laboratory Gerstner

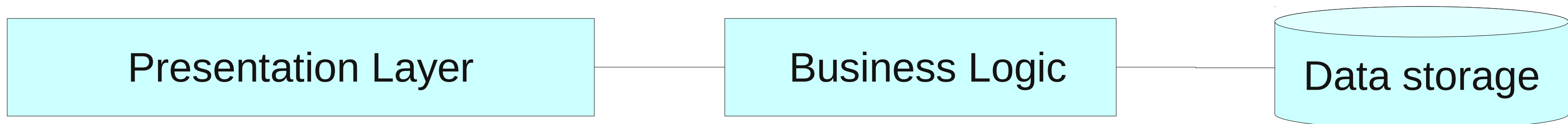
ORM and JPA 2.0

Petr Křemen

petr.kremen@fel.cvut.cz

What is Object-relational mapping ?

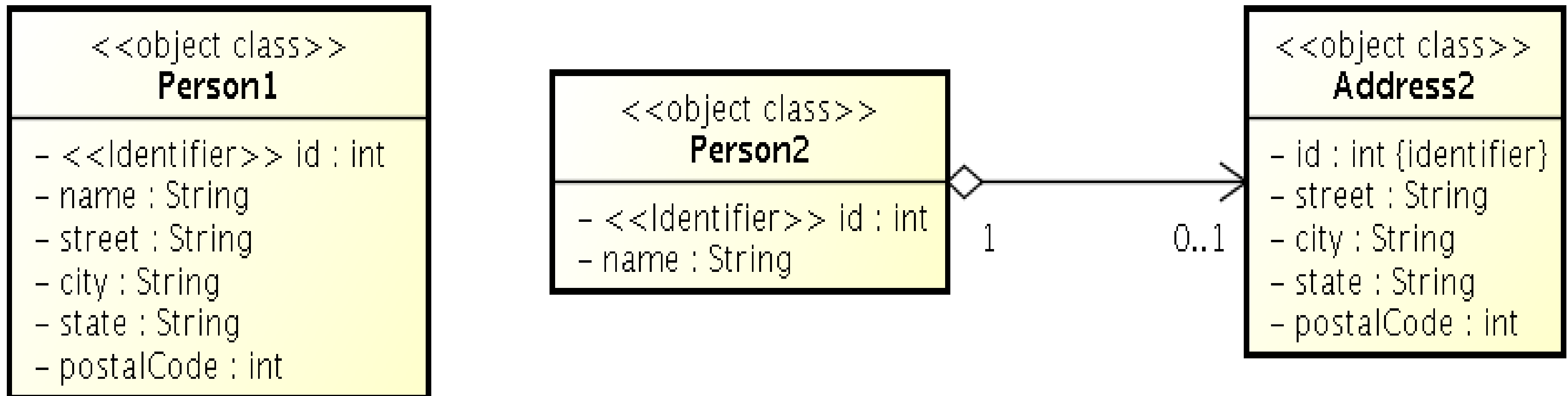
- a typical information system architecture:



- How to avoid data format transformations when interchanging data from the (OO-based) presentation layer to the data storage (RDBMS) and back ?
- How to ensure persistence in the (OO-based) business logic ?

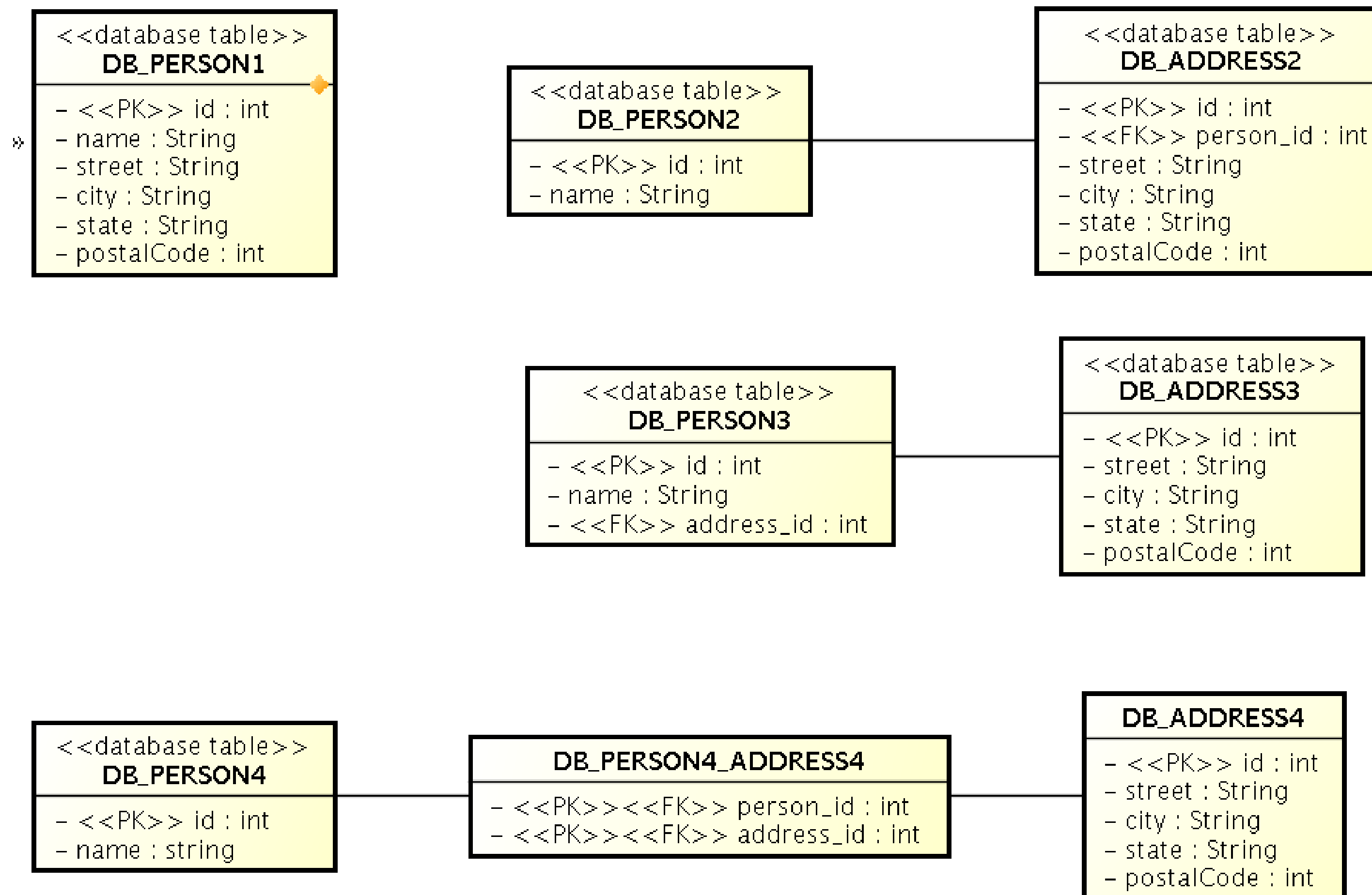
Example – object model

- When would You stick to one of these options ?



Example – database

- … and how to model it in SQL ?



Object-relational mapping

- Mapping between the database (declarative) schema and the data structures in the object-oriented language.
- Let's take a look at JPA 2.0

JPA 2.0

- Java Persistence API 2.0 (JSR-317)
- Although part of Java EE 6 specifications, JPA 2.0 can be used both in EE and SE applications.
- Main topics covered:
  - Basic scenarios
  - Controller logic – `EntityManager` interface
  - ORM strategies
  - JPQL + Criteria API

JPA 2.0 – Entity Example

- Minimal example (configuration by exception):

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private Integer id;
    private String name;
    // setters + getters
}
```

JPA 2.0 - Basics

- Let's have a set of „suitably annotated“ POJOs, called ***entities***, describing your domain model.
- A set of entities is logically grouped into a ***persistence unit***.
- JPA 2.0 providers :
  - generate persistence unit from existing database,
  - generate database schema from existing persistence unit.
- What is the benefit of the keeping Your domain model in the persistence unit entities (OO) instead of the database schema (SQL)

JPA 2.0 – Persistence Context

- In runtime, the application accesses the object counterpart (represented by entity instances) of the database data. These (*managed*) entities comprise a ***persistence context (PC)***.
  - PC is synchronized with the database on demand (refresh, flush) or at transaction commit.
  - PC is accessed by an `EntityManager` instance and can be shared by several `EntityManager` instances.

JPA 2.0 – EntityManager

- **`EntityManager` (EM)** instance is in fact a generic DAO, while entities can be understand as DPO (managed) or DTO (detached).
- Selected operations on EM (CRUD) :
  - ***C***reate : em.persist(Object o)
  - ***R***ead : em.find(Object id), em.refresh(Object o)
  - ***U***pdate : em.merge(Object o)
  - ***D***elete : em.remove(Object o)
  - native/JPQL queries: createNativeQuery, createQuery, etc.
  - Resource-local transactions: getTransaction().[begin(),commit(),rollback()]

ORM - Basics

- Simple View
  - Object classes = entities = SQL tables
  - Object properties (fields/accessor methods) = entity properties = SQL columns
- The ORM is realized by means of Java annotations/XML.
- Physical Schema annotations
  - @Table, @Column, @JoinColumn, @JoinTable, etc.
- Logical Schema annotations
  - @Entity, @OneToMany, @ManyToMany, etc.
- Each property can be fetched lazily/eagerly.

ORM – Basic data types

- Primitive Java types: String → varchar/text, Integer → int, Date → TimeStamp/Time/Date, etc.
- Wrapper classes, basic type arrays, Strings, temporal types
- @Column – physical schema properties of the particular column (insertable, updatable, precise data type, defaults, etc.)
- @Lob – large objects
- Default EAGER fetching (except Lobs)

```
@Column(name=“id“)
private String getName();
```

ORM – Enums, dates

- `@Enumerated(value=EnumType.String)`

  `private EnumPersonType type;`

  - Stored either in text column, or in int column

- `@Temporal(TemporalType.Date)`

  `private java.util.Date datum;`

  - Stored in respective column type according to the TemporalType.

ORM – Identifiers

- Single-attribute: @Id,
- Multiple-attribute – an identifier class must exist
  - Id. class: @IdClass, entity ids: @Id
  - Id. class: @Embeddable, entity id: @EmbeddedId
- How to write `hashCode`, `equals` for entities ?
- ```
  @Id
  @GeneratedValue(strategy=GenerationType.SEQUENCE)
  private int id;
  ```
  ```

ORM – Relationships

- Unidirectional vs. Bidirectional
- `@OneToMany`
  - Forgotten `mappedBy`
- `@ManyToOne`
- `@ManyToMany`
  - Two `ManyToMany` relationships from two different entities
- `@OneToOne`
- `@JoinColumn, @JoinTable` – in the owning entity (holding the foreign key)
- Cascading

ORM – advanced topics

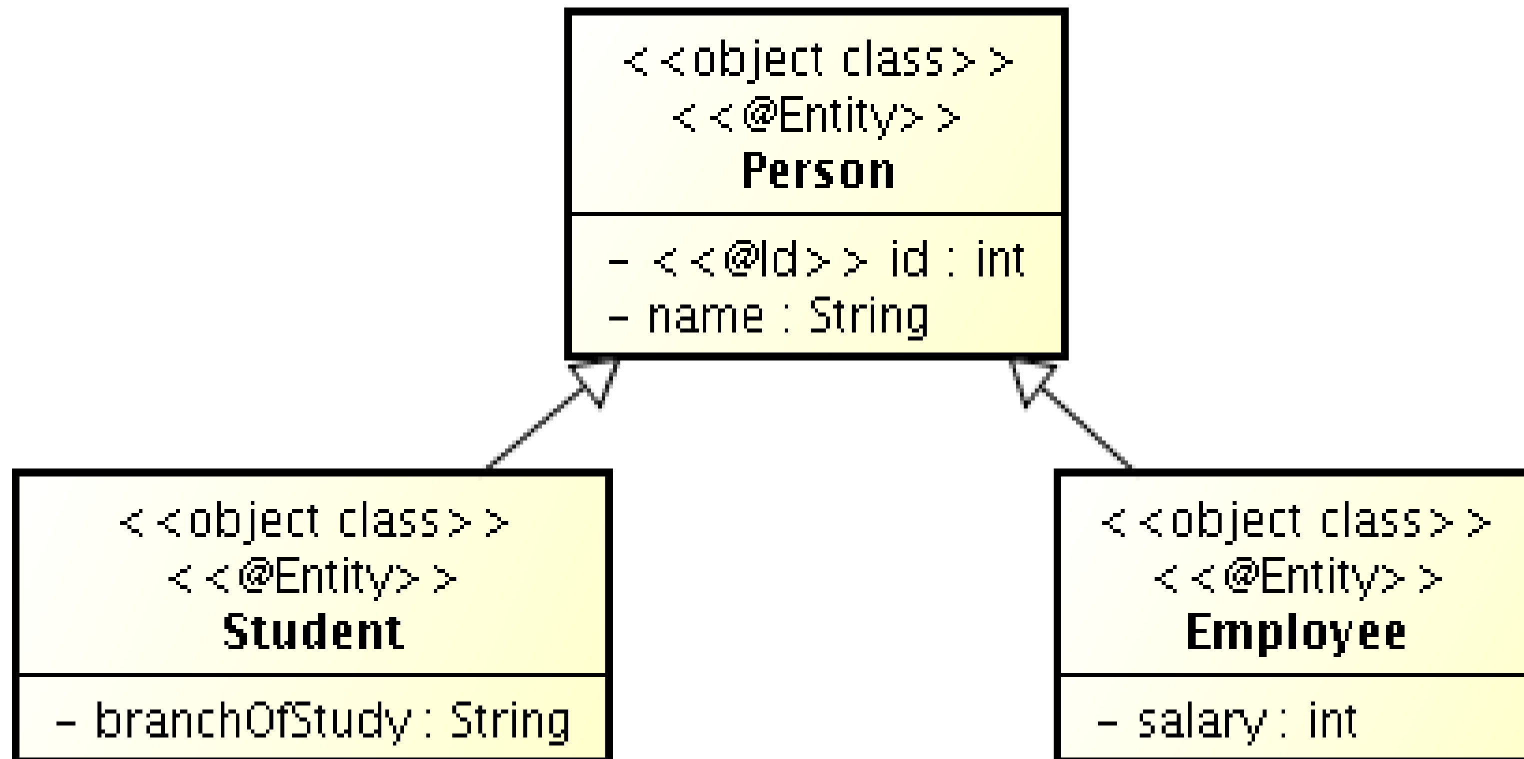
- Embeddables, embedded objects
- Collections of basic data types and embeddables
  - Lists, Sets,
  - Maps

ORM – RelationShips

- Unidirectional vs. Bidirectional
- `@OneToMany`
  - Forgotten `mappedBy`
- `@ManyToOne`
- `@ManyToMany`
  - Two `ManyToMany` relationships from two different entities
- `@OneToOne`
- `@JoinColumn, @JoinTable` – in the owning entity (holding the foreign key)
- Cascading

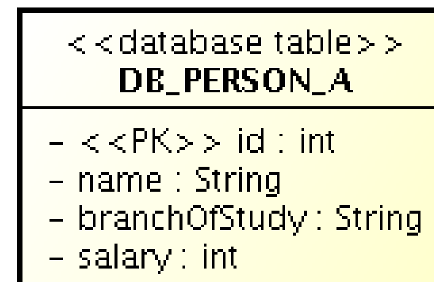
Inheritance

- How to map inheritance into RDBMS ?

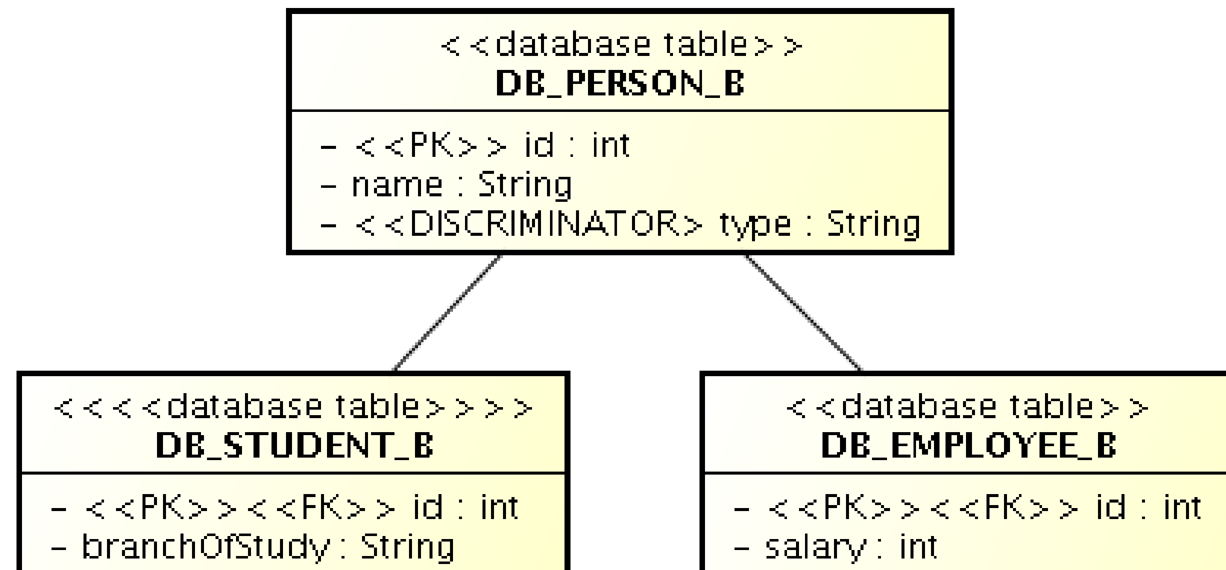


Strategies for inheritance mapping

- Single table

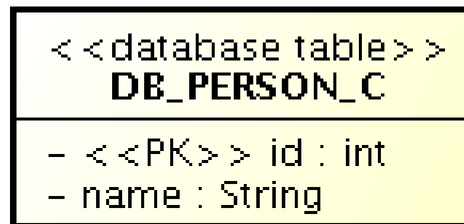


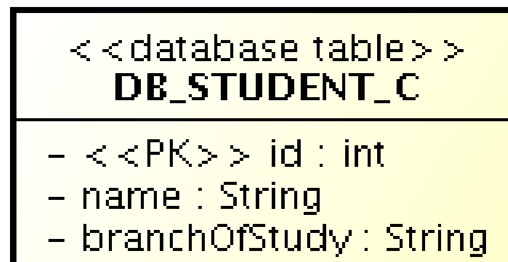
- Joined

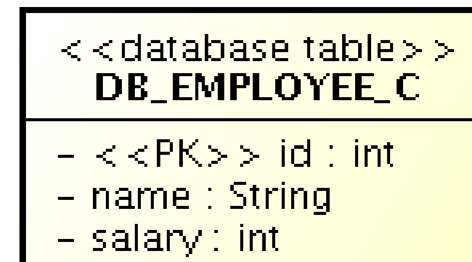


Strategies for inheritance mapping

- Table-per-concrete-class

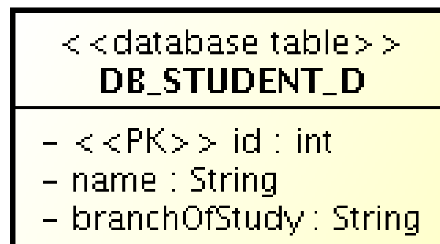


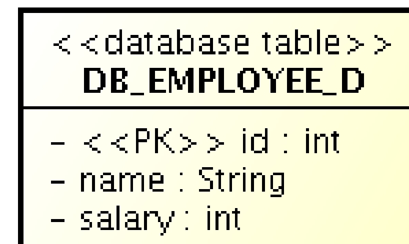




Strategies for inheritance mapping

- If Person is not an `@Entity`, but a `@MappedSuperClass`





- If `Person` is not an `@Entity`, neither `@MappedSuperClass`, the deploy fails as the `@Id` is in the `Person` (non-entity) class.

Criteria API, Metamodel API

- Criteria API – for building queries represented as Java Objects (not strings)
- Metamodel API – to represent metamodel of the persistence unit.