Object-relational Mapping
Prerequisites

• Solid knowledge in Java including but not limited to Java Class Library, JDBC, Annotations and Generics
• Solid knowledge in Relational databases, SQL
• XML
• Common programming pattern like DAO, Inversion of Control / Dependency Injection, ...
What is Object-relational Mapping?

• Wikipedia: “...technique for converting data between incompatible type systems in object-oriented programming languages. This creates, in effect, a ‘virtual object database’ that can be used from within the programming language”

• An object-relational mapper is a layer between the application layer and the database - it handles the mapping between objects (applications) and relations (database)

• Until recently the most efficient way to store data was in a relational database
  – Store large amount of data in a structured way to allow efficient queries
Why use O-RM?

- Eliminates all of the 'hand' mapping in Java from a SQL ResultSet to a Java POJO
- Reduces the amount of work required to the persistence code base with a domain data model and/or relational data model change
- Leverages large persistence library to avoid developing solutions to problems that others have already solved
- Avoids low level JDBC and SQL code (not portable)
- Leverages object oriented programming and object model usage
- Provides database and schema independence
- Object-relational impedance mismatch
Object-relational gap

- There is a paradigm mismatch between OO and Relational Algebra

<table>
<thead>
<tr>
<th>The object</th>
<th>Relation Gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>transient</td>
<td>persistent</td>
</tr>
<tr>
<td>classes</td>
<td>tables</td>
</tr>
<tr>
<td>• inheritance / polymorphism</td>
<td></td>
</tr>
<tr>
<td>attributes, properties</td>
<td>column</td>
</tr>
<tr>
<td>objects</td>
<td>rows</td>
</tr>
<tr>
<td>• Instance attributes</td>
<td>• field</td>
</tr>
<tr>
<td>• Implicit ID: reference</td>
<td>• Explicit ID: primary key</td>
</tr>
</tbody>
</table>
Object-relational gap cont.

<table>
<thead>
<tr>
<th>The object</th>
<th>Relation Gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>relations</td>
<td>relations</td>
</tr>
<tr>
<td>• references</td>
<td>• Foreign keys</td>
</tr>
<tr>
<td>• unidirectional</td>
<td>• bidirectional</td>
</tr>
<tr>
<td>• Ordered (lists)</td>
<td></td>
</tr>
<tr>
<td>behavior centric</td>
<td>data centric</td>
</tr>
<tr>
<td>• navigate object</td>
<td>• operate on set</td>
</tr>
<tr>
<td>• methods</td>
<td>• stored procedures</td>
</tr>
<tr>
<td>• Tight behavior-data coupling</td>
<td>• Triggers</td>
</tr>
<tr>
<td>• Data hiding, encapsulation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• permissions</td>
</tr>
</tbody>
</table>
The Java Persistence API (JPA)

- JPA bridges the gap between object-oriented domain models and relational database systems
  - JPA provides a POJO persistence model for object-relational mapping
  - Mapping is metadata driven
  - Provided by the javax.persistence package
- Implementations: Hibernate, DataNucleus, OpenJPA, EclipseLink, ...
- Java persistence consists of four areas
  - Java Persistence API
  - Object/relational mapping metadata: External XML or annotation based
  - Query Language (JPQL)
  - The Java Persistence Criteria API
JPA main features

- **POJO Persistence**: there is nothing special about the objects (entities) being persisted, any existing non-final object with a default constructor can be made persistable.

- **Non-intrusiveness**: the persistence API exists as a separate layer from the persistent objects; i.e. persisted objects are not aware of the persistence layer.

- **Object queries**: JPA offers a powerful query language to query across entities and their relationships without having to use concrete foreign keys or database columns.
Some History

• JPA 1.0 was developed by the EJB software expert group as part of the EJB 3.0 specification (JSR-220)
  – Developed as replacement for EJB 2.0 beans
  – was heavily influenced by Gavin King (founder of Hibernate)
  – Final Release May 2006
Some history cont.

• JPA 2.0 specified by JSR 317
  – final Release: December 2009
  – Includes new features like Expanded ORM functionality, Criteria query API, Standardization of metadata for DDL generation, ...
  – Part of Java 6 EE

• JPA 2.1 by JSR 338
  – final Release May 2013
  – New Features: Converters, Criteria Update and Deletes, Stored procedures, DDL generation, Entity graphs, ...
Entity

- An Entity refers to a logical collection of data that can be stored or retrieved as a whole
- An entity is a lightweight persistence domain object
- Typically, represents a table in a relational database
- An Entity class must follow these requirements (JPA)
  - Annotated with the `javax.persistence.Entity` annotation
  - Have a public or protected, no-argument constructor
  - Not be declared final, including methods or persistence instance variables
  - May extend both entity and non-entity classes
  - Persistence instance variables (fields) must be declared private protected or package-private, and can be accessed directly by the entity class methods
The entity Lifecycle

* = Extended persistence context
Annotations

- `@Entity` specifies that the class is an entity
- If the table name is different from the entity name, the `@Table` annotation will be used to specify the name
- The primary key of an entity is marked by the `@Id` annotation
- `@Column` annotation is used to specify the name of the column, if it's different from the property name
- To persist temporal data types in Java (`java.util.Date` and `java.util.Calendar`), use `@Temporal`
- `@Transient` is used to specify attributes which are part of the entity, but are not required to be persisted
Mapping Relationship

• Relationships between Entities can be defined using
  – @OneToOne
  – @ManyToOne and @OneToMany
  – @ManyToMany

• Join tables and columns can be configured using @JoinColumn and @JoinTable
Mapping inheritance Hierarchies

- done using the @Inheritance Annotation
- Multiple strategies available:
  - Single Table: simple, not normalized, no joins required, can result in large tables
  - Joined Subclass: normalized, Queries cause joins (poor performance for deep hierarchies, polymorphic queries and relationships)
  - Table per Class: not normalized, Polymorphic queries cause union (poor performance), not mandatory
Case study: Library

- The library contains several books
- The library can loan one or more books to a person
- Actors: Library, Book, Person, Loan
public class Person {
    private long id;
    private String name;
    private String surname;
    private Date birthday;
    private Set<Loan> loans;
    public Person() {} // getter and setter..}

public class Loan {
    private long id;
    private Date fromDate;
    private Date toDate;
    private Person person;
    private Book book;
    public Loan() {} // getter and setter
}

public class Book {
    private long id;
    private String title;
    private String description;
    public Book() {} // getter and setters..
}
@Entity
@Entity(name = "loans")
public class Loan implements Serializable {
    private static final long serialVersionUID = 35534737290088846840L;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="id")
    private long id;

    @Column(name="from_date")
    @Temporal(TemporalType.DATE)
    private Date fromDate;

    @Column(name="to_date")
    @Temporal(TemporalType.DATE)
    private Date toDate;

    @ManyToOne(cascade=CascadeType.ALL)
    @JoinColumn(name="person_id")
    private Person person;

    @ManyToOne(cascade=CascadeType.ALL)
    @JoinColumn(name="book_id")
    private Book book;

    public Loan() {
        // getter and setter..
    }
}
The Person Entity

@Entity
@Table(name = "persons")
public class Person implements Serializable {
    private static final long serialVersionUID = 967058688305386270L;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="id")
    private long id;

    @Column(name="name")
    private String name;

    @Column(name="surname")
    private String surname;

    @Column(name="birthday")
    @Temporal(TemporalType.DATE)
    private Date birthday;

    @OneToMany(mappedBy="person", orphanRemoval=true, cascade=CascadeType.ALL)
    private Set<Loan> loans;

    public Person() {}
    // getter and setter..
}
The Entity Manager

- manages the state and life cycle of all entities within a given persistence context
- Created using the EntityManagerFactory obtained by the Persistence Class
Persistent Units and the META-INF/persistence.xml file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0">
    <persistence-unit name="examplePersistenceUnit" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <class>net.stoltmann.talks.jpaintroduction.step2.Book</class>
        <class>net.stoltmann.talks.jpaintroduction.step2.Loan</class>
        <class>net.stoltmann.talks.jpaintroduction.step2.Person</class>
        <properties>
            <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:mem:LibraryDb"/>
            <property name="javax.persistence.jdbc.user" value=""/>
            <property name="javax.persistence.jdbc.password" value=""/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
            <property name="hibernate.show_sql" value="false"/>
            <property name="hibernate.format_sql" value="false"/>
            <property name="hibernate.hbm2ddl.auto" value="create"/>
        </properties>
    </persistence-unit>
</persistence>
```
The Entity Manager cont.

• Creation:

(EntityManagerFactory emf = Persistence.createEntityManagerFactory("LibraryPersistenceUnit");
EntityManager em = emf.createEntityManager();

• Persist an Entity

Book book = new Book();
book.setTitle("X");
book.setDescription("A");
book.setGenre(Genre.ROMANCE);

Book book = …
EntityTransaction transaction = em.getTransaction();
transaction.begin();
em.persist(book);
transaction.commit();

takes the entity instance, adds it to the context and makes that instance managed (i.e. future updates to the entity will be tracked)
The Entity Manager cont.

```java
System.out.println(book);
em.detach(book);

book = em.find(Book.class, 1L);
book.setTitle("Y");
transaction = em.getTransaction();
transaction.begin();
book = em.merge(book);
transaction.commit();

book = em.find(Book.class, 1L);
System.out.println(book);
transaction = em.getTransaction();
transaction.begin();
em.remove(book);
transaction.commit();
```

- Book [id=1, title=X, description=A, genre=ROMANCE]
- Remove the given entity from the persistence context
- Lookup the Book using the PK value 1.
- Creates a new instance of Book, copies the state from the supplied entity, and makes the new copy managed.
- Book [id=1, title=Y, description=A, genre=ROMANCE]
- Remove the entity.
The Entity Manager cont.

```java
EntityTransaction transaction = em.getTransaction();
transaction.begin();

Book book = new Book();
book.setDescription("A");
em.persist(book);
System.out.println(book);

book.setDescription("B");
System.out.println(book);
em.flush();
transaction.commit();

book = em.find(Book.class, 2L);
System.out.println(book);

em.close();
```

Book [id=2, title=null, description=A, genre=null]

Book [id=2, title=null, description=B, genre=null]

Synchronize the persistence context to the underlying database.

Book [id=2, title=null, description=B, genre=null]
Java Persistence Query Language (JPQL)

• Platform independent object-oriented query language

• Main differences with SQL:
  – JPQL operates on classes and objects (entities)
  – SQL operates on tables, columns and rows

• Therefore: result set of a JPQL query is a collection of entities, rather than a list of tuples

• JPQL supports three types of statements: SELECT, UPDATE, DELETE
Select Queries

• **SELECT** b **FROM** Book b
  – Book is the domain that we want to query and b is intended as an identifier of type Book
  – Identifier variables can be used in other clauses of the same query (e.g. WHERE)

• Path Expressions: Given an identifier variable, we can use dot notation to access a specific field of the entity
  – **SELECT** p **FROM** People p **WHERE** p.loans is **NOT EMPTY**

• Example:
  
  ```java
  TypedQuery<Book> query = em.createQuery("SELECT b FROM Book b", Book.class);
  List<Book> resultList = query.getResultList();
  System.out.println(resultList);
  ```
JPQL: JOINS

• Joins can be either specified in the WHERE clause or in the FROM clause by means of the JOIN operator

<table>
<thead>
<tr>
<th>JOIN in WHERE-clause</th>
<th>JOIN in From-Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT l FROM Person p, Loan l</td>
<td>SELECT l FROM Person p</td>
</tr>
<tr>
<td>WHERE p = l.person AND p.name = ‘NA42’</td>
<td>JOIN p.loans l</td>
</tr>
<tr>
<td></td>
<td>WHERE p.name = ‘NA42’</td>
</tr>
</tbody>
</table>
Query Types

• Two types of queries:
  – **Named queries**: intended to be stored and reused
    • defined using the `@NamedQuery(name, query)` annotation on the entity class
    • `TypedQuery<Book> query = em.createQuery("findAllBooks", Book.class);`
  – **Dynamic queries**: created and executed on the fly
    • `TypedQuery<Book> query = em.createQuery("SELECT b FROM Book b", Book.class);`
    – The returned query object can be used to set parameters (by number or name) and to execute the query