Berner
Fachhochschule

CAS Java Microservice Development

Java Persistence API - O/R-Mapping

Simon Martinelli, simon.martinelli@bfh.ch, v2025.10

Content

1. Objection-Relational-Mapping
2. Relationship Mapping
3. Lazy Loading

O/R-Mapping

Access Type - Field Access

```
@Entity
public class Employee {

  @Id
  private Integer id;

  public Integer getId() {
    return id;
  }

  public Integer setId() {
    this.id = id;
  }
}
```

Access Type - Property Access

```
@Entity
public class Employee {

  private Integer id;

  @Id
  public Integer getId() {
    return id;
  }

  public Integer setId() {
    this.id = id;
  }
}
```

Access Type Options

- Different access types possible per class
- Mixing of access types in an inheritance hierarchy

```
@Entity
@Access(FIELD)
public class Vehicle {

  @Transient
  double fuelEfficiency;

  @Access(PROPERTY)
  protected double getDbFuelEfficiency() {
    return convertToImperial(fuelEfficiency);
  }

}
```

Mapping

```java
@Entity
@Table(name = "employees")
public class Employee {

  @Id
  @Column(name = "employees_id")
  private Integer id;

  @Basic(optional = false)
  private String name;
}
```

Persistent Data Types

- All primitive types, `String` and all wrapper classes (e.g., `Integer`), `BigDecimal`, Temporal Types
- `byte[]`, `Byte[]`, `char[]`, `Character[]`
- Enumerations
- References to
    - any other entity classes
    - Collections of Entities and wrapper classes and String, which are declared as `Collection`, `List`, `Set` or `Map`

Java/SQL Type Mapping

- Defined implicitly by JDBC "Data Type Conversion Table" https://download.oracle.com/otn-pub/jcp/jdbc-4_1-mrel-spec/jdbc4.1-fr-spec.pdf?AuthParam=1572082201_b508cd35046b6f60a93969dd4f9cdcce
- Explicitly by `@Column` annotation, e.g.

```
@Column(columnDefinition = "VARCHAR(20)"
private String name;
```

- Product specific
  - JPA implementation
  - JDBC Driver

Large Objects

- Storing data in BLOB or CLOB

```
public class Employee {

    @Lob // BLOB
    private byte[] picture;

    @Lob // CLOB
    private char[] largeText;

}
```

Enumerations

- Enumerations can be persisted. Either as ordinal value (position) or as a string (name of the constant)

```
public enum Color { RED, BLUE, GREEN}
```

```
@Enumerated(EnumType.ORDINAL) // Default
private Color color;
```

```
@Enumerated(EnumType.STRING)
private Color color;
```

Be careful when changing Enums!

Temporal Types

Java 8 Date/Time API is recommended

- Without timezone
  - `java.time.LocalDate`
  - `java.time.LocalTime`
  - `java.time.LocalDateTime`
- With timezone
  - `java.time.OffsetTime`
  - `java.time.OffsetDateTime`

Temporal Types Legacy Support

- Supported Types
  - `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`,
  - `java.util.Date`
  - `java.util.Calendar`
- `java.sql` types do not require further definition but for `java.util` the JDBC type must be specified
  - `TemporalType.DATE`, `TemporalType.TIME`, `TemporalType.TIMESTAMP`

```
@Temporal(TemporalType.DATE)
private java.util.Date dateOfBirth;
```

Transient Attributes

- Attributes can be excluded from the persistence
- Either transiently using the keyword `transient`

```
transient private String translatedName;
```

- or if the attribute must be serializable using an annotation

```
@Transient
private String translatedName;
```

Entity Identity - The Primary Key

- Each entity class must have a primary atttibute annotated with `@Id`
- An Id can be of the following types
  - Primitive Java types: `byte`, `short`, `int`, `long`, `char`
  - Wrapper classes: `Byte`, `Short`. `Integer`, `Long`, `Character`
  - Array of primitive types or wrapper classes
  - `java.lang.String`, `java.math.BigInteger`
  - Temporal types
  - (Floating Point types are also allowed, but are not recommended because of possible rounding errors)

Primary Key Generation

- Primary Keys can be generated by JPA with the database
- There are four strategies: Sequence, Identity, UUID, Table, and Auto

> Source: https://vladmihalcea.com/jpa-entity-identifier-sequence/

Generation Type

- SEQUENCE allows using a database sequence object to generate identifier values. This is the best generation strategy when using JPA and Hibernate.
- IDENTITY allows using a table identity column, like the MySQL AUTO_INCREMENT.
- UUID generates a UIID as primary key.
- TABLE emulates the database sequence generator using a separate table. This is a terrible strategy, and you shouldn’t use it.
- AUTO picks any of the previous strategies based on the underlying database capabilities. It shouldn't be used.

Sequence

- For JPA and Hibernate, you should prefer using SEQUENCE if the relational database supports it because Hibernate cannot use automatic JDBC batching when persisting entities using the IDENTITY generator.
- Plus IDENTITY must execute the insert statement to get the ID

```
@Entity public class Employee {

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  public Integer id;

}
```

Sequence Generator

- By default, Hibernate creates a HIBERNATE_SEQUENCE sequence
- If you want to use your own sequence, most often per table, you can use the `@SequenceGenerator` annotation

```
@Entity public class Employee {

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE, generator="employee_seq")
  @SequenceGenerator(name = "employee_seq", sequenceName = "employee_seq")
  public Integer id;

}
```

Make sure that the allocation size matches the sequence allocation size!

UUID Issues

- The index pages will be sparsely populated because each new UUID will be added randomly across the B+Tree clustered index.
- There are going to be more page splits because of the randomness of the Primary Key values
- The UUID is huge, needing twice as many bytes as a bigint column. Not only it affects the Primary Key but all the associated Foreign Keys as well.
- More, if you're using SQL Server, MySQL, or MariaDB, the default table is going to be organized as a Clustered Index, making all these problems even worse.
- So, you are better off avoiding using the UUID for entity identifiers. If you really need to generate unique identifiers from the application, then you are better off using a 64-bit time-sorted random TSID instead.

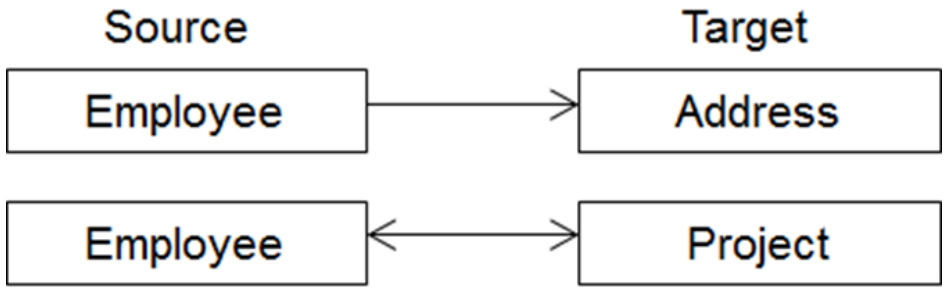
Relationship Mapping

Relationships

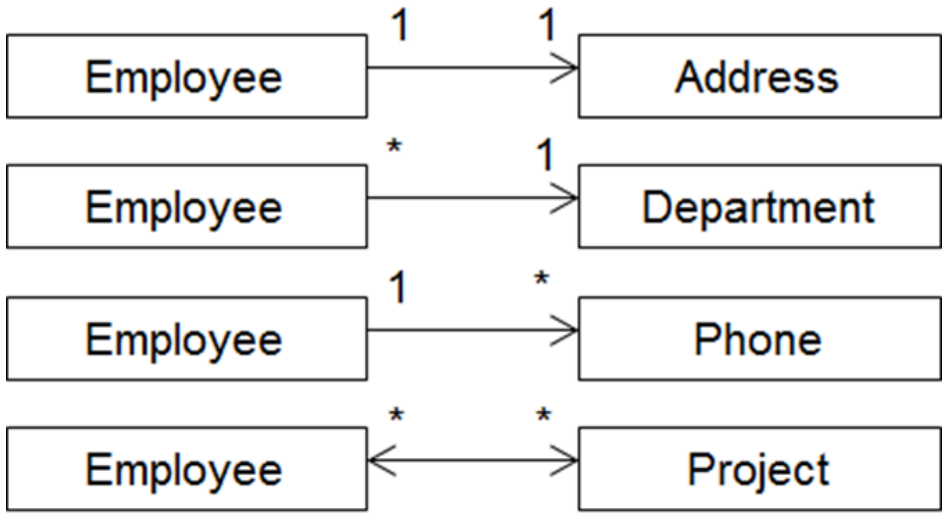
- Relationships between entities are principally represented by corresponding references or collections in the entity classes.
- They must be declared, and details are often necessary for O/R mapping and behavior.
- The following relationship characteristics play a role:
    - Direction
    - Cardinality
    - Aggregation, Composition

Characteristics

- Unidirectional
- Bidirectional



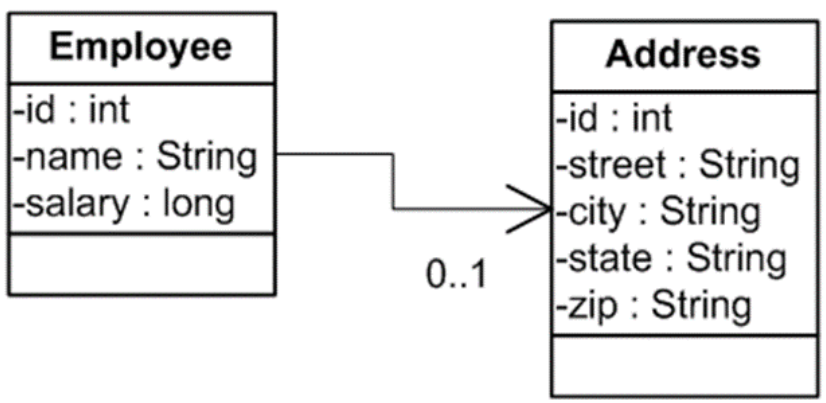
- OneToOe
- ManyToOne
- OneToMany
- ManyToMany

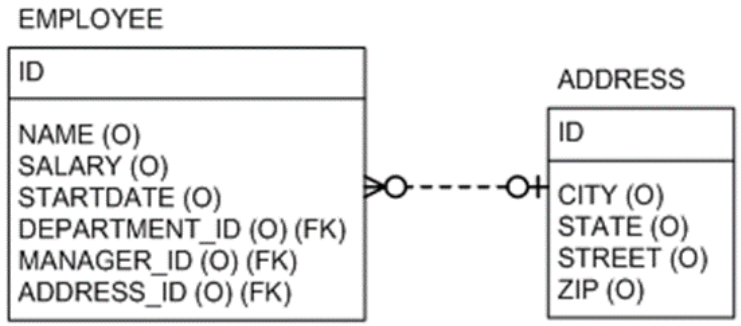


Owning and Inverse Side

- JPA distinguishes between **owning and inverse side**
- **The owning side is responsible for managing the relationship** in the database (Foreign Key)
- The **inverse side** is marked by the attribute `mappedBy`
- In unidirectional relationships, the inverse side is missing

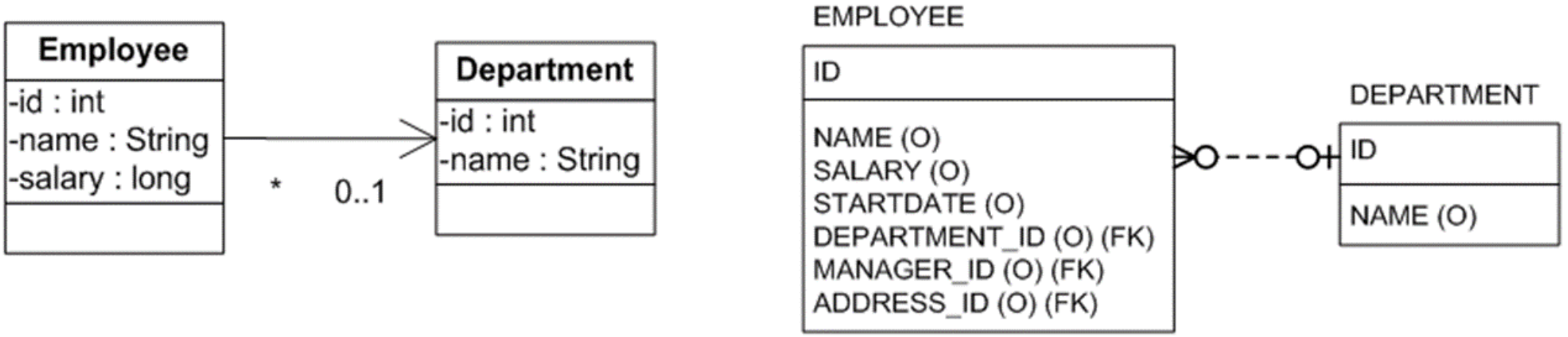
OneToOne, Unidirectional





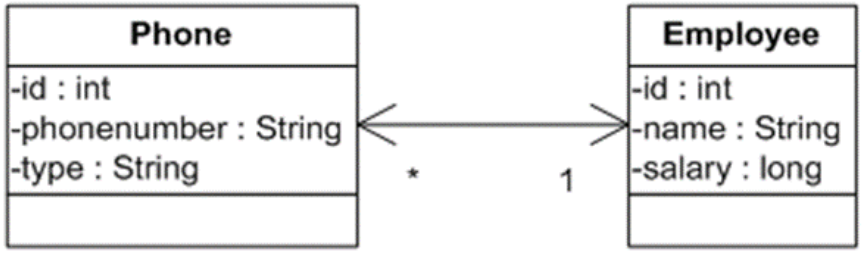

```
public class Employee {

  @OneToOne
  private Address address;
}
```

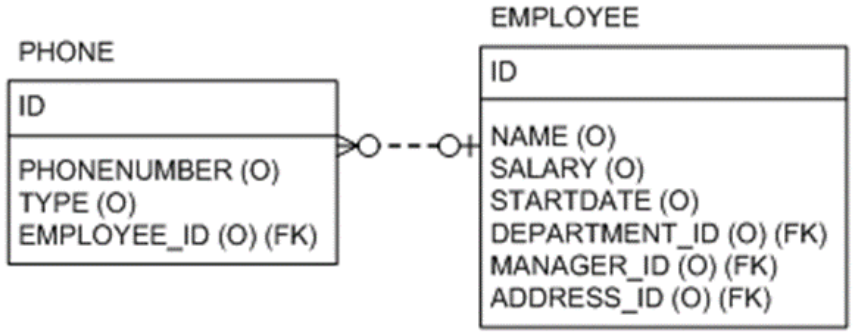
ManyToOne, Unidirectional




```java
public class Employee {

  @ManyToOne
  private Department department;
}
```

OneToMany, Bidirectional

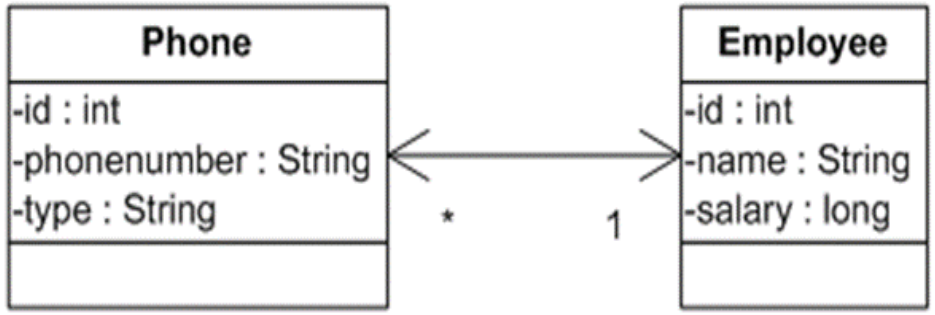


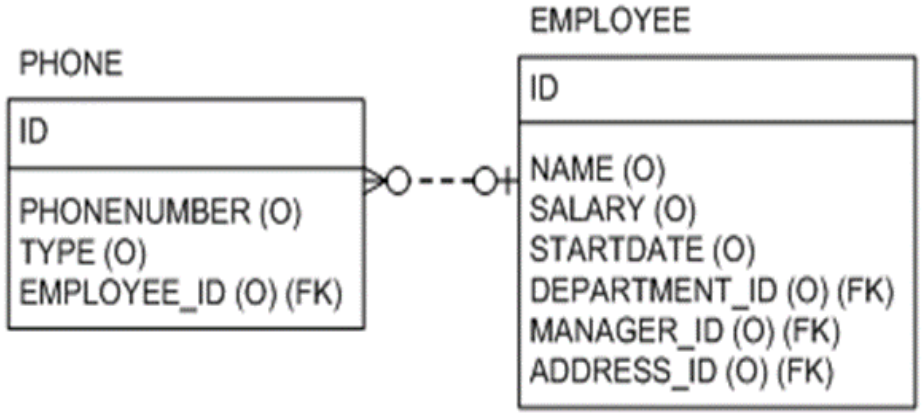



```
public class Employee {
  @OneToMany(mappedBy = "employee")
  private Set<Phone> phones;
}
```

```
public class Phone {
  @ManyToOne(optional = false)
  private Employee employee;
}
```

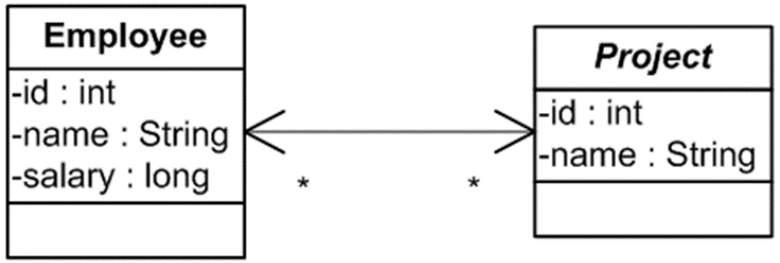
OneToMany, Unidirectional

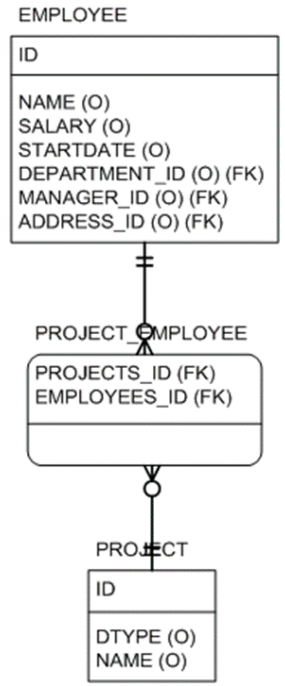





```
public class Employee {

  @OneToMany
  @JoinColumn(name = "employee_id", nullable = false)
  private Set<Phone> phones;
}
```

ManyToMany, Bidirectional





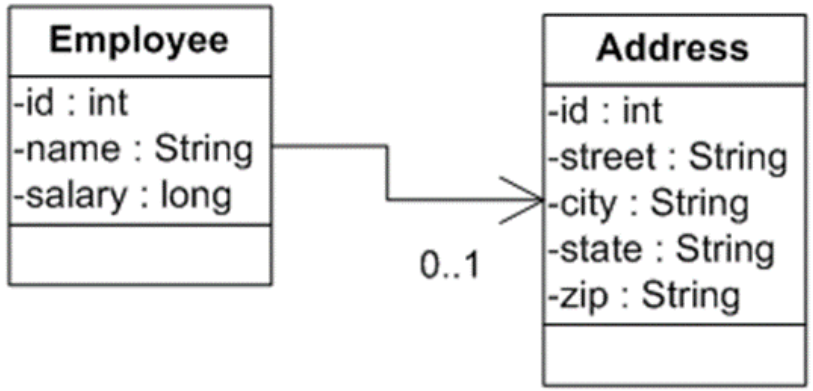
ManyToMany, Bidirectional

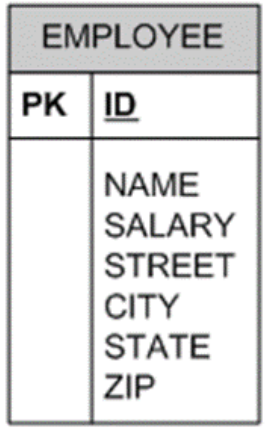
```
public class Employee {

  @ManyToMany(mappedBy = "employees")
  private Set <Project> projects;
}
```

```
public class Projects {

  @ManyToMany
  private Set<Employee> employees;
}
```

Embeddables

- A way to implement composition
- Embedded objects do not have their own identity
- Entity and embeddables are stored in the same table





Example

```
@Embeddable
public class Address {
  private String street;
  private String city;
  private String state;
  private String zip;
}
```

```
@Entity
public class Employee {
  @Id
  private int id;
  private String name;
  private long salary;
  @Embedded
  private Address address;
}
```

Collections of Non-Entities and Embeddables

```
@Entity
public class Employee {

    @ElementCollection
    @Column(name = "PHONE_NUMBER")
    private List<String> phoneNumbers;
}
```

Collections

- `java.util.Set` : unique according to `equals()`

```
@OneToMany
private Set<Phone> phones;
```

- `java.util.List` : ordered, can be sorted

```
@OneToMany @OrderBy("phonenumber ASC")
private List<Phone> phones;
```

- `java.util.Map` : key/value pairs

```
@OneToMany @MapKey(name = "phonenumber")
private Map<String, Phone> phones;
```

Persistent Ordering

- The order of a list can be persistent

```
@Entity
public class Employee {

  @OneToMany
  @OrderColumn(name = "PHONE_POS")
  List<Phone> phones;
}
```

Enhanced Map Support

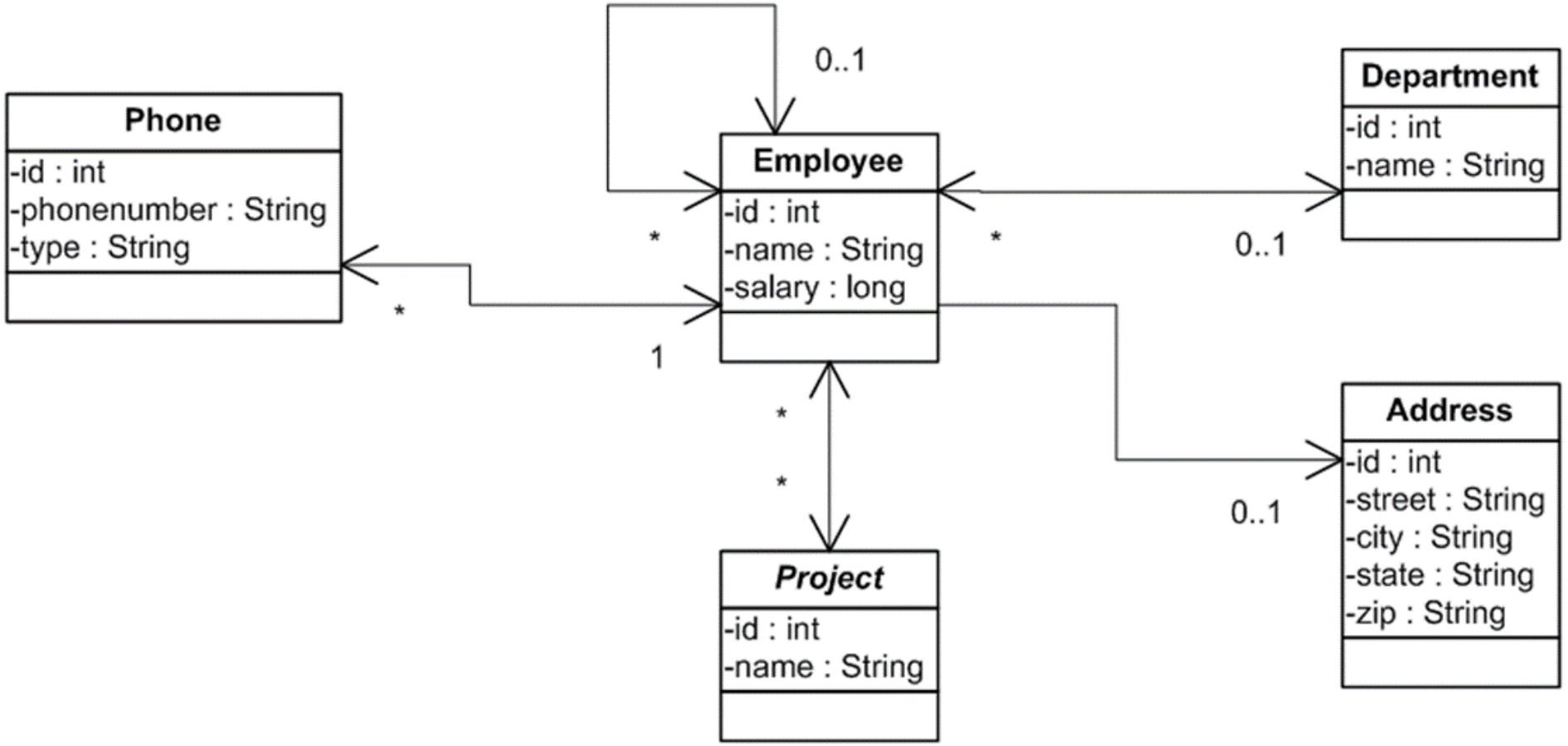
- Use objects, embeddables and entities as map key and value

```
@Entity
public class Vehicle {

  @OneToMany
  @MapKeyJoinColumn(name="PART_ID")
  Map<Part, Supplier> suppliers;
}
```

Exercise: Ids and Relationships

0. Remove Flyway and use Hibernate to create the database model
1. Implement the class model
   (please use Integer instead of int)
2. Select the Id generation strategy
3. Define the relationship types
4. Define the relationship mappings
5. Create Repositories
6. Test your work by creating n Employee with all relationships

Entity Model



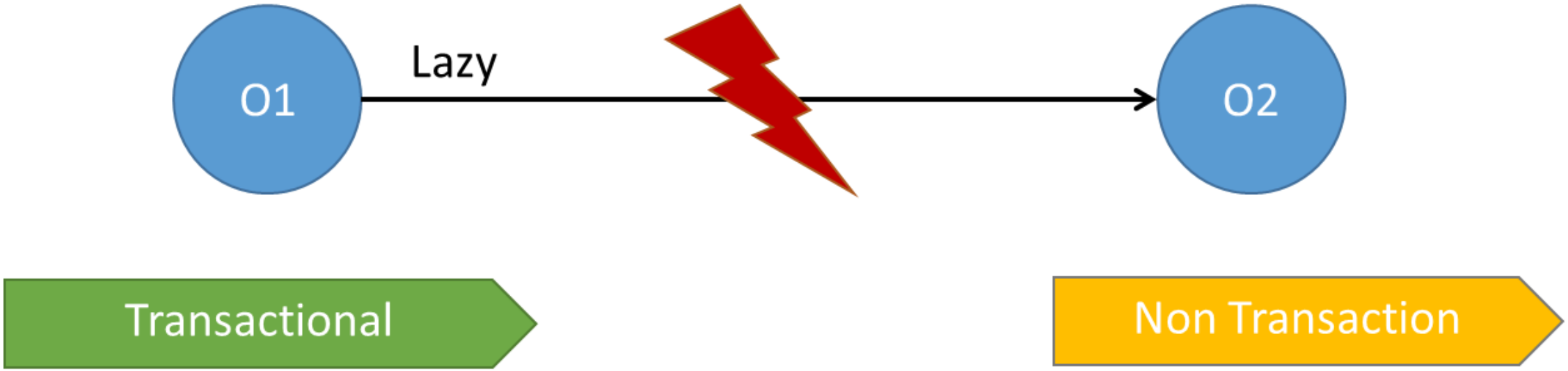
Lazy Loading

Lazy Loading

- Default with `OneToOne` and `ManyToOne` `FetchType.EAGER`
- Default with `OneToMany` and `ManyToMany` `FetchType.LAZY`
- The default behavior can be overridden

```
@OneToMany(fetch = FetchType.EAGER)
private Set<Phone> phones;
```

Problems with Lazy Loading



Solutions

Deactivate Lazy Loading

- Advantage
  - Simple `FetchType.EAGER`
- Disadvantages
  - May lead to performance issues because it may not JOIN the tables
  - Generates n+1 bombs
  - Hibernate can only eagerly load more than one List per Entity
- Recommendation
  - Don’t use! Instead, set all toOne, and toMany annotations to `FetchType.LAZY`

Open Session In View Pattern

- The OpenSessionInView pattern is a pattern that assumes that in an application, the session (EntityManager) is not closed until the request is finished
- Advantage
  - The data is reloaded as needed
- Disadvantages
  - No clean separation of the layers
  - Not all data is loaded in the same transaction
- Recommendation: Don’t use. This is an anti-pattern! Article from Vlad
  Caution: Spring Boot version >=2 has activated this pattern by default
  `spring.jpa.open-in-view=true`

JOIN FETCH

- Advantage
  - Simple

    ```
    select e from Employee e join fetch e.phones p order by e.name
    ```
- Disadvantage
  - Leads to many queries
- Recommendation
  - Use. Very useful and pretty clear what happens.

JPA Entity Graph

```
@Entity
@NamedEntityGraph(name="includePhones", attributeNodes={@NamedAttributeNode("phones")})
public class Employee {

}
```

```
EntityGraph includePhones = em.getEntityGraph("includePhones");
Query query = em.createQuery("select e from Employee e order by e.name");
query.setHint("javax.persistence.loadgraph", includePhones);
return query.getResultList();
```

JPA Entity Graph

- Advantage
    - EntityGraph is reusable
- Disadvantage
    - Leads to lots of annotations
- Recommendation
    - Use if you need to reuse the entity graphs