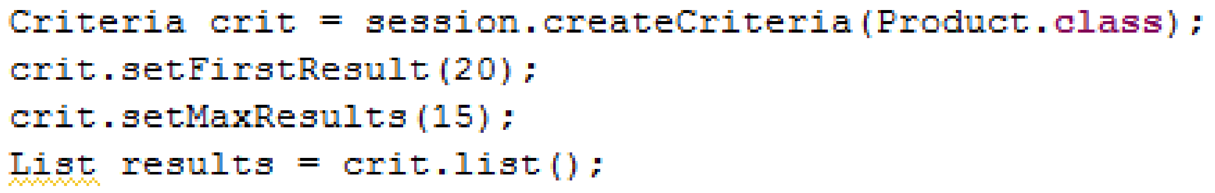


```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.gt("price",new Double(25.0)));
crit.add(Restrictions.like("name","K%"));
List results = crit.list();
```

  – **Criteria OR:**

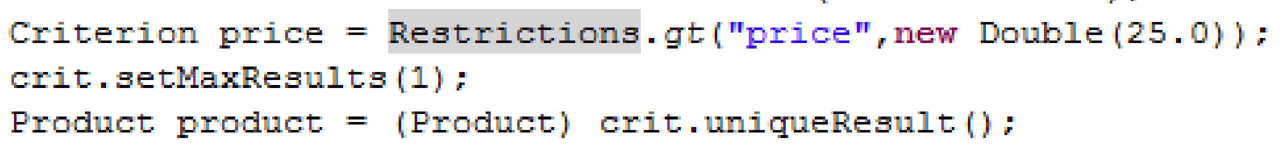
The Criteria Query API

- Paging Through the Result Set
  - setFirstResult(): takes an integer that represents the first row in your result set
  - setMaxResults() : set the maximum number of rows to retrieve



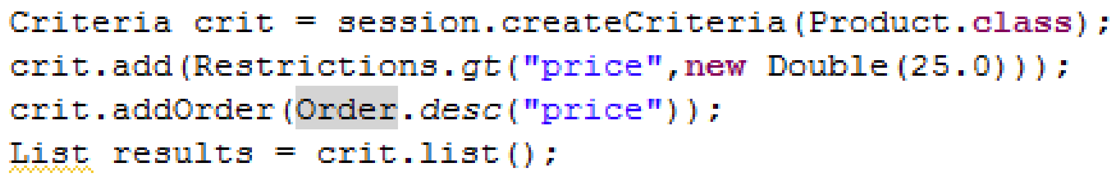
```
Criteria crit = session.createCriteria(Product.class);
crit.setFirstResult(20);
crit.setMaxResults(15);
List results = crit.list();
```

- Obtaining a Unique Result
  – Make sure that your query only returns one or zero results if you use the uniqueResult() method. Otherwise, Hibernate will throw an exception



```
Criteria crit = session.createCriteria(Product.class);
Criterion price = Restrictions.gt("price",new Double(25.0));
crit.setMaxResults(1);
Product product = (Product) crit.uniqueResult();
```

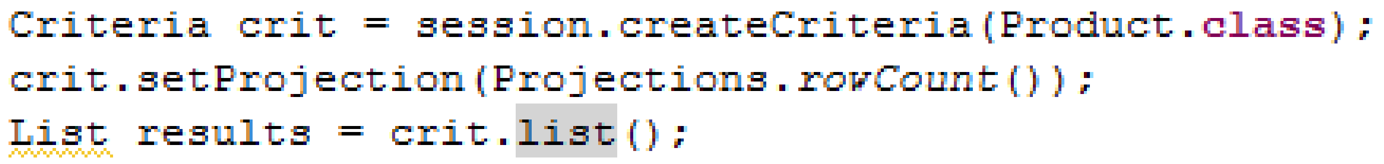
- Sorting the Query's Results



```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.gt("price",new Double(25.0)));
crit.addOrder(Order.desc("price"));
List results = crit.list();
```

The Criteria Query API

- Projections and Aggregates
  - The row-counting functionality provides a simple example of applying projections.

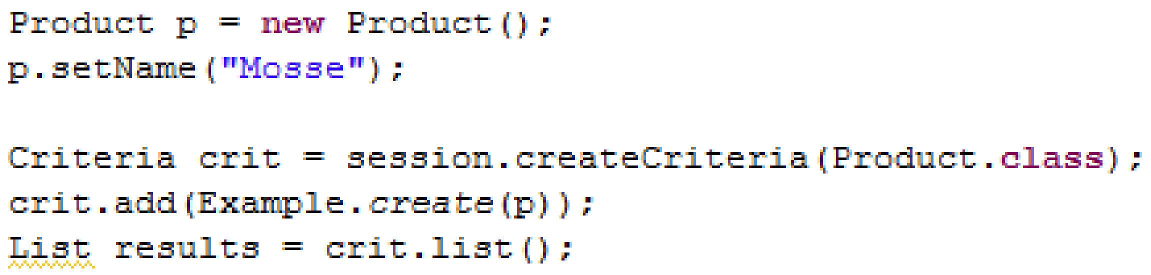


```
Criteria crit = session.createCriteria(Product.class);
crit.setProjection(Projections.rowCount());
List results = crit.list();
```

  - Other aggregate functions include the following:
    - avg(String propertyName): Gives the average of a property’s value
    - count(String propertyName): Counts the number of times a property occurs
    - countDistinct(String propertyName): Counts the number of unique values the property contains
    - max(String propertyName): Calculates the maximum value of the property values
    - min(String propertyName): Calculates the minimum value of the property values
    - sum(String propertyName): Calculates the sum total of the property values

The Criteria Query API

- Query by Example (QBE)
  - Use instance as a template and have Hibernate build the criteria for you based upon its values



```
Product p = new Product();
p.setName("Mosse");

Criteria crit = session.createCriteria(Product.class);
crit.add(Example.create(p));
List results = crit.list();
```

  - All the properties on our Example objects get examined.
  - The default is to ignore null-valued properties.

Native SQL

- Using a SQLQuery:

```
List list  = session.createSQLQuery("SELECT * FROM Student").list();
```

  – These will return a List of Object arrays **(Object[])** with scalar values for each column in the Student table.

  – To get entity objects from a native sql query via addEntity().

```
List list  = session.createSQLQuery("SELECT * FROM Student").addEntity(Student.class).list();
```

- Using Named SQL Queries

  – Named SQL queries can also be defined in the mapping document, in this case, you do not need to call addEntity()

    - *Named sql query using the <sql-query> maping element*

```
<sql-query name="persons">
    <return alias="person" class="Person"/>
        SELECT person.NAME AS {person.name},
        person.AGE AS {person.age},
        person.SEX AS {person.sex}
        FROM PERSON person
        WHERE person.NAME LIKE :namePattern
</sql-query>
```

    - *Execution of a named query*

```
List people = session.getNamedQuery("persons")
        .setString("namePattern", namePattern).setMaxResults(50)
        .list();
```

Hibernate?
- How to page through the

Hibernate new features

Hibernate New Features

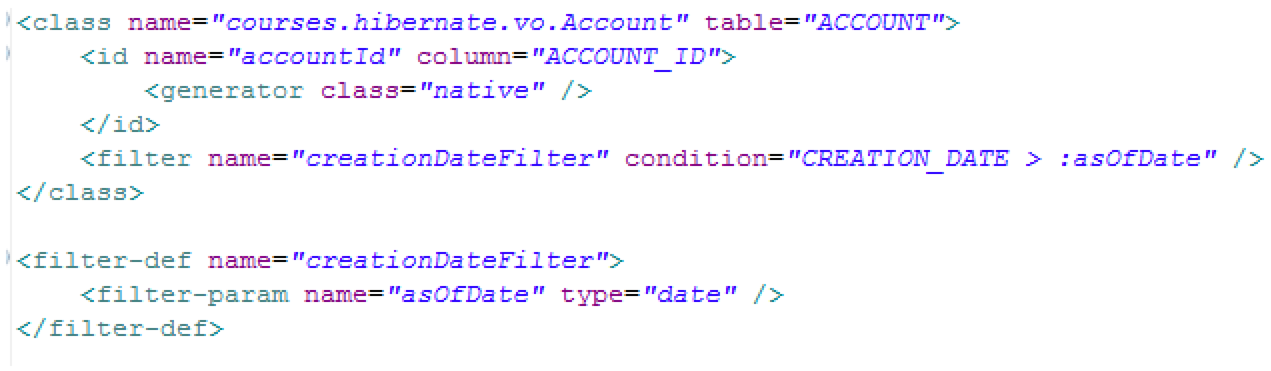
- Data Filtering
- Interceptors
- Calling Triggers and Stored Procedures

Data Filtering

- Limit the amount of data visible without modifying query parameters
- Often used for security purposes
  - Users often only have access to certain levels of information
- Step to set up Data Filters
  - Define the filter within the mapping file of the targeted entity
    - Identify the attributes to filter on, and their types
  - Apply the filter on the desired class or collection by indicating it within the <class> or <*collection-type*> *tags*
  - After obtaining a session with which to perform your actions, enable the appropriate filter, setting any applicable parameters

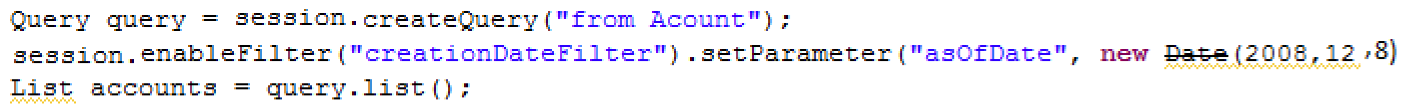
Data Filtering

- Account mapping file



```
<class name="courses.hibernate.vo.Account" table="ACCOUNT">
    <id name="accountId" column="ACCOUNT_ID">
        <generator class="native" />
    </id>
    <filter name="creationDateFilter" condition="CREATION_DATE > :asOfDate" />
</class>

<filter-def name="creationDateFilter">
    <filter-param name="asOfDate" type="date" />
</filter-def>
```

- Account class filter



```
Query query = session.createQuery("from Acount");
session.enableFilter("creationDateFilter").setParameter("asOfDate", new ~~Date~~(2008,12 ,8)
List accounts = query.list();
```

Interceptors

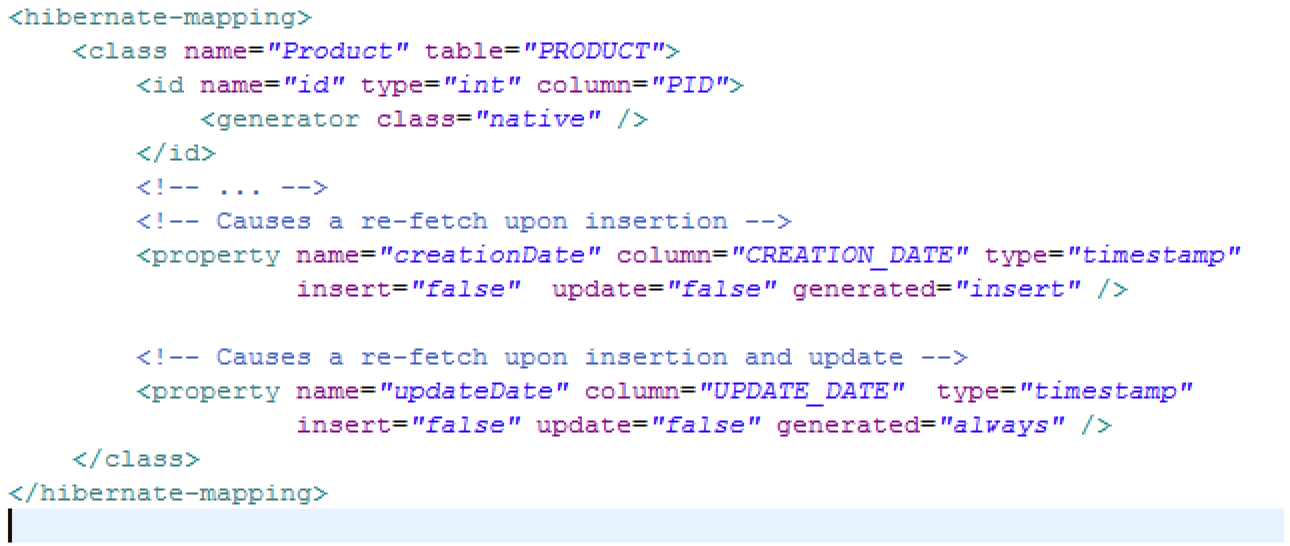
- Callbacks from the session allowing the application to inspect and/or manipulate properties of a persistent object
  - Before it Is saved, updated, deleted or loaded
- Implemented one of two ways
  - Implement Interceptor directly
  - Extend EmptyInterceptor (preferred)
- Comes in two flavors
  - Session-scoped
    - Specified when a session is opened
    - SessionFactory.openSession(Interceptor)
  - SessionFactory-scoped
    - Registered on the configuration during factory creation
    - Applies to all sessions

Interceptors

- Creating Interceptor
  1. Extend the EmptyInterceptor class
  2. Implement the desired callback methods
     - afterTransactionBegin(…)
     - afterTransactionCompletion (…)
     - onSave (…)
     - onDelete(…)
     - onLoad(…)
     - etc...
  3. Configure the interceptor use
     - Either during factory creation
     - After obtaining a session

Triggers

- Identify columns that are modified automatically by the database in the object mapping file
  - generated="insert | always“
  - Also need to tell Hibernate NOT to insert or update these columns, as appropriate
- Hibernate will re-read the object as appropriate
  - For insert, after the insert statement is executed
  - For always, after insert or update statements
- Setting up Triggers



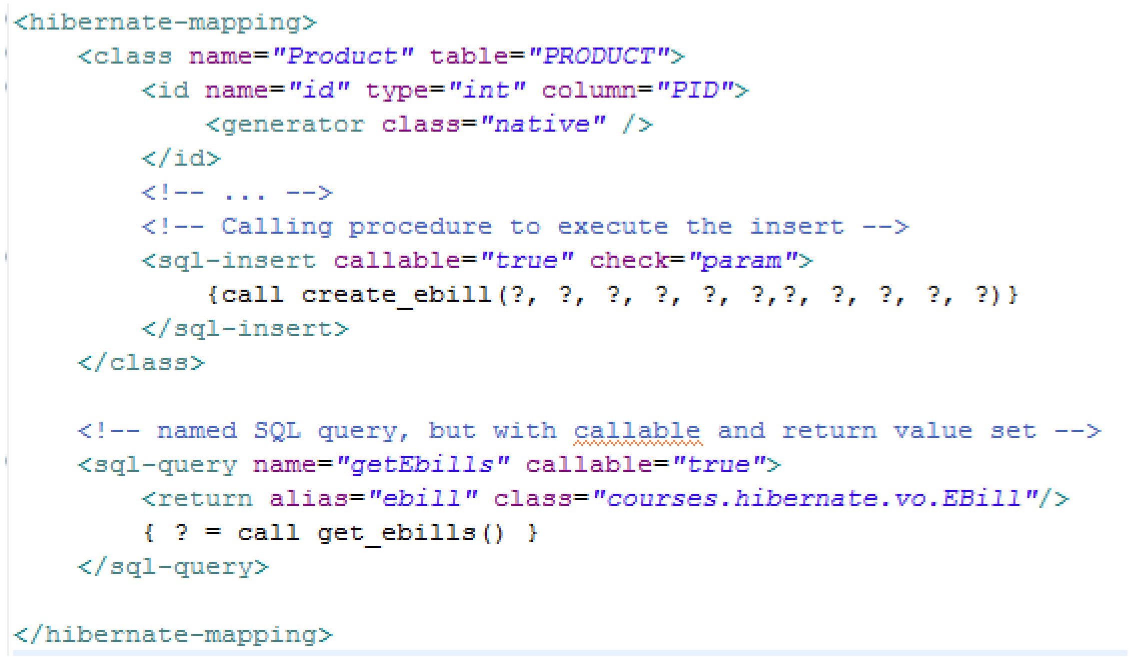
```
<hibernate-mapping>
    <class name="Product" table="PRODUCT">
        <id name="id" type="int" column="PID">
            <generator class="native" />
        </id>
        <!-- ... -->
        <!-- Causes a re-fetch upon insertion -->
        <property name="creationDate" column="CREATION_DATE" type="timestamp"
                  insert="false"  update="false" generated="insert" />

        <!-- Causes a re-fetch upon insertion and update -->
        <property name="updateDate" column="UPDATE_DATE"  type="timestamp"
                  insert="false" update="false" generated="always" />
    </class>
</hibernate-mapping>
```

Stored Procedures

- Calling Stored Procedures
  - For querying, similar syntax and process as named sql-query
    - Defined inside or outside the class tags in the mapping file
    - If returning a value, can set an alias and return type
  - For insert, update, or delete, must be defined inside the class tag
  - Must set the ‘callable’ attribute

Stored Procedures

- Stored Procedures Setup



```
<hibernate-mapping>
    <class name="Product" table="PRODUCT">
        <id name="id" type="int" column="PID">
            <generator class="native" />
        </id>
        <!-- ... -->
        <!-- Calling procedure to execute the insert -->
        <sql-insert callable="true" check="param">
            {call create_ebill(?, ?, ?, ?, ?, ?,?, ?, ?, ?, ?)}
        </sql-insert>
    </class>

    <!-- named SQL query, but with callable and return value set -->
    <sql-query name="getEbills" callable="true">
        <return alias="ebill" class="courses.hibernate.vo.EBill"/>
        { ? = call get_ebills() }
    </sql-query>

</hibernate-mapping>
```

Points to remembers

- Data Filtering
- Interceptors and Events
- Calling Triggers and Stored Procedures

Hibernate demo

Hibernate practice

Thank You

Revision History

Date	Version	Description	Updated by	Reviewed and Approved By
01/01/2011	1.0	Initialize	Hien Vo	
06/29/2016	1.1	Update Image and template	Lam Tang	

CSC

BUSINESS SOLUTIONS
TECHNOLOGY
OUTSOURCING