

# ORM and JPA 2.0

Zdeněk Kouba, Petr Křemen

# Compound primary keys

## Id Class

EMPLOYEE	
PK	<u>COUNTRY</u>
PK	<u>EMP_ID</u>
	NAME
	SALARY

No setters. Once created, can not be changed.

```
@Entity
@IdClass(EmployeeId.class)
public class Employee {
    @Id private String country;
    @Id
    @Column(name="EMP_ID")
    private int id;
    private String name;
    private long salary;
    // ...
}
```

```
public class EmployeeId
    implements Serializable {
    private String country;
    private int id;

    public EmployeeId() {}
    public EmployeeId(String country,
        int id) {
        this.country = country;
        this.id = id;
    }

    public String getCountry() {...};
    public int getId() {...}

    public boolean equals(Object o) {...}

    public int hashCode() {
        Return country.hashCode() + id;
    }
}
```

```
EmployeeId id = new EmployeeId(country, id);
Employee emp = em.find(Employee.class, id);
```

# Compound primary keys

## Embedded Id Class

EMPLOYEE	
PK	<u>COUNTRY</u>
PK	<u>EMP_ID</u>
	NAME
	SALARY

```
@Embeddable
public class EmployeeId
    private String country;
    @Column(name="EMP_ID")
    private int id;

    public EmployeeId() {}
    public EmployeeId(String country,
        int id) {
        this.country = country;
        this.id = id;
    }
    // ...
}
```

```
@Entity
public class Employee {
    @EmbeddedId private EmployeeId id;
    private String name;
    private long salary;
    // ...
    public String getCountry() {return id.getCountry();}
    public int getId() {return id.getId();}
    // ...
}
```

# Compound primary keys

## Embedded Id Class

EMPLOYEE	
PK	<u>COUNTRY</u>
PK	<u>EMP_ID</u>
	NAME
	SALARY

```
@Embeddable
public class EmployeeId
    private String country;
    @Column(name="EMP_ID")
    private int id;

    public EmployeeId() {}
    public EmployeeId(String country,
                       int id) {
        this.country = country;
        this.id = id;
    }
    // ...
}
```

Referencing an embedded IdClass in a query:

```
em.createQuery("SELECT e FROM Employee e " +
              "WHERE e.id.country = ?1 AND e.id.id = @2")
    .setParameter(1, country)
    .setParameter(2, id)
    .getSingleResult();
```

# Optionality

```
@Entity
public class Employee
    // ...

    @ManyToOne(optional=false)
    @JoinColumn(name="DEPT_ID", insertable=false, updatable=false)
    private Department department;
    // ...
}
```

Optionality (parciality) can be used only for **@ManyToOne** and **@OneToOne** relations making the „1“ side of the cardinality „0...1.“

# Compound Join Columns

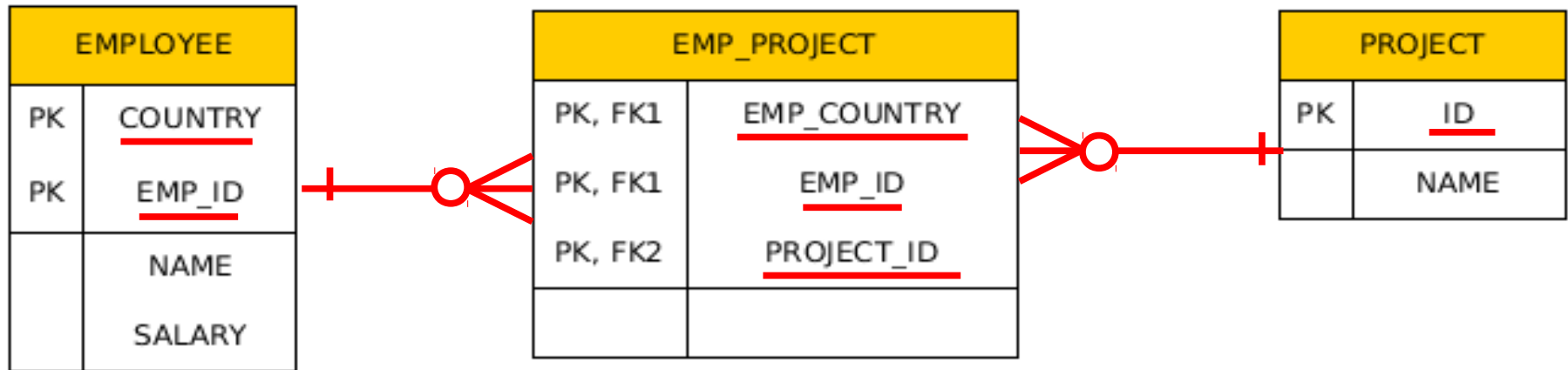
EMPLOYEE	
PK	<u>COUNTRY</u>
PK	<u>EMP_ID</u>
	NAME
	SALARY
FK1	MGR_COUNTRY
FK1	MGR_ID

```
@Entity
@IdClass(EmployeeId.class)
public class Employee {
    @Id private String country;
    @Id
    @Column(name="EMP_ID")
    private int id;

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="MGR_COUNTRY",
                    referencedColumnName="COUNTRY"),
        @JoinColumn(name="MGR_ID",
                    referencedColumnName="EMP_ID")
    })
    private Employee manager;

    @OneToMany(mappedBy="manager")
    private Collection<Employee> directs;
    // ...
}
```

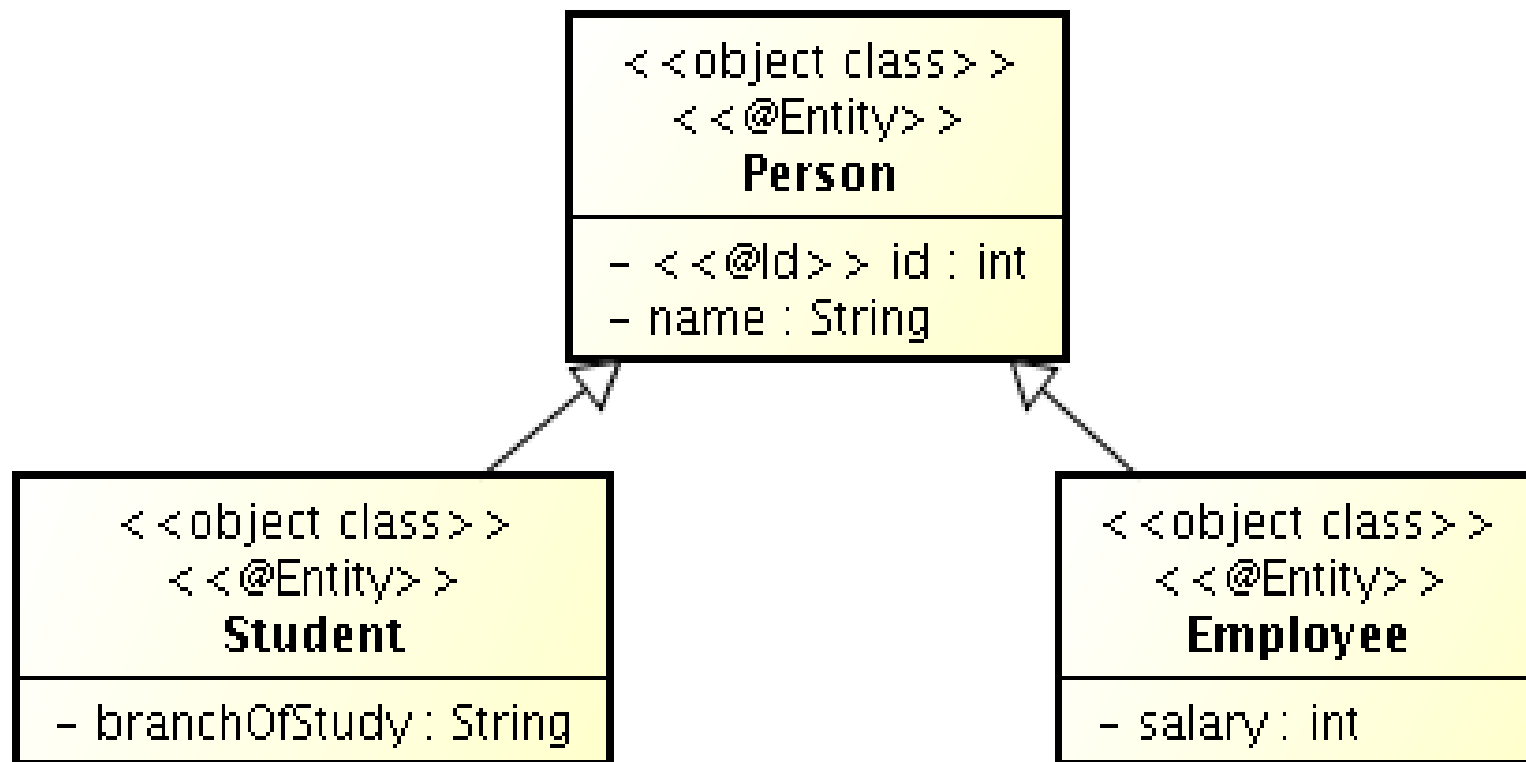
# Compound Join Columns



```
@Entity
@IdClass(EmployeeId.class)
public class Employee
    @Id private String country;
    @Id
    @Column(name="EMP_ID")
    private int id;
    @ManyToMany
    @JoinTable(
        name="EMP_PROJECT",
        joinColumns={
            @JoinColumn(name="EMP_COUNTRY", referencedColumnName="COUNTRY"),
            @JoinColumn(name="EMP_ID", referencedColumnName="EMP_ID")},
        inverseJoinColumns=@JoinColumn(name="PROJECT_ID"))
    private Collection<Project> projects;
}
```

# Inheritance

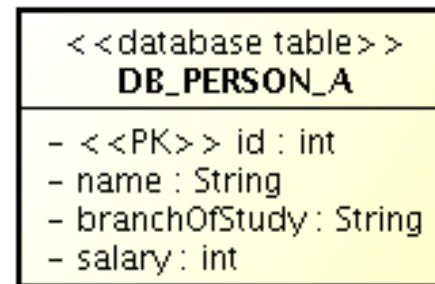
- How to map inheritance into RDBMS ?



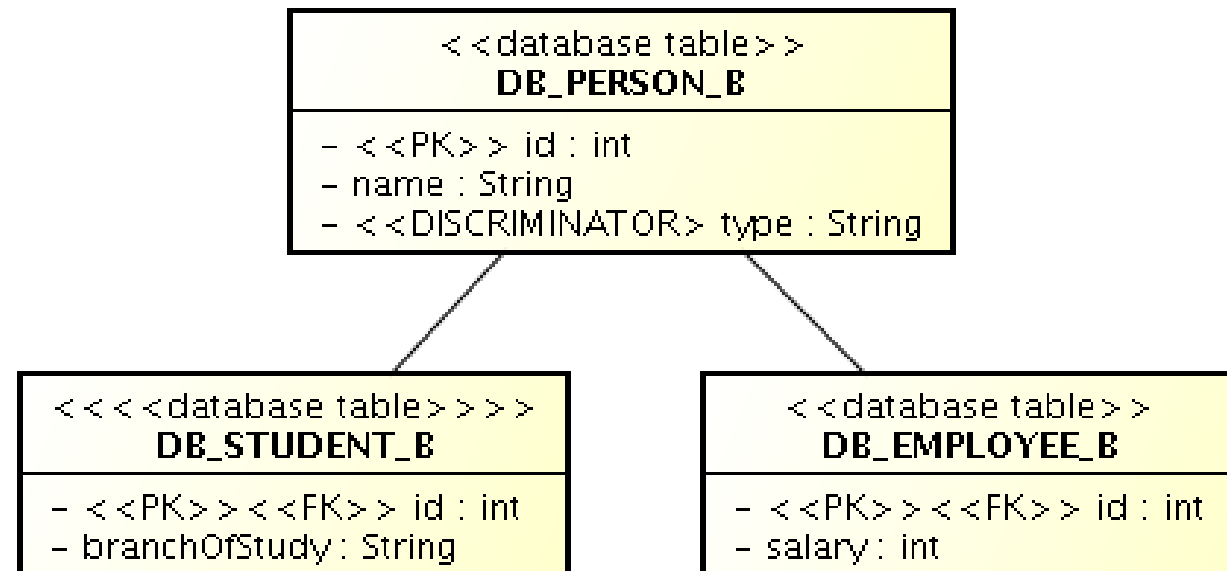


# Strategies for inheritance mapping

- Single table

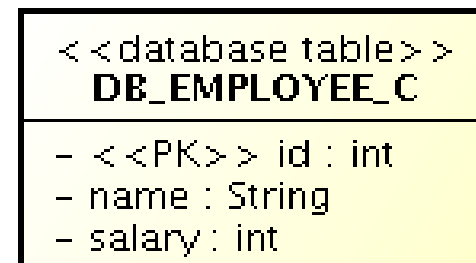
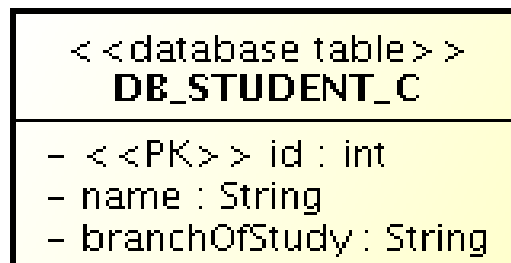
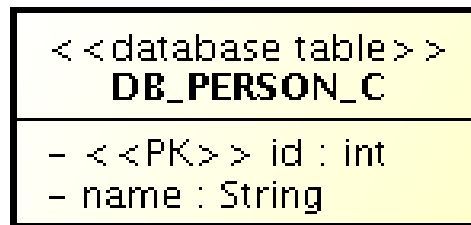


- Joined



# Strategies for inheritance mapping

- Table-per-concrete-class



# Inheritance mapping

## single-table strategy

```
@Entity
@Table(name="DB_PERSON_A")
@Inheritance //same as @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminationColumn(name="EMP_TYPE")
public abstract class Person { ...}

@Entity
@DiscriminatorValue("Emp")
Public class Employee extends Person {...}

@Entity
@DiscriminatorValue("Stud")
Public class Student extends Person {...}
```

# Inheritance mapping

## joined strategy

```
@Entity
@Table(name="DB_PERSON_B")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="EMP_TYPE",
                    discriminatorType=discriminatorType.INTEGER)
public abstract class Person { ...}

@Entity
@Table(name="DB_EMPLOYEE_B")
@DiscriminatorValue("1")
public class Employee extends Person {...}

@Entity
@Table(name="DB_STUDENT_B")
@DiscriminatorValue("2")
public class Student extends Person {...}
```

# Inheritance mapping

## table-per-concrete-class strategy

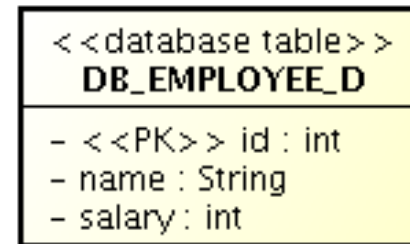
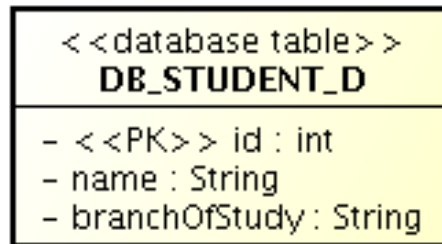
```
@Entity
@Table(name="DB_PERSON_C")
public abstract class Person { ...}

@Entity
@Table(name="DB_EMPLOYEE_C")
@AttributeOverride(name="name", column=@Column(name="FULLNAME"))
@DiscriminatorValue("1")
public class Employee extends Person {...}

@Entity
@Table(name="DB_STUDENT_C")
@DiscriminatorValue("2")
public class Student extends Person {...}
```

# 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.

# Queries

- JPQL (Java Persistence Query Language)
- Native queries (SQL)

# JPQL

JPQL very similar to SQL (especially in JPA 2.0)

```
SELECT p.number  
FROM Employee e JOIN e.phones p  
WHERE e.department.name = 'NA42' AND p.type = 'CELL'
```

Conditions are not defined on values of database columns,  
but on entities and their properties.

```
SELECT d, COUNT(e), MAX(e.salary), AVG(e.salary)  
FROM Department d JOIN d.employees e  
GROUP BY d  
HAVING COUNT(e) >= 5
```



# JPQL – query parameters

- positional

```
SELECT e
FROM Employee e
WHERE e.department = ?1 AND e.salary > ?2
```

- named

```
SELECT e
FROM Employee e
WHERE e.department = :dept AND salary > :base
```

# JPQL – defining a query dynamically

```
public class Query {
    EntityManager em;

    //...

    public long queryEmpSalary(String deptName, String empName)
    {
        String query = "SELECT e.salary FROM Employee e " +
            "WHERE e.department.name = '" + deptName +
            "' AND e.name = '" + empName + "'";
        return em.createQuery(query, Long.class)
            .getSingleResult();
    }
}
```

# JPQL – using parameters

```
static final String QUERY = "SELECT e.salary FROM Employee e " +  
    "WHERE e.department.name = :deptName " +  
    "AND e.name = :empName";  
  
public long queryEmpSalary(String deptName, String empName) {  
    return em.createQuery(QUERY, Long.class)  
        .setParameter("deptName", deptName)  
        .setParameter("empName", empName)  
        .getSingleResult();  
}
```

# JPQL – named queries

```
@NamedQuery(name="Employee.findByName",  
            query="SELECT e FROM Employee e " +  
                "WHERE e.name = :name")
```

```
public Employee findEmployeeByName(String name) {  
    return em.createNamedQuery("Employee.findByName",  
                               Employee.class)  
                .setParameter("name", name)  
                .getSingleResult();  
}
```

# JPQL – named queries

```
@NamedQuery(name="Employee.findByDept",  
            query="SELECT e FROM Employee e " +  
                "WHERE e.department = ?1")
```

```
public void printEmployeesForDepartment(String dept) {  
    List<Employee> result =  
        em.createNamedQuery("Employee.findByDept",  
                            Employee.class)  
            .setParameter(1, dept)  
            .getResultList();  
    int count = 0;  
    for (Employee e: result) {  
        System.out.println(++count + ":" + e.getName);  
    }  
}
```

# JPQL – pagination

```
private long pageSize      = 800;
private long currentPage = 0;

public List getCurrentResults() {
    return em.createNamedQuery("Employee.findByDept",
                               Employee.class)
               .setFirstResult(currentPage * pageSize)
               .setMaxResults(pageSize)
               .getResultList();
}

public void next() {
    currentPage++;
}
```

# JPQL – bulk updates

Modifications of entities not only by `em.persist()` or `em.remove()`;

```
em.createQuery("UPDATE Employee e SET e.manager = ?1 " +  
              "WHERE e.department = ?2")  
    .setParameter(1, manager)  
    .setParameter(2, dept)  
    .executeUpdate();
```

```
em.createQuery("DELETE FROM Project p " +  
              "WHERE p.employees IS EMPTY")  
    .executeUpdate();
```

If REMOVE cascade option is set for a relationship, cascading remove occurs.

Native SQL update and delete operations should not be applied to tables mapped by an entity (transaction, cascading).

# Native (SQL) queries

```
@NamedNativeQuery(  
    name="getStructureReportingTo",  
    query = "SELECT emp_id, name, salary, manager_id," +  
            "dept_id, address_id " +  
            "FROM emp",  
    resultClass = Employee.class  
)
```

Mapping is straightforward



# Native (SQL) queries

```
@NamedNativeQuery(  
    name="getEmployeeAddress",  
    query = "SELECT emp_id, name, salary, manager_id," +  
            "dept_id, address_id, id, street, city," +  
            "state, zip " +  
            "FROM emp JOIN address " +  
            "ON emp.address_id = address.id"  
)
```

Mapping less straightforward

```
@SqlResultSetMapping(  
    name="EmployeeWithAddress",  
    entities={@EntityResult(entityClass=Employee.class),  
              @EntityResult(entityClass=Address.class)}
```

# Native (SQL) queries

```
@SqlResultSetMapping(name="OrderResults",
    entities={
        @EntityResult(entityClass=Order.class,
            fields={
                @FieldResult(name="id", column="order_id"),
                @FieldResult(name="quantity",
                    column="order_quantity"),
                @FieldResult(name="item",
                    column="order_item")
            }
        )
    },
    columns={
        @ColumnResult(name="item_name")
    }
)
```

```
Query q = em.createNativeQuery(
    "SELECT o.id AS order_id, " +
        "o.quantity AS order_quantity, " +
        "o.item AS order_item, " +
        "i.name AS item_name, " +
    "FROM order o, item i " +
    "WHERE (order_quantity > 25) AND (order_item = i.id)",
    "OrderResults");
```

```
List<Object[]> results = q.getResultList();
```

```
results.stream().forEach((record) -> {
    Order order = (Order)record[0];
    String itemName = (String)record[1];
    /...
});
```