



# Modelowanie i Analiza Systemów informacyjnych (MAS)

dr inż. Mariusz Trzaska  
[mtrzaska@pjwstk.edu.pl](mailto:mtrzaska@pjwstk.edu.pl)

## Wykład 10

### Wykorzystanie modelu relacyjnego w obiektowych językach programowania (2)



# Zagadnienia

- Niezgodność impedancji i jej konsekwencje
- Różne sposoby rozwiązywania problemu
- Microsoft LINQ
- Hibernate
  - Wprowadzenie,
  - Mapowanie klas/obiektów oraz atrybutów,
  - Mapowanie asocjacji,
  - Mapowanie dziedziczenia,
  - Mapowanie atrybutów powtarzalnych.
- Podsumowanie

# Niezgodność impedancji

- Łączenie:
  - modelu obiektowego z języka programowania,
  - modelu relacyjnego ze składu danych

skutkuje zjawiskiem zwanym niezgodnością impedancji.

```
// [...]
// Wykonaj zapytanie
ResultSet result = db_statement.executeQuery("select * from employee");

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

- W efekcie zamiast operować na obiektach (np. *Pracownik*) działamy w oparciu o ich elementy, np. *String* z nazwiskiem.

# Niezgodność impedancji (2)

- Rozwiązania problemu
  - Stosowanie tego samego modelu w języku programowania oraz źródle danych.
    - Mało kto zdecyduje się na programowanie w modelu relacyjnym (programowanie strukturalne).
    - Można wprowadzić istotne udogodnienia z zakresu baz danych do języka programowania m. in. język zapytań.
      - Microsoft C# i LINQ i ewentualnie warstwa trwałości, np. [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.](#)
  - Wykorzystanie dedykowanych bibliotek, np. Hibernate.

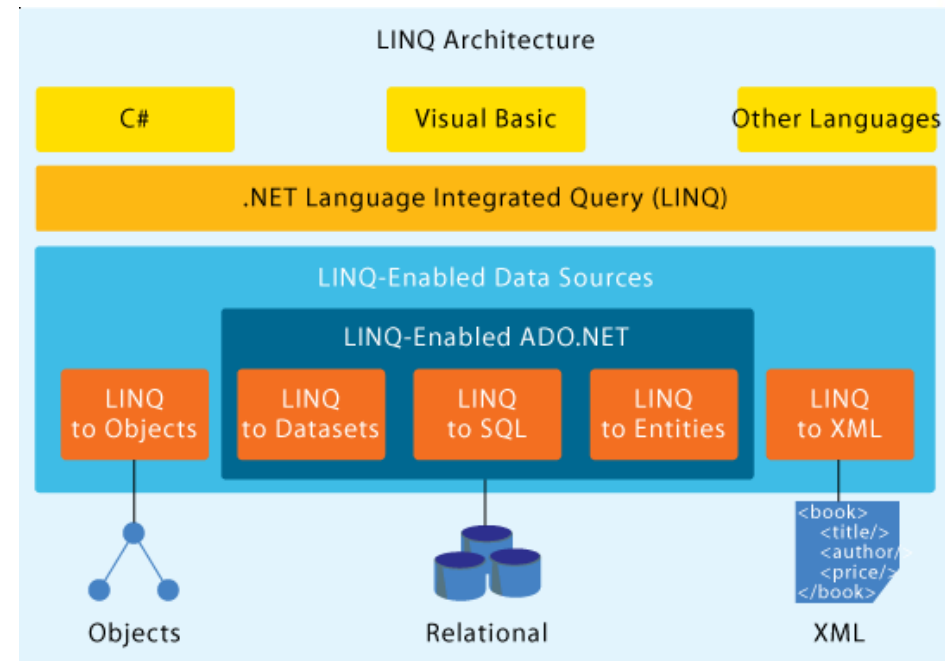
# LINQ

- *Language Integrated Query*
- Autorem tej technologii jest Anders Hejlsberg, który:
  - jako pierwszy stworzył IDE (Borland Turbo Pascal).
  - jest również autorem TypeScript.
- Do istniejącego języka programowania (m. in. MS C#) dodano składnię i funkcjonalność umożliwiającą pisanie zapytań podobnych do SQL.
- Podobne rozwiązanie dla Java 8+:  
[strumienie funkcyjne](#) w `java.util.stream`



# LINQ (2)

- Dzięki temu problem niezgodności jest znacząco zredukowany.
- Dodatkowe korzyści:
  - Wykorzystanie metadanych czasu wykonania,
  - Kontrola typologiczna w czasie kompilacji,
  - Wykorzystanie IntelliSense®,
- Różne warianty:
  - LINQ to Objects,
  - LINQ to XML,
  - LINQ to ADO.NET,
  - LINQ to JSON (Json.NET),
  - ...
- Rewolucja?



# LINQ (3)

- Przykłady

```
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)

- Przykłady – c. d.

```
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);
}
```



# Hibernate

- Kiedyś twórcy Hibernate opisywali go jako:
  - *Hibernate is a powerful, high performance object/relational persistence and query service*
  - Wydajna usługa zapewniająca trwałość danych oraz umożliwiającą zadawanie zapytań.
- Istnieją wersje dla Java oraz MS .NET.
- Projekt rozwijany jako open source od roku 2001:
  - 76 000 linii kodu podstawowego,
  - 36 000 linii kodu unit test.
- <http://www.hibernate.org/>

# Hibernate (2)

- Teraz piszą o nim: *More than an ORM, discover the Hibernate galaxy.*
- Galaktyka Hibernate, m.in.:
  - **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.
- Niestety nie likwiduje całkowicie problemu niezgodności impedancji.

# Hibernate - wydajność

- Twórcy twierdzą, że biblioteka jest wydajna i wykorzystuje różne techniki optymalizacji:
  - Cache'owanie obiektów,
  - Cache'owanie wyników zapytań,
  - Polecenia SQL wykonywane dopiero wtedy gdy są naprawdę potrzebne,
  - Brak uaktualniania niezmodyfikowanych obiektów,
  - Efektywne zarządzanie kolekcjami,
  - Łączenie wielu zmian w jedno UPDATE,
  - Leniwa inicjalizacja obiektów i kolekcji.

# Hibernate - środowisko testowe

- Bazujemy na oficjalnym tutorialu, ale stworzymy własne klasy biznesowe.
- Hibernate z Java domyślnie działa w oparciu o bazę danych poprzez JDBC.
- Jako bazę danych możemy zastosować:
  - Standardowe rozwiązania, np. [MariaDB](#) , [MySQL](#).
  - Lekkie systemy napisane w Java, np. [H2](#), [HSQL](#) lub [Apache Derby](#).
    - Zwykle wystarczy wykorzystać niewielki plik *jar* zawierający wszystkie niezbędne komponenty.
    - Do celów testowych wygodne jest zastosowanie trybu *in-memory*.

# Hibernate - środowisko testowe (2)

- Wykorzystanie bazy danych H2
  - [Pobierz archiwum](#).
  - Rozpakuj je do dowolnego katalogu.
  - Skopiuj plik *bin/\*.jar* do katalogu *lib* swojego projektu Java (katalog ten powinien być skonfigurowany w IDE jako źródło dodatkowych bibliotek).
  - Ewentualnie można skonfigurować IDE aby pobrało odpowiednią bibliotekę z repozytorium Maven, np.  
*com.h2database:h2:1.4.199*

# Hibernate - środowisko testowe (3)

- Wykorzystanie bazy danych H2 – kont.
  - Uruchomienie bazy danych:
    - klasycznie w trybie serwera – plik *bin/h2.bat* lub *bin/h2.sh* (dostępna prosta konsola konfiguracyjna przez przeglądarkę).
    - uproszczone – zrobi to automatycznie Hibernate po odpowiednim skonfigurowaniu (patrz dalej). Wymaga to dodania, wcześniej wspomnianego pliku *jar*, do bibliotek uruchomieniowych projektu w IDE.

# Plik konfiguracyjny *hibernate.cfg.xml*

- Lokalizacja: główny katalog z plikami źródłowymi

```
<!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>
```



# Plik konfiguracyjny *hibernate.cfg.xml* (2)

- W zależności od rodzaju/trybu działania bazy danych H2, np.:
  - tryb *in-memory*. Wygodny do testowania (bez zachowania danych na stałe), system zarządzania BD jest uruchamiany automatycznie.

```
<!-- 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>
```

- tryb plikowy. Wymaga uprzedniego uruchomienia serwera H2. Może również potrzebować podania dodatkowych ustawień, np. hasła.

```
<!-- 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>
```

# Podstawy biblioteki Hibernate

- Stworzymy prostą aplikację umożliwiającą:
  - przechowywanie informacji o filmach,
  - łączenie filmów z aktorami.
- Każda klasa, która chce wykorzystywać pełnię możliwości biblioteki, musi posiadać specyficzny atrybut służący do identyfikacji wystąpień.
  - `private long id;`
  - Zmianą jego wartości zajmuje się Hibernate.
- Zalecane wykorzystywanie konwencji JavaBean
  - `set...()`,
  - `get...()`.

# Mapowanie danych

- Niestety, Hibernate nie działa w pełni automatycznie (podobnie jak inne ORM-y) i trzeba doprecyzować sposób mapowania konstrukcji z BD (relacyjnych) na obiektowe (Java).
- Sposoby mapowania:
  - natywne adnotacje Hibernate,
  - adnotacje JPA,
  - plik mapujący XML.
- Który wybrać?
- Zalety i wady?

# Mapowanie klasy - przykład

- Standardowa klasa Java przechowująca informacje o filmie.
- Kilka atrybutów.
- Settery i gettery.
- Brak konieczności dziedziczenia ze specjalnej nadklasy.

```
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;
    }
}
```

# Mapowanie klasy – przykład (2)

- Korzystamy z adnotacji **JPA**.
- Klasę oznaczamy za pomocą adnotacji `@javax.persistence.Entity` (opcjonalny parametr `Name`).
- Wymagany publiczny lub chroniony bezparametrowy konstruktor.
- Dodatkowe informacje korzystając z adnotacji: `@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;
    }

    // [...]
}
```

# Mapowanie klasy – identyfikator

- Dodaliśmy atrybut pełniący rolę identyfikatora.
- Utworzyliśmy *setter* oraz *getter* dla niego.
- *Getter* został oznaczony odpowiednimi adnotacjami.
  - **@Id**
  - **@GeneratedValue**
  - **@GenericGenerator**
- Można też oznaczyć atrybut zamiast *getter'a*.

```
@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;
    }

    // [...]
}
```

# Mapowanie klasy – atrybuty proste

- Adnotacja: `@javax.persistence.Basic`
  - ma zastosowanie do:
    - typów prostych,
    - kilku innych, m.in. `String` oraz związanych z datą.
  - można ją pominąć bo jest to domyślne zachowanie Hibernate.
  - opcjonalne parametry:
    - `optional`. Określa dopuszczalność `null`'i (`True`),
    - `fetch`. Sposób pobierania wartości (`Eager`).
- Adnotacja `@javax.persistence.Column`. Większe możliwości dostosowywania, np. nazwa kolumny w BD.
- Adnotacja `@javax.persistence.Type`. Doprecyzowanie typu w BD.
- Możliwość utrwalania własnych typów podstawowych.

```
@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();
    }
}
```



# Mapowanie klasy – atrybuty pochodne

- Adnotacja:  
`@javax.persistence.Transient`
  - oznacza ignorowanie przez Hibernate określonego atrybutu lub gettera i skojarzonego atrybutu.
  - umożliwia implementację metod (głównie *getter'y*) wykorzystywanych przez atrybuty pochodne, np.
    - `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());
    }
}
```

# Mapowanie klasy – enum

- Adnotacja: `@javax.persistence.Enumerated`.  
Mapowanie enum'ów. Dodatkowe parametry:
  - `EnumType.ORDINAL`. Wykorzystuje wartość liczbową,
  - `EnumType.STRING`. Używa nazwy enum'a.

```
@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();
    }
}
```

# Mapowanie klasy – BLOB/LOB-y

- Hibernate umożliwia również mapowanie BLOB-ów/LOB-ów (*database Large Objects*).
  - Uwaga na zużycie zasobów (pamięci).
  - Techniki optymalizacyjne, np. strumieniowanie.
  - Mapowanie na różne typy języka Java, np. `String` lub dostęp za pomocą strumieni.
  - Więcej informacji o [LOB-ach w dokumentacji Hibernate](#).

# Tożsamość obiektów w Hibernate

- Metody `equals()` oraz `hashCode()`.
- Hibernate gwarantuje, że ten sam obiekt (klucz główny) pobrany z bazy danych, w **ramach jednej sesji**, będzie miał tę samą instancję w środowisku Java.
- Czasami pomocna może być własna implementacja ww. metod z adnotacją atrybutu: `@NaturalId`.
- Więcej w [dokumentacji Hibernate](#).

# Praca z Hibernate - *Session*

- Tworzymy *registry*.
- Tworzymy *session factory*.
- Otwieramy sesję.
- Rozpoczynamy transakcję.
- **Wykonujemy operacje.**
- Potwierdzamy transakcję oraz zamykamy sesję.

```
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 – tworzenie obiektów

- Dodajemy informację o filmach.
  - Na tym etapie klucze główne nie mają wygenerowanej wartości. Są one nadawane po potwierdzeniu transakcji.

```
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 – pobieranie danych

- Korzystamy z języka zapytań podobnego do SQL.
- Jak widać pracujemy na kompletnych obiektach (klasa `Movie`), a nie (jak w przypadku „czystego” JDBC) na wartościach (np. `String`) z bazy danych.
- Pobrane obiekty zawierają prawidłowe wartości kluczy głównych.

```
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 – plik log (fragment)

```
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 – schemat danych

- Oczywiście Hibernate automatycznie tworzy odpowiedni schemat/model danych relacyjnych w BD.
- Po „wygenerowaniu” danych (stabilna wersja modelu danych), należy zakomentować/usunąć fragment pliku *hibernate.cfg.xml* (w przeciwnym wypadku istniejące dane będą usuwane po starcie programu).

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

# Asocjacje w Hibernate

- Asocjacje są (prawie) automatycznie mapowane na relacje w bazie danych.
- Elementy, które trzeba uwzględnić to:
  - Kierunek,
  - Liczności,
  - Zachowanie się kolekcji służącej do implementacji (po stronie Javy).

# Dodanie asocjacji skierowanej

- Stworzymy klasę Aktor (*Actor*), która będzie powiązana z filmem (*Movie*).
- Nowy wpis do pliku konfiguracyjnego *hibernate.cfg.xml*.

```
<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.
}
```

# Dodanie asocjacji skierowanej (2)

- Do klasy *Movie* dodamy informację o aktorach w nim grających (*actors*).
- Wykorzystaliśmy pojemnik typu `List` (obsługiwane są też inne).
- Adnotacja **@OneToMany**.

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

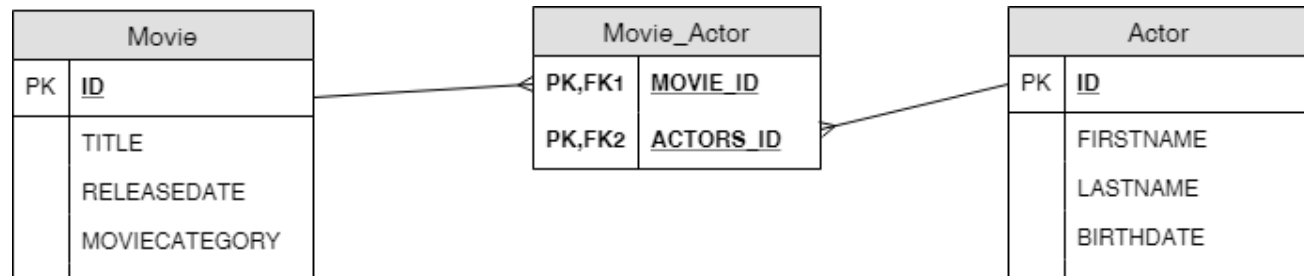
    // [...]

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

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

# Dodanie asocjacji skierowanej (3)

- W efekcie otrzymaliśmy:
  - poniższy schemat relacyjny (tabela pośrednicząca została wygenerowana automatycznie, chociaż nie jest potrzebna dla 1-\*).



- oraz połączenie asocjacją (referencjami języka Java) od klasy *Movie* do *Actor* (ale nie w drugą stronę).



# Użycie asocjacji skierowanej

- Powiązanie dodajemy poprzez modyfikację pojemnika języka Java,
- Hibernate to automatycznie wykrywa i uaktualnia bazę danych,
- Analogiczna „automatyka” istnieje dla atrybutów.

```
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();
```



# Użycie asocjacji skierowanej (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 (?, ?)

# Użycie asocjacji skierowanej (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)***

# Użycie asocjacji skierowanej (4)

## Movie

<u>ID</u>	<u>MOVIECATEGORY</u>	<u>RELEASEDATE</u>	<u>TITLE</u>
1	2	1984-10-26	Terminator 1
2	2	2003-08-08	Terminator 3

## MOVIE\_ACTOR

<u>MOVIE_ID</u>	<u>ACTORS_ID</u>
2	1
2	2
2	3

## Actor

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

# Dodanie asocjacji dwukierunkowej

- Schemat relacyjny nie ulega zmianie.
- Adnotacja `@ManyToOne`.
- Samodzielnie należy zadbać o spójność obu kierunków (dedykowana logika w metodzie tworzącej połączenie).
- Trzeba skorzystać z parametru `inverse` lub `mappedBy`.

```
@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);
    }
}
```

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

    // [...]

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

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

# Dodanie asocjacji dwukierunkowej (2)

- Podobnie mapujemy inne licznosci korzystając z adnotacji:
  - **@ManyToMany,**
  - **@OneToOne.**
- Uwaga na zdefiniowanie „właściciela” powiązania. Ważne w przypadku usuwania obiektów.
- Specjalne przypadki asocjacji:
  - **@NotFound.** Gdy nie znaleziono powiązanego klucza głównego.
  - **@Any.**
  - **@JoinFormula, @JoinColumnOrFormula.**

# Atrybuty powtarzalne

- W Hibernate nazywane są kolekcją wartości (*Collection of values*).
- Różnica w stosunku do asocjacji polega na tym, że wartości nie są współdzielone (a obiekty oczywiście tak).
- Muszą być zadeklarowane za pomocą jakiegoś interfejsu, a nie konkretnej implementacji.
- Zachowanie takiego atrybutu powtarzalnego zależy od typu interfejsu (np. `List`, `Set`).

# Atrybuty powtarzalne (2)

- Dla klasy aktor (`Actor`) dodamy listę adresów internetowych.
- Anotacja `@javax.persistence.ElementCollection`
  - Oznacza, że kolekcja nie zawiera odniesień do innych wystąpień, ale listę elementów, np. typu `String`.

```
@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;
    }
}
```

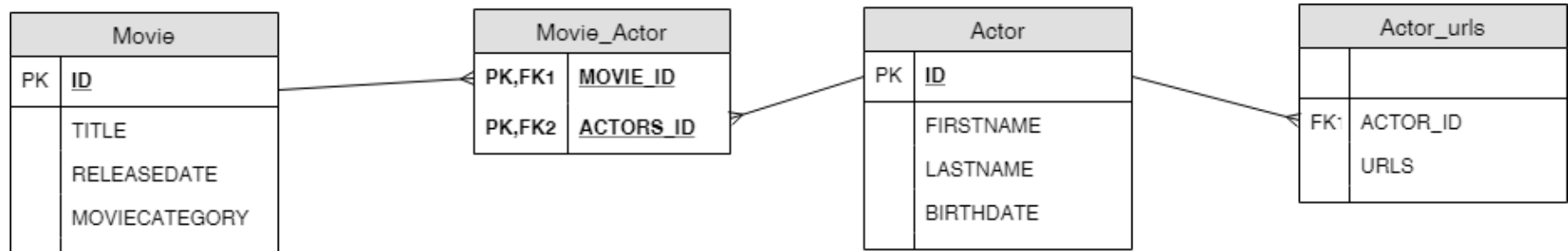
```
actor1.setUrls(List.of("http://www.schwarzenegger.com/",
                      "https://pl.pinterest.com/schwarzenegger/",
                      "https://www.facebook.com/arnold"));
```

**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>]



# Atrybuty powtarzalne (3)

- Uaktualniony schemat relacyjny



# Hibernate - dziedziczenie

- Mapowanie nadklasy (*MappedSuperclass*).
- Pojedyncza tabela (*Single table, Table Per Hierarchy - TPH*).
- Złączona tabela (*Joined table, table-per-subclass/type - TPT*).
- Tabela dla podklasy (*Table per class, table-per-concrete-class - TPC*).
- (*Patrz też poprzedni wykład*)

# Dziedziczenie - mapowanie nadklasy

- Odzwierciedlenie tylko w modelu, ale nie w BD. Brak możliwości odnoszenia się do nadklasy.
- W BD powstaną tylko dwie tabele (powtarzające zawartość nadklasy).

```
@MappedSuperclass
public static 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 static class DebitAccount extends Account {
    private BigDecimal overdraftFee;
    //Getters and setters are omitted for brevity
}
@Entity(name = "CreditAccount")
public static class CreditAccount extends Account {
    private BigDecimal creditLimit;
    //Getters and setters are omitted for brevity
}
```

*Źródło: dokumentacja Hibernate*

# Dziedziczenie - pojedyncza tabela

- Jedna tabela zawierająca elementy z nadklasy oraz wszystkich podklas.
- Specjalna kolumna – dyskryminator.

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

*Źródło: dokumentacja Hibernate*

# Dziedziczenie - złączona tabela

- Każda klasa ma swoją tabelę. Połączenia za pomocą relacji (klucz główny - obcy).

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

*Źródło: dokumentacja Hibernate*

# Dziedziczenie - tabela dla podklasy

- Tabele są generowane dla każdej podklasy i umieszczana jest w nich również zawartość nadklasy.

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

*Źródło: dokumentacja Hibernate*

# Zapytania w Hibernate

- Zapytania definiowane za pomocą *Criteria*
  - mocna kontrola typologiczna,
  - dość skomplikowana budowa.

```
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();
```



# Zapytania w Hibernate (2)

- Przykładowe zapytania w [Hibernate Query Language \(HQL\)](#) – podobny do 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());
```

# Podsumowanie

- Niezgodność impedancji czyli trudności w pogodzeniu świata relacyjnego oraz obiektowego, jest bardzo poważnym, praktycznym problemem.
- Można wyróżnić dwa generalne podejścia do jego rozwiązania:
  - Modyfikacje języka programowania, tak aby posiadał jak najwięcej zalet baz danych,
  - Stworzenie dodatkowych bibliotek ułatwiających pracę z danymi.

# Podsumowanie (2)

- Przykładem pierwszego podejścia do problemu jest Microsoft C# razem z LINQ.
  - Język zapytań (funkcjonalnie podobny do SQL) stał się częścią języka programowania.
  - Niezgodność impedancji jest znacząco zredukowana, czyli nie musimy mapować konstrukcji obiektowych na relacyjne i odwrotnie.
  - Dzięki temu mamy m. in. kontrolę typologiczną.

# Podsumowanie (3)

- Drugie podejście jest reprezentowane przez Hibernate
  - Biblioteka rzeczywiście ułatwia pracę z bazami danych,
  - Niestety czasami występuje konieczność operowania identyfikatorami (kluczami relacyjnymi).
- Wydaje się, że zdecydowanie lepszym podejściem jest dodanie cech baz danych do języka programowania, tak jak w Microsoft LINQ.

# Pliki źródłowe

---

- Pobierz pliki źródłowe do wszystkich wykładów MAS



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