Design and Analysis of Information Systems (MAS)

Mariusz Trzaska, Ph. D.
mtrzaska@pjwstk.edu.pl

Lecture 10

Utilization of the Relational Model in Object-oriented Programming Languages (2)

Outline

- The impedance mismatch and its consequences
- Different approaches to solve the problem
- Microsoft LINQ
- Hibernate
  - An introduction,
  - A class/object's and attributes mapping,
  - An association's mapping,
  - An inheritance's mapping,
  - Multi-value attributes mapping.
- The summary

The Impedance Mismatch

- Connecting:
  - An object model from a programming language,
  - A relational model from a data source
- causes the impedance mismatch.

```
// [...]

// Execute the query
ResultSet result = db_statement.executeQuery("select * from employee");

// Process results
while (result.next() )
{
        System.out.println ("ID : " + result.getInt("ID"));
        System.out.println ("Name : " + result.getString("Name"));
}
```

- As the result we process atomic values describing an object (e.g. *String* as a name) rather then the object itself.

The Impedance Mismatch (2)

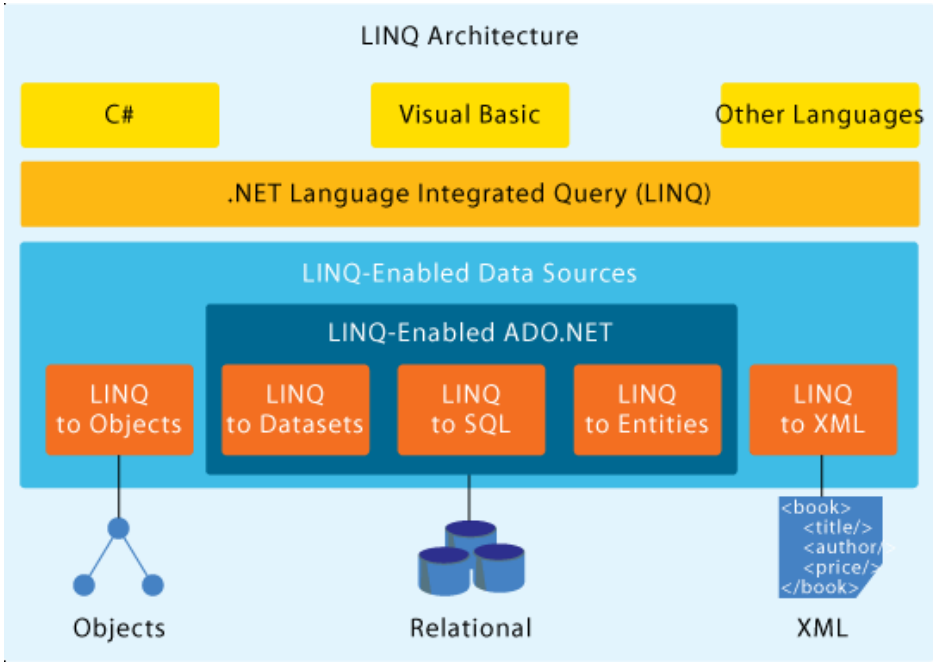
- Different solutions
  - Using the same model in both programming language and a date source.
    - It is unlikely that somebody would like to abandon object-oriented constructs, i.e. inheritance.
    - Introducing important functionalities from a database to a programming language, e.g. a query language.
      - Microsoft C# and LINQ
      - A persistency layer, e.g. Trzaska M.: Data Migration and Validation Using the Smart Persistence Layer 2.0. Acta Press. ISBN: 978-0-88986-951-6. November 12 ñ 14, 2012
  - Utilization of persistency libraries, e.g. Hibernate.

LINQ

- Language Integrated Query
- Designed by Anders Hejlsberg, who
  - is the first one who developed an IDE (Borland Turbo Pascal),
  - created the TypeScript.
- An existing programming language has been equipped with a query language similar to the SQL.
- Similar solution for Java 8+: functional streams in `java.util.stream`

LINQ (2)

- Thanks to this solution the impedance mismatch is significantly reduced.
- Additional benefits:
  - Utilization of the metadata during the runtime,
  - Compilation-time type control,
  - Support for IntelliSense.
- Different flavours:
  - LINQ to Objects,
  - LINQ to XML,
  - LINQ to ADO.NET,
  - LINQ to JSON (Json.NET),
  - …
- A revolution?



LINQ (3)

- Examples

```
var locals = from c in customers
             where c.ZipCode == 91822
             select new { FullName = c.FirstName + " " +
                          c.LastName, HomeAddress = c.Address };

IEnumerable<Product> x =
         from p in products
         where p.UnitPrice >= 10
         select p;

IEnumerable<Product> MostExpensive10 =
         products.OrderByDescending(p => p.UnitPrice).Take(10);

var custOrders =
         from c in customers
         join o in orders on c.CustomerID equals o.CustomerID
         select new { c.Name, o.OrderDate, o.Total };

IEnumerable<Product> orderedProducts1 =
         from p in products
         orderby p.Category, p.UnitPrice descending, p.Name
         select p;
```

LINQ (4)

- Examples – cont.

```
var q =
        from c in db.Customers
        where c.City == "London"
        select c;

foreach (Customer c in q)
        Console.WriteLine(c.CompanyName);



var q =
        from o in db.Orders
        where o.ShipVia == 3
        select o;
foreach (Order o in q) {
        if (o.Freight > 200)
                SendCustomerNotification(o.Customer);
        ProcessOrder(o);
}
```

The Hibernate

- The Hibernate creators described it:
  - *Hibernate is a powerful, high performance object/relational persistence and query service*
- Multi-platforms: Java, MS .NET, C++, etc.
- The project started in 2001 as open source:
  - 76 000 core code lines,
  - 36 000 unit test lines,
  - 3000 downloads each day.
- http://www.hibernate.org/

The Hibernate (2)

- Currently, they write: *More than an ORM, discover the Hibernate galaxy*.
- The Hibernate galaxy inculdes:
  - Hibernate ORM. Domain model persistence for relational databases.
  - Hibernate Search. Full-text search for your domain model.
  - Hibernate Validator. Annotation based constraints for your domain model.
  - Hibernate OGM. Domain model persistence for NoSQL datastores.
  - Hibernate Tools. Command line tools and IDE plugins for your Hibernate usages.
- Unfortunately, it does not completely eliminate the impedance mismatch problem.

Hibernate - performance

- The creators claim that the library is really fast:
  - Objects cache,
  - Query results cache,
  - No updates for not modified objects,
  - Efficient collections management,
  - Joining many changes in one UPDATE,
  - Lazy initialization of objects.

The Hibernate – a test environment

- It is based on the official tutorial, but we create our own business classes.
- The Hibernate works using the JDBC (the default behaviour).
- As a database we can use:
  - A typical solution, e.g.: MariaDB , MySQL;
  - Light systems written in Java, e.g. H2, HSQL or Apache Derby.
    - Usually, you only need to use a small jar file containing all the necessary components.
    - For test purposes, it is convenient to use the *in-memory* mode.

The Hibernate – a test environment (2)

- Utilization of the H2 database:
  - Download the archive.
  - Extract it to any folder.
  - Copy the *bin/*.jar* file to the *lib* folder in your Java project folder (the folder must be configured in the IDE as a source of additional libraries).
  - Optionally one can configure the IDE to download the required library from the Maven repository, e.g. *com.h2database:h2:1.4.199.*

The Hibernate – a test environment (3)

- Utilization of the H2 database – cont.
  - Starting the database:
    - in classic server mode - file *bin/h2.bat* or *bin/h2.sh* (a simple configuration console via browser is available).
    - simplified – the Hibernate will start it automatically after proper configuration (see further). This requires adding the previously mentioned jar file to the project launch libraries in the IDE.

The hibernate.cfg.xml configuration file

- Location: the root source folder.

```
<!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">org.h2.Driver</property>
        <property name="connection.url">jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE</property>
        <!--<property name="connection.url">jdbc:h2:~/db-test.h2</property>-->
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>

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

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.H2Dialect</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>

        <!-- Enable Hibernate stats in the logs -->
        <property name="hibernate.generate_statistics">true</property>

        <!- Full names of the annotated entity class -->
        <mapping class="mt.mas.hibernate.Movie"/>
        <mapping class="mt.mas.hibernate.Actor"/>
    </session-factory>
</hibernate-configuration>
```

The hibernate.cfg.xml configuration file (2)

- Depending on the selected mode for the H2 datbase, e.g.:
  - *in-memory* mode. Convenient for testing (without preserving data permanently), the DB management system is started automatically.

```
<!-- Database connection settings -->
<property name="connection.driver_class">org.h2.Driver</property>
<property name="connection.url">jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE</property>
<property name="connection.username">sa</property>
<property name="connection.password"></property>
```

  - *file* mode. Requires the server H2 to be started. You may also need to provide additional settings, e.g. a password.

```
<!-- Database connection settings -->
<property name="connection.driver_class">org.h2.Driver</property>
<property name="connection.url">jdbc:h2:~/db-test.h2</property>
<property name="connection.username">sa</property>
<property name="connection.password">sa</property>
```

The Hibernate - Basics

- Let's create a simple application allowing:
  - storing information about movies,
  - linking movies with actors.
- Each business class, which is going to use full power of the Hibernate, needs a special attribute used to identification instances (entities).
  - `private long id;`
  - managed by the Hibernate.
- Recommended JavaBean convention:
  - `set…()`,
  - `get…()`.

The mappings

- Unfortunately, Hibernate does not work fully automatically (just like other ORMs) and one needs to refine the way the DB (relational) structure is mapped to object-oriented (Java).
- Available approaches to mapping:
  - native Hibernate annotations,
  - JPA annotations,
  - XML mapping file.
- Which one to choose?
- Advantages and disadvantages?

The mapping – an example

- A standard Java class that stores information about a movie.
- Several attributes.
- Setters and getters.
- No need to inherit from a special superclass.

```
public class Movie {
    public enum MovieCategory {Unknown, Comedy, SciFi}

    private LocalDate releaseDate;
    private String title;
    private MovieCategory movieCategory;

    public Movie(String title, LocalDate releaseDate) {
        this.releaseDate = releaseDate;
        this.title = title;
    }

    public LocalDate getReleaseDate() {
        return releaseDate;
    }
    public void setReleaseDate(LocalDate releaseDate) {
        this.releaseDate = releaseDate;
    }

    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }

    public MovieCategory getMovieCategory() {
        return movieCategory;
    }
    public void setMovieCategory(MovieCategory
movieCategory) {
        this.movieCategory = movieCategory;
    }
}
```

The mapping – an example (2)

- We use the **JPA annotations.**
- The class is marked with the annotation: **`@javax.persistence.Entity`** (optional parameter: `Name`).
- A `public`/`protected` parametrless constructor is required.
- Additional information using the annotation: **`@javax.persistence.Table`**.

```
@Entity(name = "Movie")
public class Movie {
    private long id;
    private LocalDate releaseDate;
    private String title;
    private MovieCategory movieCategory;

    /** Required by Hibernate */
    public Movie() {}

    public Movie(String title,
                 LocalDate releaseDate) {
        this.releaseDate = releaseDate;
        this.title = title;
    }

    // [...]
}
```

The mapping – identifier

- We have added an attribute that acts as an identifier.
- We've created setter and getter for it.
- The getter has been marked with the appropriate annotations:
  - **`@Id`**
  - **`@GeneratedValue`**
  - **`@GenericGenerator`**
- It is also possible to mark the attribute rather then the getter.

```
@Entity(name = "Movie")
public class Movie {
    private long id;
    private LocalDate releaseDate;
    private String title;
    private MovieCategory movieCategory;

    /** Required by Hibernate */
    public Movie() {}

    public Movie(String title, LocalDate
releaseDate) {
        this.releaseDate = releaseDate;
        this.title = title;
    }

    @Id
    @GeneratedValue(generator="increment")
    @GenericGenerator(name="increment", strategy
= "increment")
    public long getId() {
        return id;
    }

    private void setId(long id) {
        this.id = id;
    }

    // [...]
}
```

The mapping – simple attributes

- The annotation: **`@javax.persistence.Basic`**

  is utilized for:

  - simple types,
  - a few others like `String` or date related.

  It is possible to ommit it.

  Optional parameters:

  - `optional`. desribes if `nulls` are available (`True`),
  - `fetch`. How to retrieve the value (`Eager`).
- Annotation **`@javax.persistence.Column`**. More customizations, e.g. column name in a DB.
- Annotation **`@javax.persistence.Type`**. Defines the DB type.
- It is also possible to persist custom types.

```java
@Entity(name = "Movie")
public class Movie {
    private LocalDate releaseDate;
    private String title;
    // [...]

    @Basic
    public LocalDate getReleaseDate() {
        return releaseDate;
    }
    public void setReleaseDate(LocalDate releaseDate)
{
        this.releaseDate = releaseDate;
    }

    @Basic
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }

    @Override
    public String toString() {
        // The code could be optimized.
        var sb = new StringBuilder();

        sb.append(String.format("Movie: %s released on
%s (#%s @%s)", getTitle(), getReleaseDate(), getId(),
super.hashCode()));
        return sb.toString();
    }
}
```

The mapping – derived attributes

- The annotation: **`@javax.persistence.Transient`**
  - means that Hibernate ignores the specified attribute or getter and the associated attribute.
  - allows the implementation of methods (mainly getters) used by derived attributes, e.g.
    - `getName()`,
    - `getAge()`.

```
@Entity(name = "Actor")
public class Actor {
    private long id;
    private String firstName;
    private String lastName;
    private LocalDate birthDate;

    // [...]

    @Basic
    public LocalDate getBirthDate() {
        return birthDate;
    }

    public void setBirthDate(LocalDate birthDate) {
        this.birthDate = birthDate;
    }

    @Transient
    public String getName() {
        return getFirstName() + " " + getLastName();
    }

    @Transient
    public int getAge() {
        return Period.between(getBirthDate(),
                LocalDate.now()).getYears();
    }

    @Override
    public String toString() {
        return String.format("Actor: %s born on %s,
age: %s (#%s @%s)", getName(), getBirthDate(),
getAge(), getId(), super.hashCode());
    }
}
```

The mapping – enum

- The annotation: **`@javax.persistence.Enumerated`**. Mapping of enums (enumerations). Additional parameters:
  - `EnumType.ORDINAL`. Utilizes a number approach,
  - `EnumType.STRING`. Uses an enum's name.

```
@Entity(name = "Movie")
public class Movie {
    public enum MovieCategory {Unknown, Comedy, SciFi}

    private LocalDate releaseDate;
    private String title;
    private MovieCategory movieCategory;

    // [...]

    @Enumerated
    public MovieCategory getMovieCategory() {
        return movieCategory;
    }
    public void setMovieCategory(MovieCategory movieCategory) {
        this.movieCategory = movieCategory;
    }

    @Override
    public String toString() {
        // The code could be optimized.
        var sb = new StringBuilder();

        sb.append(String.format("Movie: %s released on %s as %s (#%s @%s)", getTitle(), getReleaseDate(),
getMovieCategory(), getId(), super.hashCode()));

        return sb.toString();
    }
}
```

The mapping – complex attribute

- The annotation: (docs):
  - **`@javax.persistence.Embeddable`**
  - **`@javax.persistence.Embedded`**

```java
@Entity(name = "Actor")
public class Actor {

    private Address address;

    // [...]

    @Override
    public String toString() {
        return String.format("Actor: %s born on %s, age: %s, address: %s, movie:
            getName(), getBirthDate(), getAge(), getAddress() != null ? getAddres
            getMovie() != null ? getMovie().getTitle() : "---", getId(), super.ha
    }

    @Embedded
    public Address getAddress() {
        return address;
    }

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

```java
@Embeddable
public class Address {
    private String street;
    private String city;
    private String zipCode;

    // [...]

    public Address() { }

    @Basic
    public String getStreet() {
        return street;
    }

    @Basic
    public String getCity() {
        return city;
    }

    @Basic
    public String getZipCode() {
        return zipCode;
    }
}
```

The mapping – BLOB/LOBs

- Hibernate supports mapping of BLOBs/LOBs (*database Large Objects*).
  - Be careful with resource (memory) utilization.
  - Optimization techniques, e.g. streaming.
  - Mapping to various Java types, e.g.. `String` or access with a stream.
  - More information in the LOB's documentation.

Objects identity in Hibernate

- The `equals()` and `hashCode()` methods.
- Hibernate ensures that the same object (primary key) retrieved from the database in one session will have the same instance in the Java environment.
- Sometimes, your own implementation of the above-mentioned may be helpful. Use the attribute annotation: `@NaturalId`.
- More in the Hibernate documentation.

Working with the Hibernate - *Session*

- Create the *registry*.
- Create a *session factory*.
- Start the session.
- Start a transaction.
- **Execute some operations.**
- Commit the transaction and close the session.

```
StandardServiceRegistry registry = null;
SessionFactory sessionFactory = null;

try {
    registry = new StandardServiceRegistryBuilder()
            .configure() // configures settings from
hibernate.cfg.xml
            .build();
    sessionFactory = new MetadataSources(registry)
                        .buildMetadata()
                        .buildSessionFactory();

    Session session = sessionFactory.openSession();
    session.beginTransaction();

    // Do something within the session, e.g. create/retrieve objects,
    // etc.

    session.getTransaction().commit();
    session.close();
}
catch (Exception e) {
    e.printStackTrace();
    StandardServiceRegistryBuilder.destroy( registry );
}
finally {
    if (sessionFactory != null) {
        sessionFactory.close();
}
```

Hibernate – create objects

- Add information about movies.
  - In this step, the ids are uninitialized. They will be updated after the commiting of a transaction.

```
        try {
            // [...]
            System.out.println("Created movies:");
            var movie1 = new Movie("Terminator 1", LocalDate.of(1984, 10,26),
Movie.MovieCategory.SciFi);
            var movie2 = new Movie("Terminator 3", LocalDate.of(2003, 8,8),
Movie.MovieCategory.SciFi);

            System.out.println(movie1);
            System.out.println(movie2);

            Session session = sessionFactory.openSession();
            session.beginTransaction();
            session.save(movie1);
            session.save(movie2);
            session.getTransaction().commit();
            session.close();
        }
```

```
Created movies:
Movie: Terminator 1 released on 1984-10-26 as SciFi (#0 @1499588909)
Movie: Terminator 3 released on 2003-08-08 as SciFi (#0 @1339052072)
```

Hibernate – retrieving data

- There is a query language similar to SQL.
- As it can be seen, we work with objects rather then simple/atomic values (like in the JDBC).
- Retrieved objects contain valid values of primary keys.

```
        try {
            // [...]

            System.out.println("\nMovies from the db:");

            session = sessionFactory.openSession();
            session.beginTransaction();
            List<Movie> moviesFromDb = session.createQuery("from
Movie").list();
            for ( var movie : moviesFromDb) {
                System.out.println(movie);
            }
            session.getTransaction().commit();
            session.close();
        }

        // [...]
```

```
Movies from the db:
Movie: Terminator 1 released on 1984-10-26 as SciFi (#1 @1989924937)
Movie: Terminator 3 released on 2003-08-08 as SciFi (#2 @1842571958)
```

Hibernate – the log file

1:01:10 PM org.hibernate.Version logVersion INFO: HHH000412: Hibernate Core {5.4.1.Final}
1:01:11 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure WARN: HHH10001002: Using Hibernate built-in connection pool (not for production use!)
1:01:11 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator INFO: HHH10001005: using driver [org.h2.Driver] at URL [jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE]
1:01:11 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator INFO: HHH10001001: Connection properties: {password=****, user=sa}
1:01:11 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator INFO: HHH10001003: Autocommit mode: false
1:01:11 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl$PooledConnections <init> INFO: HHH000115: Hibernate connection pool size: 1 (min=1)
1:01:11 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJdbcConnectionAccess@423c5404] for (non-JTA) DDL execution was not in auto-commit mode; the Connection 'local transaction' will be committed and the Connection will be set into auto-commit mode.
Hibernate: **drop table Movie if exists**
1:01:11 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJdbcConnectionAccess@6add8e3f] for (non-JTA) DDL execution was not in auto-commit mode; the Connection 'local transaction' will be committed and the Connection will be set into auto-commit mode.
Hibernate: **create table Movie (id bigint not null, movieCategory integer, releaseDate date, title varchar(255), primary key (id))**
1:01:11 PM org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformInitiator initiateService INFO: HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]

Created movies:
Movie: Terminator 1 released on 1984-10-26 as SciFi (#0 @1499588909)
Movie: Terminator 3 released on 2003-08-08 as SciFi (#0 @1339052072)

Hibernate: **select max(id) from Movie**
Hibernate: **insert into Movie (movieCategory, releaseDate, title, id) values (?, ?, ?, ?)**
Hibernate: **insert into Movie (movieCategory, releaseDate, title, id) values (?, ?, ?, ?)**

Movies and actors from the db:
Hibernate: **select movie0_.id as id1_1_, movie0_.movieCategory as movieCat2_1_, movie0_.releaseDate as releaseD3_1_, movie0_.title as title4_1_ from Movie movie0_**
Movie: Terminator 1 released on 1984-10-26 as SciFi (#1 @1989924937)
Movie: Terminator 3 released on 2003-08-08 as SciFi (#2 @1842571958)
1:01:12 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl$PoolState stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE]

Hibernate – the data scheme

- Of course, Hibernate automatically creates an appropriate relational scheme/model in DB.
- After "generating" the data (stable version of the data model), the fragment of the ***hibernate.cfg.xml*** file should be commented/removed (otherwise the existing data will be deleted after the program has been started).

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

Associations in Hibernate

- Associations are (almost) automatically mapped to relationships in the database.
- Elements that need to be included are:
  - direction,
  - cardinality
  - the behavior of the implementation collection (on the Java side).

Adding a directed association

- We will create the *Actor* class, which will be related to the *Movie*.
- New entry in the *hibernate.cfg.xml* configuration file.

```
<mapping
class="mt.mas.hibernate.Actor
"/>
```

```
@Entity(name = "Actor")
public class Actor {
    private long id;
    private String firstName;
    private String lastName;
    private LocalDate birthDate;

    public Actor() {    }

    @Id
    @GeneratedValue(generator="increment")
    @GenericGenerator(name="increment", strategy =
                          "increment")
    public long getId() {
        return id;
    }

    @Basic
    public String getFirstName() {
        return firstName;
    }

    @Basic
    public String getLastName() {
        return lastName;
    }

    @Basic
    public LocalDate getBirthDate() {
        return birthDate;
    }

    // Other methods, setters, etc.
}
```

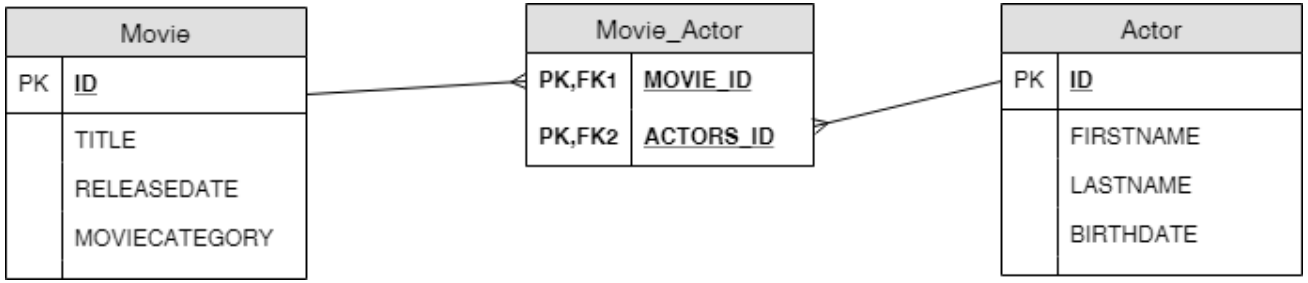
Adding a directed association (2)

- To the *Movie* class we will add information about actors playing in it.
- We use a `List` container (other ones are also supported).
- **`@OneToMany`** annotation

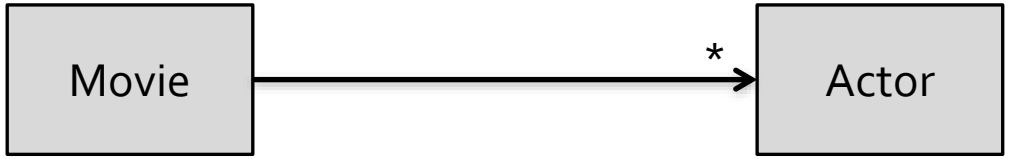
```java
@Entity(name = "Movie")
public class Movie {
    private List<Actor> actors = new ArrayList<>();

     // [...]

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    public List<Actor> getActors() {
        return actors;
    }

    private void setActors(List<Actor> actors) {
        this.actors = actors;
    }
}
```

Adding a directed association (3)

- As a result, we received:
  - the relational schema (the intermediate table was generated automatically, although it is not needed for 1- *).



  - a connection (Java references) from the *Movie* class to the *Actor* (but not the other way).



Utilization of the directed association

- We add the connection by modifying the Java container,
- Hibernate automatically detects it and updates the database,
- Similar "automation" exists for attributes.

```
var movie1 = new Movie("Terminator 1", LocalDate.of(1984, 10,26),
                    Movie.MovieCategory.SciFi);
var movie2 = new Movie("Terminator 3", LocalDate.of(2003, 8,8),
                    Movie.MovieCategory.SciFi);

var actor1 = new Actor("Arnold", "Schwarzenegger",
                    LocalDate.of(1947, 7, 30));
var actor2 = new Actor("Claire", "Danes", LocalDate.of(1979, 4,
12));
var actor3 = new Actor("Kristanna", "Loken",
                    LocalDate.of(1979, 10, 8));

movie2.getActors().add(actor1);
movie2.getActors().add(actor2);
movie2.getActors().add(actor3);

Session session = sessionFactory.openSession();
session.beginTransaction();
session.save(movie1);
session.save(movie2);
session.save(actor1);
session.save(actor2);
session.save(actor3);
session.getTransaction().commit();
session.close();

session = sessionFactory.openSession();
session.beginTransaction();
List<Movie> moviesFromDb = session.createQuery( "from Movie"
).list();
for ( var movie : moviesFromDb) {
    System.out.println(movie);
}
List<Actor> actorsFromDb = session.createQuery( "from Actor"
).list();
for ( var actor : actorsFromDb) {
    System.out.println(actor);
}
session.getTransaction().commit();
session.close();
```

Utilization of the directed association (2)

Created movies:
Movie: Terminator 1 released on 1984-10-26 as SciFi (#0 @612635506)
Actors: ---
Movie: Terminator 3 released on 2003-08-08 as SciFi (#0 @1997623038)
Actors: ---
Created actors:
Actor: Arnold Schwarzenegger born on 1947-07-30, age: 71 (#0 @2122267901)
Actor: Claire Danes born on 1979-04-12, age: 39 (#0 @987834065)
Actor: Kristanna Loken born on 1979-10-08, age: 39 (#0 @1185188034)
Hibernate: select max(id) from Movie
Hibernate: select max(id) from Actor
Hibernate: insert into Movie (movieCategory, releaseDate, title, id) values (?, ?, ?, ?)
Hibernate: insert into Movie (movieCategory, releaseDate, title, id) values (?, ?, ?, ?)
Hibernate: insert into Actor (birthDate, firstName, lastName, id) values (?, ?, ?, ?)
Hibernate: insert into Actor (birthDate, firstName, lastName, id) values (?, ?, ?, ?)
Hibernate: insert into Actor (birthDate, firstName, lastName, id) values (?, ?, ?, ?)
Hibernate: insert into Movie_Actor (Movie_id, actors_id) values (?, ?)
Hibernate: insert into Movie_Actor (Movie_id, actors_id) values (?, ?)
Hibernate: insert into Movie_Actor (Movie_id, actors_id) values (?, ?)

Utilization of the directed association (3)

Movies and actors from the db:
Hibernate: select movie0_.id as id1_1_, movie0_.movieCategory as movieCat2_1_, movie0_.releaseDate as releaseD3_1_, movie0_.title as title4_1_ from Movie movie0_
Hibernate: select actors0_.Movie_id as Movie_id1_2_0_, actors0_.actors_id as actors_i2_2_0_, actor1_.id as id1_0_1_, actor1_.birthDate as birthDat2_0_1_, actor1_.firstName as firstNam3_0_1_, actor1_.lastName as lastName4_0_1_ from Movie_Actor actors0_ inner join Actor actor1_ on actors0_.actors_id=actor1_.id where actors0_.Movie_id=?
Movie: Terminator 1 released on 1984-10-26 as SciFi (#1 @61334373)
Actors: ---
Hibernate: select actors0_.Movie_id as Movie_id1_2_0_, actors0_.actors_id as actors_i2_2_0_, actor1_.id as id1_0_1_, actor1_.birthDate as birthDat2_0_1_, actor1_.firstName as firstNam3_0_1_, actor1_.lastName as lastName4_0_1_ from Movie_Actor actors0_ inner join Actor actor1_ on actors0_.actors_id=actor1_.id where actors0_.Movie_id=?
Movie: Terminator 3 released on 2003-08-08 as SciFi (#2 @331918455)
Actors: Actor: Arnold Schwarzenegger born on 1947-07-30, age: 71 (#1 @263233676); Actor: Claire Danes born on 1979-04-12, age: 39 (#2 @1651795723); Actor: Kristanna Loken born on 1979-10-08, age: 39 (#3 @1406018450);
Hibernate: select actor0_.id as id1_0_, actor0_.birthDate as birthDat2_0_, actor0_.firstName as firstNam3_0_, actor0_.lastName as lastName4_0_ from Actor actor0_
Actor: Arnold Schwarzenegger born on 1947-07-30, age: 71 (#1 @263233676)
Actor: Claire Danes born on 1979-04-12, age: 39 (#2 @1651795723)
Actor: Kristanna Loken born on 1979-10-08, age: 39 (#3 @1406018450)

Utilization of the directed association (4)

Movie

ID	MOVIECATEGORY	RELEASEDATE	TITLE
1	2	1984-10-26	Terminator 1
2	2	2003-08-08	Terminator 3

MOVIE_ACTOR

MOVIE_ID	ACTORS_ID
2	1
2	2
2	3

Actor

ID	BIRTHDATE	FIRSTNAME	LASTNAME
1	1947-07-30	Arnold	Schwarzenegger
2	1979-04-12	Claire	Danes
3	1979-10-08	Kristanna	Loken

Adding a bidirectional association

- The relational diagram remains the same.
- `@ManyToOne` Annotation.
- It is necessary to ensure the consistency of both directions (dedicated logic in the method that creates the connection).
- You must use the `inverse` or `mappedBy` parameter.

```java
@Entity(name = "Movie")
public class Movie {
    private List<Actor> actors = new ArrayList<>();

    // [...]

    @OneToMany(
       mappedBy = "movie",
       cascade = CascadeType.ALL,
       orphanRemoval = true)
    private List<Actor> getActors() {
        return actors;
    }
    public void addActor(Actor actor) {
        getActors().add(actor);
        actor.setMovie(this);
    }
    public void removeActor(Actor actor) {
        getActors().remove(actor);
        actor.setMovie(null);
    }
}
```

```java
@Entity(name = "Actor")
public class Actor {
    private Movie movie;

    // [...]

    @ManyToOne
    public Movie getMovie() {
        return movie;
    }

    public void setMovie(Movie movie) {
        this.movie = movie;
    }
}
```

Adding a bidirectional association (2)

- Similarly, we map other numbers using the annotation:
  - `@ManyToMany`,
  - `@OneToOne`.
- Note on defining the "owner" of the association. Important when removing objects.
- Special association cases:
  - `@NotFound`. When no associated primary key was found.
  - `@Any`.
  - `@JoinFormula`, `@JoinColumnOrFormula`.

Multi-valued Attributes

- In the Hibernate they are called *Collection of values*.
- The difference to associations is that the values cannot be shared (and objects could be shared).
- They must be declared using an interface, not a specific implementation.
- The behavior of this repetitive attribute depends on the interface type (e.g., `List`, `Set`).

Multi-valued Attributes (2)

- For the `Actor` class we add a list of his/her urls.
- The annotation: **`@javax.persistence.ElementCollection`**
  - It means that the collection does not contain connections to other instances, but the list of items, e.g. `String` type.

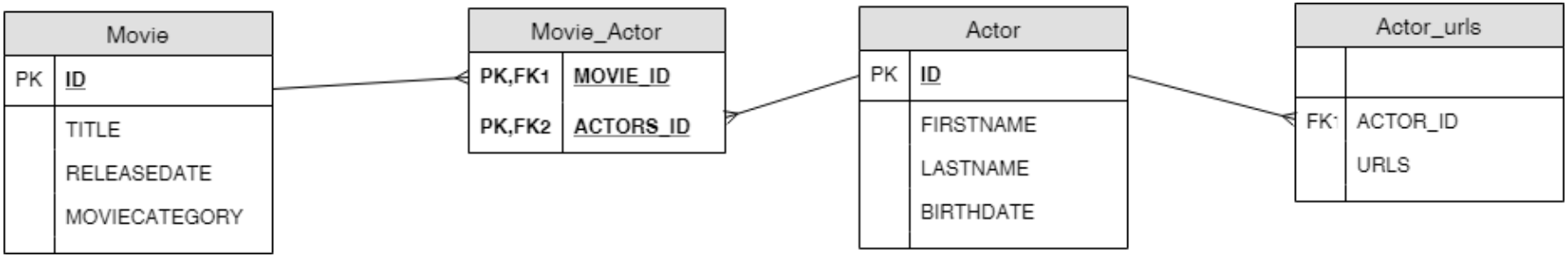
```
@Entity(name = "Actor")
public class Actor {

    // [...]

    private List<String> urls;

    @ElementCollection
    public List<String> getUrls() {
        return urls;
    }

    public void setUrls(List<String> urls) {
        this.urls = urls;
    }
}
```

```
actor1.setUrls(List.of("http://www.schwarzenegger.com/",
                       "https://pl.pinterest.com/schwarzenegger/",
                       "https://www.facebook.com/arnold"));
```

Actor: Arnold Schwarzenegger born on 1947-07-30, age: 71, movie: Terminator 3 (#1 @1989924937)
Hibernate: select urls0_.Actor_id as Actor_id1_1_0_, urls0_.urls as urls2_1_0_ from Actor_urls urls0_ where urls0_.Actor_id=?
[http://www.schwarzenegger.com/, https://pl.pinterest.com/schwarzenegger/, https://www.facebook.com/arnold]

Multi-valued Attributes (3)

- The updated relational scheme



Hibernate - inheritance

- Mapped superclass.
- Single table, Table Per Hierarchy - TPH.
- Joined table, table-per-subclass/type - TPT.
- Table per class, table-per-concrete-class - TPC.
- *(See also the previous lecture)*

Inheritance - mapped superclass

- Reflected only in the model, but not in the DB. There is no possibility to refer to the superclass.
- Only two tables will be created in the DB (repeating the contents of the superclass).

```
@MappedSuperclass
public class Account {
        @Id
        private Long id;
        private String owner;
        private BigDecimal balance;
        private BigDecimal interestRate;
        //Getters and setters are omitted for brevity
}
@Entity(name = "DebitAccount")
public class DebitAccount extends Account {
        private BigDecimal overdraftFee;
        //Getters and setters are omitted for brevity
}
@Entity(name = "CreditAccount")
public class CreditAccount extends Account {
        private BigDecimal creditLimit;
        //Getters and setters are omitted for brevity
}
```

Source: documentation of the Hibernate

Inheritance - single table

- One table containing elements from the superclass and all subclasses.
- Special column - discriminator.

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class Account {
          @Id
          private Long id;
          private String owner;
          private BigDecimal balance;
          private BigDecimal interestRate;
          //Getters and setters are omitted for brevity
}
@Entity(name = "DebitAccount")
public class DebitAccount extends Account {
          private BigDecimal overdraftFee;
          //Getters and setters are omitted for brevity
}
@Entity(name = "CreditAccount")
public class CreditAccount extends Account {
          private BigDecimal creditLimit;
          //Getters and setters are omitted for brevity
}
```

Source: documentation of the Hibernate

Hibernate - joined table

- Each class has its own table. Connections using relationships (master – foreign key).

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.JOINED)
public class Account {
          @Id
          private Long id;
          private String owner;
          private BigDecimal balance;
          private BigDecimal interestRate;
          //Getters and setters are omitted for brevity
}
@Entity(name = "DebitAccount")
public class DebitAccount extends Account {
          private BigDecimal overdraftFee;
          //Getters and setters are omitted for brevity
}
@Entity(name = "CreditAccount")
public class CreditAccount extends Account {
          private BigDecimal creditLimit;
          //Getters and setters are omitted for brevity
}
```

Source: documentation of the Hibernate

Inheritance - table per class

- Tables are generated for each subclass and the contents of the superclass is also placed in them.

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Account {
          @Id
          private Long id;
          private String owner;
          private BigDecimal balance;
          private BigDecimal interestRate;
          //Getters and setters are omitted for brevity
}
@Entity(name = "DebitAccount")
public class DebitAccount extends Account {
          private BigDecimal overdraftFee;
          //Getters and setters are omitted for brevity
}
@Entity(name = "CreditAccount")
public class CreditAccount extends Account {
          private BigDecimal creditLimit;
          //Getters and setters are omitted for brevity
}
```

Source: documentation of the Hibernate

Queries in the Hibernate

- Queries expressed using the *Criteria*
  - support for strong typing,
  - quite complicated construction.

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Person> criteria = builder.createQuery(Person.class);
Root<Person> root = criteria.from(Person.class);
criteria.select(root);
criteria.where(builder.equal(root.get(Person_.name), "John Doe"));

List<Person> persons =
entityManager.createQuery(criteria).getResultList();
```

Queries in the Hibernate (2)

- Sample queries in *Hibernate Query Language* (HQL) – similar to SQL.

```
List cats = session.createQuery(
    "from Cat as cat where cat.birthdate < ?")
    .setDate(0, date)
    .list();

List mothers = session.createQuery(
    "select mother from Cat as cat join cat.mother as mother where cat.name = ?")
    .setString(0, name)
    .list();

List kittens = session.createQuery(
    "from Cat as cat where cat.mother = ?")
    .setEntity(0, pk)
    .list();

Cat mother = (Cat) session.createQuery(
    "select cat.mother from Cat as cat where cat = ?")
    .setEntity(0, izi)
    .uniqueResult();]]

Query mothersWithKittens = (Cat) session.createQuery(
    "select mother from Cat as mother left join fetch mother.kittens");
Set uniqueMothers = new HashSet(mothersWithKittens.list());
```

The Summary (1)

- The impedance mismatch is a real, serious problem.
- There are two general approaches for solving it:
  - A modification of the programming language (platform) by introducing some DB functionalities (e.g. query language),
  - A creation of additional libraries making easier working with data.

The Summary (2)

- The first approach is represented by Microsoft C# together with the LINQ technology.
  - A query language (similar to the SQL) becomes a part of the programming language.
  - The impedance mismatch is significantly reduced. Hence we do not need perform OR mapping (at least in the theory).
  - As additional benefits we have e.g. type checking during the compilation.

The Summary (3)

- The second approach is represented by the Hibernate
  - The library really simplifies processing the data,
  - Unfortunately sometimes it requires identifiers rather then references.
- It seems that a much better solution is the first one (i.e. Microsoft LINQ).

Source files

Download source files for all MAS lectures

http://www.mtrzaska.com/plik/mas/mas-source-files-lectures