JPA Workshop

Meniu

Antreu

1. *Salată de JPA, cu istorie* ..... 5 min
   Motivation, philosophy, a pinch of history

Felul 1 – Așezarea-n bază

2. *Mapare de **câmpuri** ca la mama acasă* ..... 8 min
   Basic field mapping, enums, clobs
3. ***Legături** fierbinți înșiropate în bază de date* ... 20 min
   OneToMany & co, ElementCollection, orphan removal
4. *Fasole **moștenită** sau luată din vecini* ..... 10 min
   Inheritance strategies, Embeddable, spicy debate
5. ***Chei primare** proaspăt culese* ..... 5 min
   PK generation strategies, spicy debate

Felul 2 – Viața-n doi (cu JPA)

6. *Vită **leneșă** adusă **harnic** pe masa* ..... 15 min
   Lazy load: concepts, defaults, best practices
7. ***Tranzacție** bine făcută la tavă* ..... 20 min
   Controlling @Transactional, propagation, rollback only
8. *File de **cache** sub acoperire* ..... 20 min
   JPA as a read-write cache, flush, commit
9. *Pește care .**merge**() cu vin* ..... 10 min
   Merge behavior, cascading on children
10. *Biftec de vițel **optimist*** ..... 10 min
    Concurrency control: optimistic vs pessimistic
11. *Rață **căutată** în codru* ..... 20 min
    JPQL, querying, limitations and workarounds
12. *Pachetele cu date de **primăvară** (Spring)* ..... 15 min
    Spring Data: method names, explicit Query, architecture

Desert

13. *Papanași **performanți** (discuție)* ..... 20 min
    N+1 query, useless rows/columns, batch_size, streaming, PKs

Menu

Appetizers

1. *JPA salad, with history* ..... 5 min
   Motivation, philosophy, a pinch of history

1st Course – Laying-down in database

2. *Friendly **field** mapping* ..... 8 min
   Basic field mapping, enums, clobs
3. *Hot **relations** in sweet database syrup* ..... 20 min
   OneToMany & co, ElementCollection, orphan removal
4. ***Inherited** or stolen beans* ..... 20 min
   Inheritance strategies, Embeddable, spicy debate
5. *Freshly picked **primary keys*** ..... 5 min
   PK generation strategies, spicy debate

2nd Course – Life in two (with JPA)

6. ***Lazy** beef eagerly served* ..... 15 min
   Lazy load: concepts, defaults, best practices
7. *Well baked **transaction*** ..... 20 min
   Controlling @Transactional, propagation, rollback only
8. *Undercover **cache** file* ..... 20 min
   Persistence Context as a read-write cache, flush, commit
9. *Fish .**merge**()d with wine* ..... 10 min
   Merge behavior, cascading on children
10. ***Optimistic** veal calf steak* ..... 10 min
    Concurrency control: optimistic vs pessimistic
11. *Duck **queried** in the forest* ..... 20 min
    JPQL, querying, limitations and workarounds
12. ***Spring** Data Packages* ..... 15 min
    Spring Data: method names, explicit Query, architecture

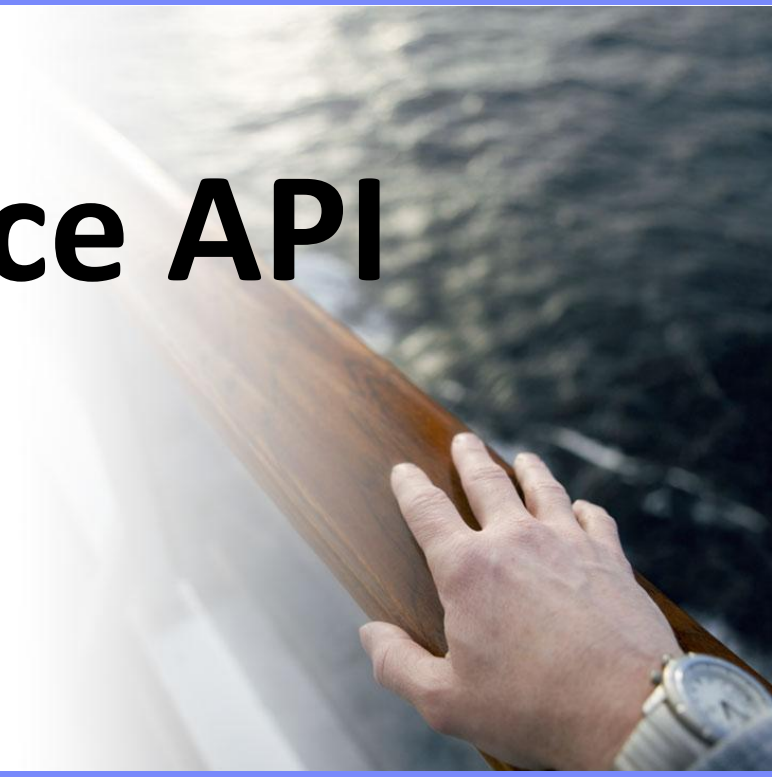
Dessert

13. ***Performant** pancakes (discussion)* ..... 20 min
    N+1 query, useless rows/columns, batch_size, streaming, PKs

- JPA - A la carte menu

Enterprise Java Training

Java Persistence API Best Practices



Victor Rentea

A TRAINING BY VICTOR RENTEA

www.victorrentea.ro

ORM

- Field Mapping
- Collection Mapping
- Inheritance
- Generating @Id
- @Lob

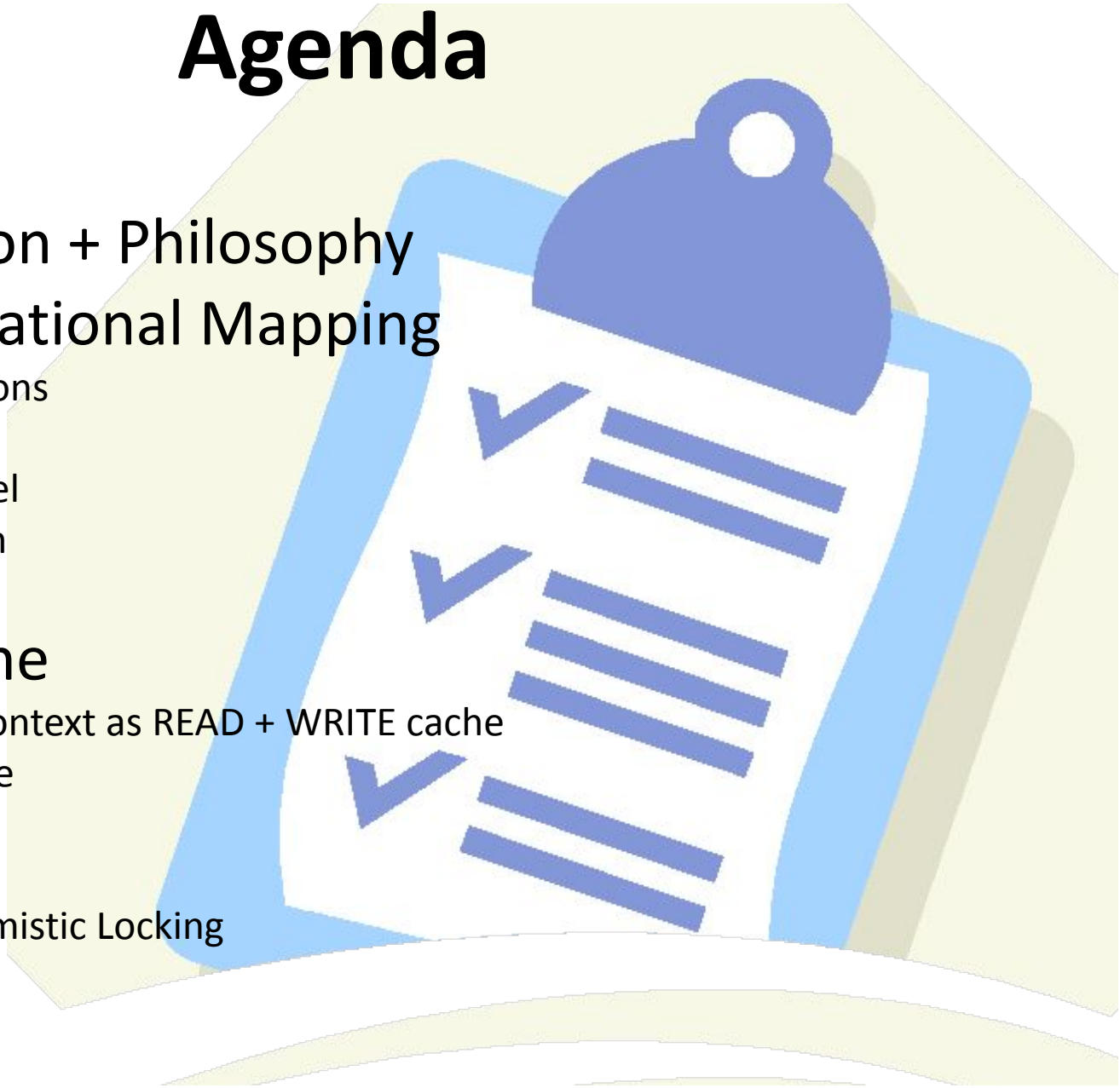
JPA Runtime - Persistence Context

- Entity Lifecycle
- Write-Behind
- 1st Level Cache
- Lazy Loading
- Cascading
- Merge
- Locking

Performance

- N+1 Query
-
- Batching
- Caching

Agenda



- Introduction + Philosophy
- Object-Relational Mapping
  - Storing Relations
  - Embeddable
  - Rich JPA Model
  - Polymorphism
  - Large Objects
- JPA Runtime
  - Persistence Context as READ + WRITE cache
  - Entity Lifecycle
  - Cascading
  - Lazy Loading
  - Merge + Optimistic Locking

Since 2006...

implements the **Repository pattern**
(trying to *make it feel like* the objects are saved in memory)

Java Persistence API

store Java objects to a Relational DB
of which you don't care much

(until your un into performance issues 😊)

a standard with different implementations:

Hibernate

(and others like EclipseLink, TopLink)

DB Agnostic

- Less boilerplate ❤️
  - Less repetition: Customer.lastName, CUSTOMER.LAST_NAME
- Decouple from DB schema ❤️
  - Allows you to think in terms of Objects, not FKs
- Requires less SQL expertise 🪤
  - Until you hit bad performance
- Uses features available in any RDB: no 🔒 -in 😕
  - You'll need native queries only for DB-specific features

```
CONNECT BY
/*+hint */
UTL_MATCH
...
```

The Repository Pattern

= JPA tries to *make you feel like*
the objects are kept in memory

The Repository Pattern

= JPA tries to *make you feel like* the objects are kept in memory

😱 Auto-Flushing 😱☣️

The new salary is persisted automatically (you'll see an UPDATE at the end of the @Transaction)

```
@Transactional
public void updateSalary(Long id) {
   Employee e = employeeRepo.findById(id);
   Double salary = computeSalary(e);
   e.setSalary(salary);
}
```

SELECT FROM EMPLOYEE

UPDATE EMPLOYEE SET SALARY=?

😱 Lazy-Loading 😱☣️

The children of a loaded entity are loaded on demand at your first access (by default)

```
@Transactional
public void printPhones(Long id) {
   Employee e = employeeRepo.findById(id);
   for (Phone phone : e.getPhones())
    log.debug(phone);
}
```

SELECT FROM EMPLOYEE

SELECT FROM PHONES

JPA is a Standard

(JSR-338)

= A Java API

- Interfaces, annotations ...
- Heavily using annotations ~~(or XML)~~
- Reasonable defaults. Hundreds of them. ☺️

+ A PDF

- 550 pages of ❤️ explaining the expected runtime behavior
- One of the most complex standards in Java EE

Developers

Ease of development (KISS)

Performance

@Cacheable (spring)

Spring Data
JpaRepository, @Query

@Cacheable (javax.persistence)

JPA
@Entity, JPQL, Criteria, EntityManager

@Cache

Hibernate
Session, @Cache, @NaturalId, @BatchSize..

JdbcTemplate

MyBatis.org

JDBC
Connection, ResultSet, PreparedStatement

Statement Cache

DB Driver
eg ojdbc4.jar

p6spy

network

in-mem tables
materialized view

SQL ISO
datafiles, indexes, statement cache, partitions, PL/SQL

DB Server-specific
CONNECT BY, DB-specific functions,

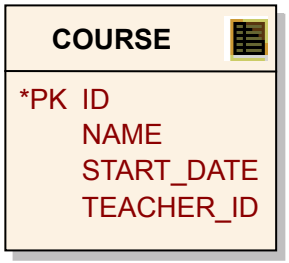
Standard

Entity

= a data structure we persist

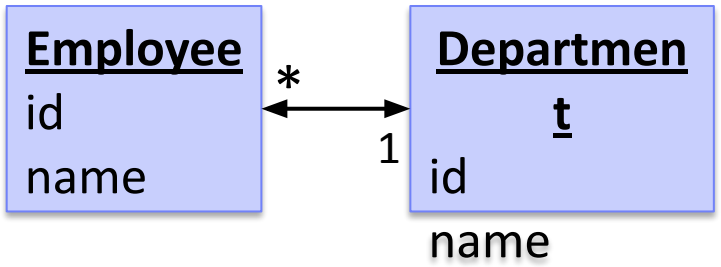
- Mapped to a table
  - Has data fields
  - Has relationships with other entities

```
public class Course {
  private Long id;
  private String name;
  private Date startDate;
  private Teacher teacher;
}
```

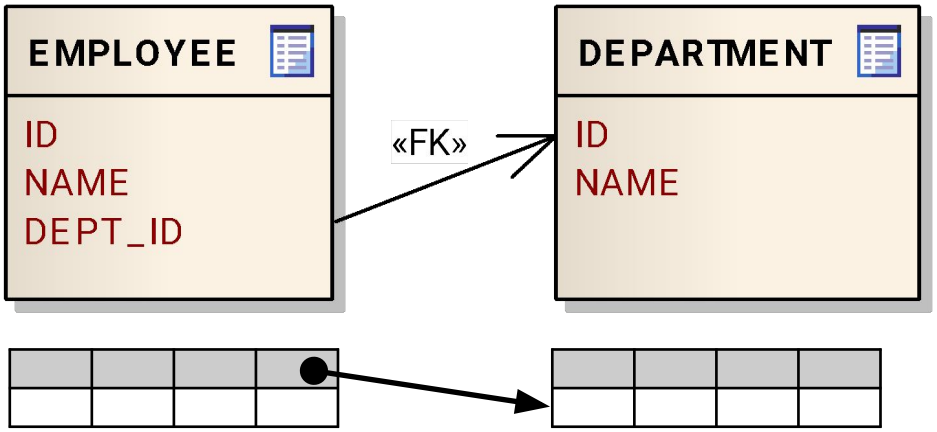


- JPA "watches over" the life of entity instances

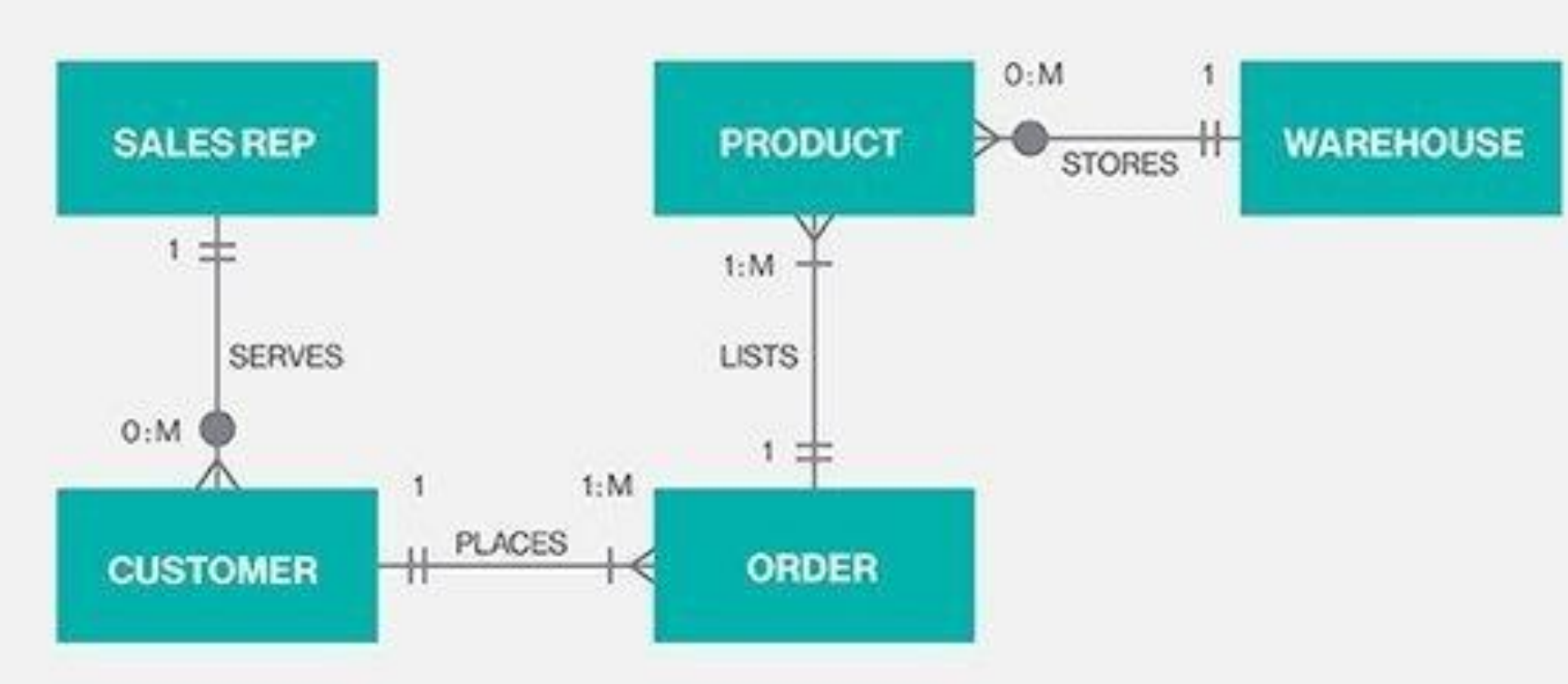
UML Conceptual diagram



Database diagram



Entity-Relationship Diagram (ERD)



Access Mode

JPA reads the state ...

- from private fields
  - If you annotate the @Id field
  - *(using reflection magic)*

or

- via getters/setters
  - If you annotate the @Id getter (no one does it)

```
public class Employee {

  private Long id;


  public Long getId() {
    return this.id;
  }
  public void setId(Long id) {
     this.id = id;
  }
}
```

Mapping Fields


```
@Entity
@Table(name = "PRODUCT_ITEM")
public class Product {
  @Id
  private Long id;

  @Column(name = "PRODUCT_NAME")
  private String name;

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

  private LocalDate startdate;

  @Enumerated(EnumType.STRING)
  private ItemStatus status;

  @Lob @Basic(fetch=LAZY)
  private String description;
}
```

- Make the class an Entity
- Set the table name to "PRODUCT_ITEM"
  (the default here would have been "PRODUCT")
- Every @Entity must have an @Id (a PK)
  The column name here defaults to "ID"
- Override the column default name ("NAME")
  Fields are mapped to DB even w/o @Column
- … or *TIME* or *TIMESTAMP*
  (to specify what DB type should be used)
  Automatically: LocalDate → DATE; LocalDateTime → TIMESTAMP
- Stores "ACTIVE" for ItemStatus.*ACTIVE*
  (without : enum .ordinal() is persisted, eg: 0 for ACTIVE)
  `enum ItemStatus { ACTIVE, INACTIVE,...}`
  Also: to store a shorter key instead → use a @Converter
- BLOB or CLOB in DB.
  Supported field types: String, byte[], Blob, Clob…

  difference? →

Field Converters

```
   // in some entity...
   @Enumerated(EnumType.STRING)
   @Convert(converter = GradeConverter.class)
   private Grade grade;
```

```
public enum Grade {
   LECTURER("L"),
   PROFESSOR("P"),
   CONF("C"),
   ASSISTANT("A");

   public final String dbValue;
   Grade(String dbValue) {
      this.dbValue = dbValue;
   }
}
```

```
@Converter // (autoApply = true)
public class GradeConverter
   implements AttributeConverter<Grade, String> {

    @Override
    public String convertToDatabaseColumn(Grade attribute) {
        return attribute.dbValue;
    }

    @Override
    public Teacher.Grade convertToEntityAttribute(String dbData) {
        return Arrays.stream(Grade.values())
                .filter(g -> g.dbValue.equals(dbData))
                .findFirst()
                .orElseThrow();
    }
}
```

Other use-cases: Storing arbitrary object as JSON in DB 👌

Large Objects

- Avoid storing large objects in Relational DB
  - Store them in a folder (network share), or in a nosql (eg Mongo)

- Problems
  1) Avoid loading the data if not used in a flow
  2) Avoid keeping the whole data in memory when reading/writing (large files)

- But if you must:
  - Use @Lob + String or byte[], or
  - Store in CLOB or BLOB columns (more efficient)

@Lob: (1) Only fetch when needed

(avoid loading the large data accidentally, on use-cases it is not actually required)

1) Lazy basic fields

- A's getter has to be bytecode-enhanced (does not work out-of-the box in Spring Boot)

```
@Entity class A { @Basic(fetch=LAZY) @Lob byte[] data; }
```

2) Lazy OneToOne (B gets proxied)

```
@Entity class A { @OneToOne(fetch=LAZY) B b; }
@Entity class B { @Lob byte[] data; }
```

3) "Backwards" link

- No more direct traversal path from A -> @Lob => no more eager loading by JPA.
- To get the data you must do `bRepo.findByAId(aId).getData()`

```
@Entity class A { … }
@Entity class B {
   @OneToOne A a;
   @Lob byte[] data;
}
```

@Lob: (2) Stream data

(avoid keeping the entire data in memory)

- Use java.sql.Clob/Blob as the type of your column:

```
@Entity class A { ... Clob data; }
```

Then, you can:

1) Download data into a file / send over http response

```
Clob data = entity.getData();
IOUtils.copy(data.getCharacterStream(), fileWriter);
```

2) Upload data directly from a file

```
Clob clob = ClobProxy.generateProxy(fileReader, file.length());
that the jdbc driver uplaods in SQL db.
```

Mapping Relationships

- In Java:

```
public class Employee {
  private Department department; // one
  private List<Customer> customers; // multiple
  ...       Set<Customer
}
```

- In database:
  - Foreign Keys (FK): EMPLOYEE.DEPT_ID = DEPARTMENT.ID

Relationship Concepts

- **Cardinality**
  - 1:1 `@OneToOne`
  - 1:n `@OneToMany : @ManyToOne`
  - n:n `@ManyToMany`

Later, we'll talk about

- **FetchType**
- **Cascade**
- **OrphanRemoval**

- **Bidirectional ↔ or UniDirectional** □

  = the Java relation can be traversed in one/both directions

- **Owner Side** ♥ ●

  = which of the two entities defines the JOIN condition

  - To persist it, only this end of the association matters to JPA ⚠


```java
@Entity
public class Employee {

  @ManyToOne
  @JoinColumn(name = "DEPT_ID")
  private Department department;
  ...
}
```

EMP.DEPT_ID = DEPT.ID

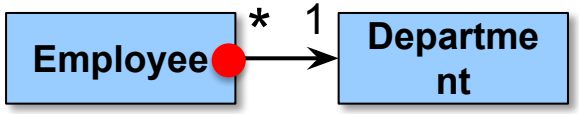
```java
@Entity
public class Department {

  @Id
  private Long id;
  ...
}
```

- The owner side
  - Defines the **@JoinColumn**

Example

links to referentials

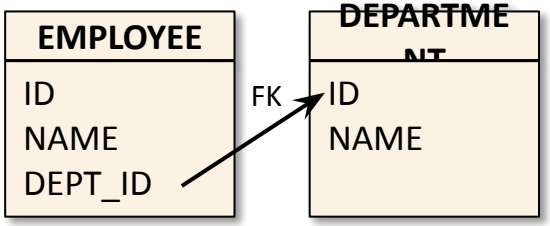
Department: Financial



unidirectional

bidirectional

@ManyToOne




```java
@Entity
public class Employee {

  @ManyToOne
  @JoinColumn(name = "DEPT_ID")
  private Department department;
  ...
}
```

EMP.DEPT_ID = DEPT.ID

```java
@Entity
public class Department {

  @Id
  private Long id;

  @OneToMany(mappedBy="department")
  private List<Employee> employees;
}
```

- The owner side
  - Defines the **@JoinColumn**
- The inverse side
  - mappedBy the owner side

Example 👍
parent-child


```
@Entity
public class Employee {

}
```

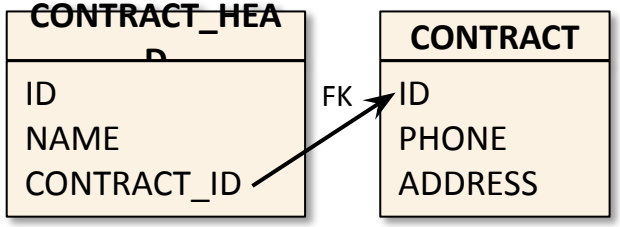
EMP.DEPT_ID = DEPT.ID

```
@Entity
public class Department {
  @Id
  private Long id;

  @OneToMany
  @JoinColumn(name = "DEPT_ID")
  private List<Employee> employees;
}
```

- Can't get its parent

Example 👍
dependent children
Person.addresses

- The owner side
  - Defines **@JoinColumn**
  - Even if the FK column is in the Table of the other entity




```java
@Entity
public class ContractHead {
  @OneToOne

@JoinColumn(name="CONTRACT_ID")
  private Contract contract;
}
```

```java
@Entity
 public class Contract {

  @OneToOne (mappedBy="contrac
                       t")
  private ContractHead head;
  <50 more fields>
}
```

- The owner side
  - Defines **@JoinColumn**
- The inverse side [optional]
  - mappedBy the owner side

Example

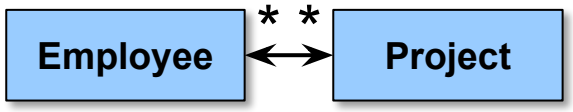
- **Performance**: BriefContract(10 columns) vs FullContract(100 columns)

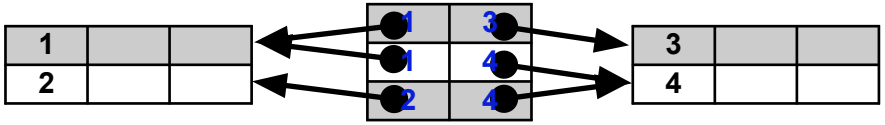
indexing and searching the BriefContract is faster

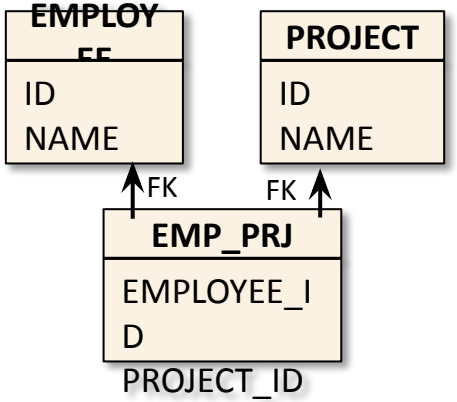
- **Business Stages**: Contract -> ContractBilling (assigned later in flow)

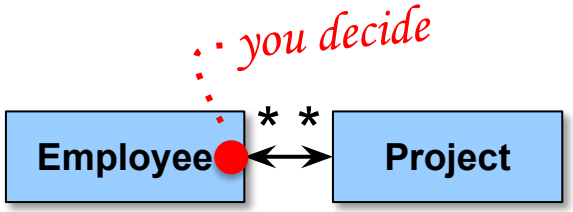
makes the domain cleaner also

Many-to-Many


Employee
* *
Project

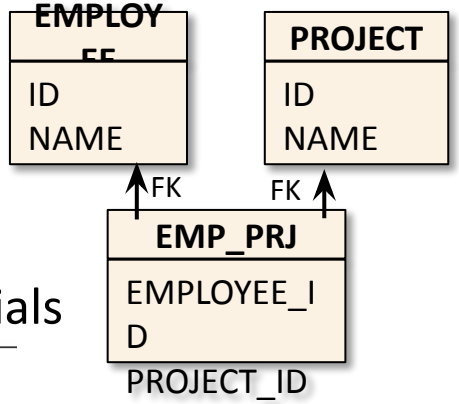

1
2
1 3
1 4
2 4
3
4


EMPLOYEE
ID
NAME
PROJECT
ID
NAME
FK
FK
EMP_PRJ
EMPLOYEE_ID
PROJECT_ID



uni- or bi-

@ManyToMany

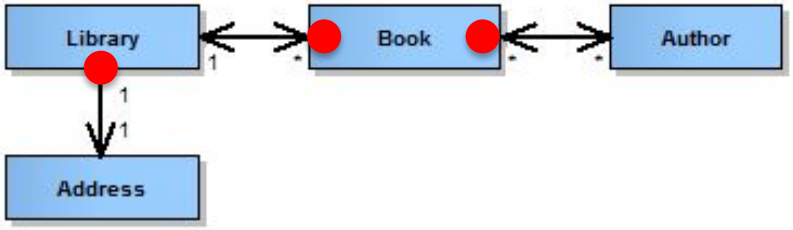


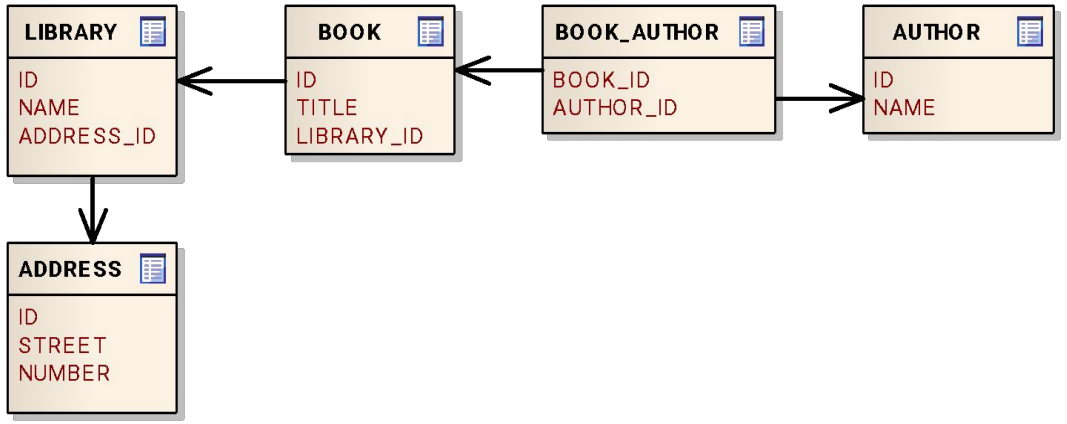
Example multi-select referentials

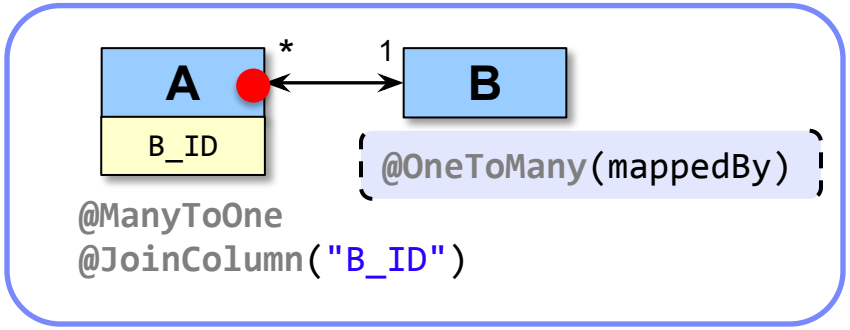

```
@Entity
public class Employee {
  @ManyToMany
  @JoinTable(name ="EMP_PRJ",
        joinColumns="EMPLOYEE_ID",
 inverseJoinColumns="PROJECT_ID")
  private List<Project> projects;
}
```

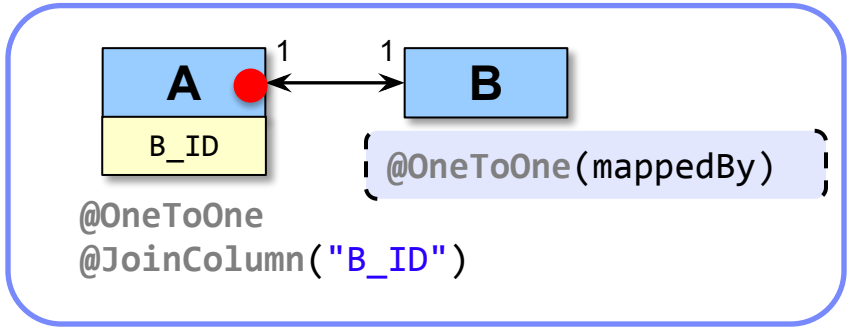
```
@Entity
public class Project {

  @ManyToMany (mappedBy="projects")
  private List<Employee> employees;
}
```

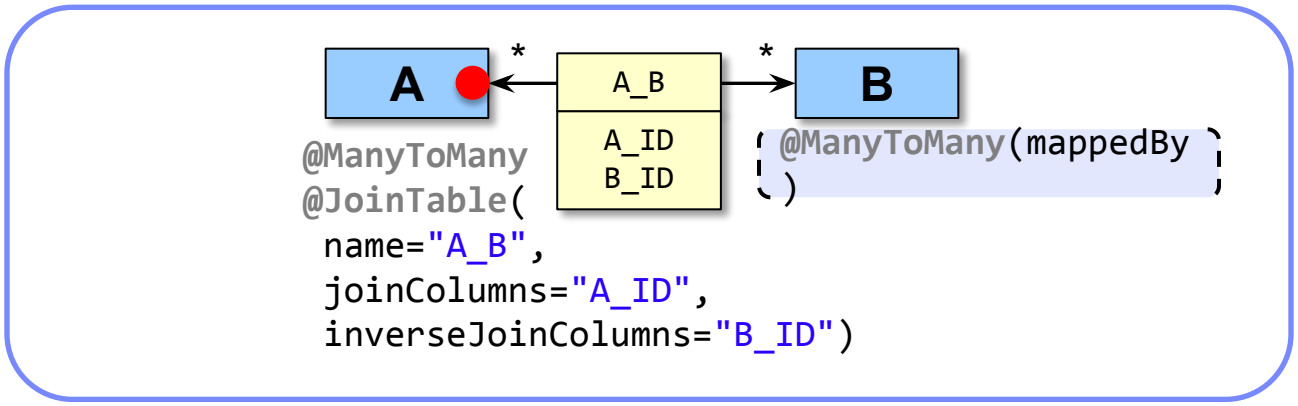
- The owner side
  - Defines the @JoinTable
- The inverse side [opt]
  - mappedBy= the owner side


Library
Book
Author
1
1
1
1
Address


LIBRARY
ID
NAME
ADDRESS_ID
BOOK
ID
TITLE
LIBRARY_ID
BOOK_AUTHOR
BOOK_ID
AUTHOR_ID
AUTHOR
ID
NAME
ADDRESS
ID
STREET
NUMBER


A
B_ID
*
1
B
@OneToMany(mappedBy)
@ManyToOne
@JoinColumn("B_ID")

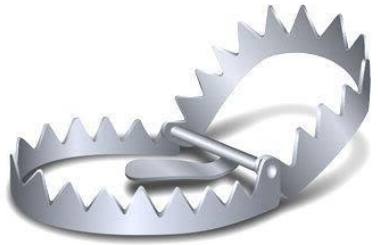

A
B_ID
1
1
B
@OneToOne(mappedBy)
@OneToOne
@JoinColumn("B_ID")


A
*
A_B
A_ID
B_ID
*
B
@ManyToMany(mappedBy
)
@ManyToMany
@JoinTable(
name="A_B",
joinColumns="A_ID",
inverseJoinColumns="B_ID")

In bidirectional associations,

Don’t forget to set the OWNER side !!

- Setting only the inverse side doesn’t persist the link

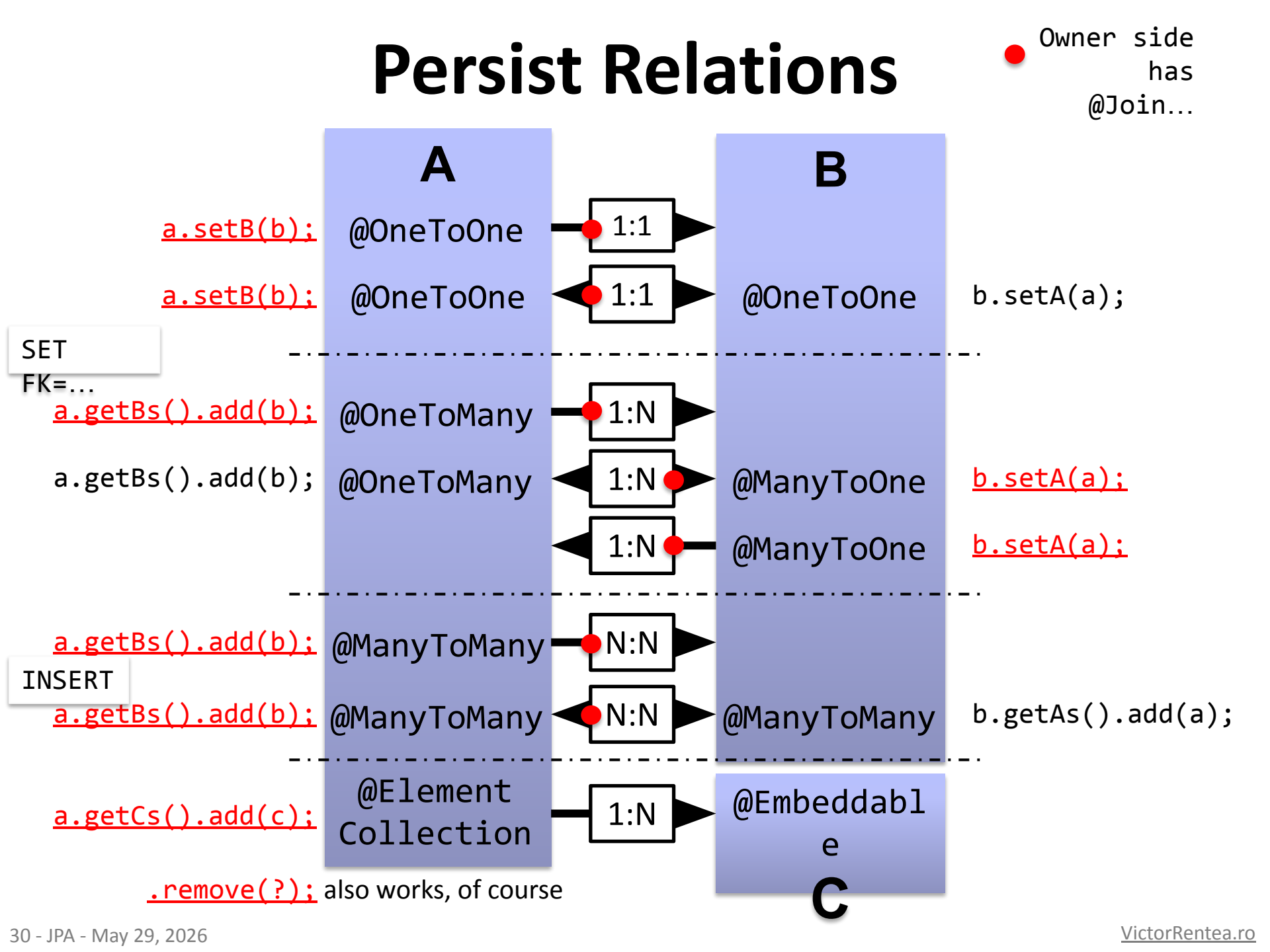


```java
@Entity
public class Employee {
  @ManyToOne
  @JoinColumn(name="DEPT_ID")
  private Department department;
}
```

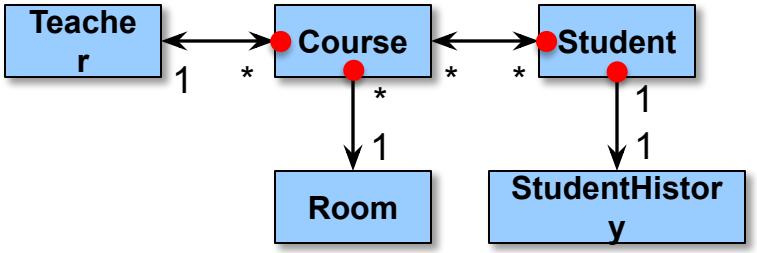
```java
@Entity
public class Department {

  @OneToMany(mappedBy="department")
  private List<Employee> employees;
}
```

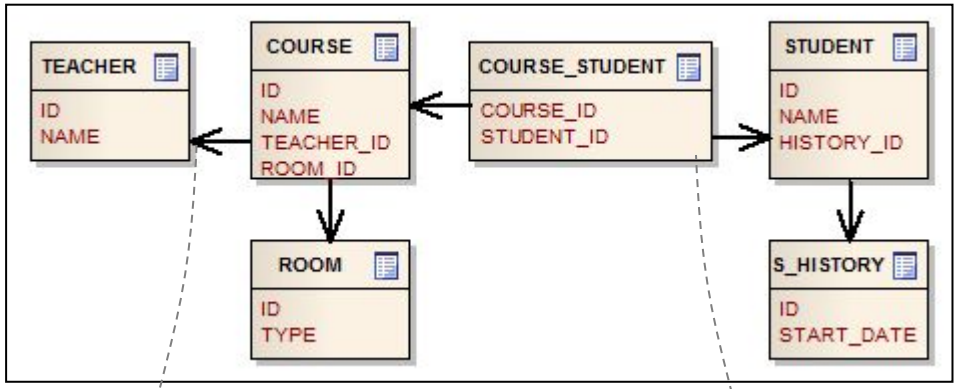
- .setDepartment(dept)
  - Persists the link in DEPT_ID 
- employees.add(emp)
  - Doing just this is NOT enough 

Persist Relations



ORM Quiz (10 min)






```
@Entity
public class Teacher {

  1
  private Long id;

  private String name;

  2
@OneToMany(mappedBy="teacher")
  private List<Course> courses;

}
```

```
3
public class Course {
  4
  @JoinColumn(name="
  5
)
  private Teacher teacher;
  6
  7
@JoinColumn
(name="ROOM_ID")
  private Room
  8

  @ManyToMany(mappedBy="courses")
  private List<Student> students;
}
```

Some field are omitted for brevity

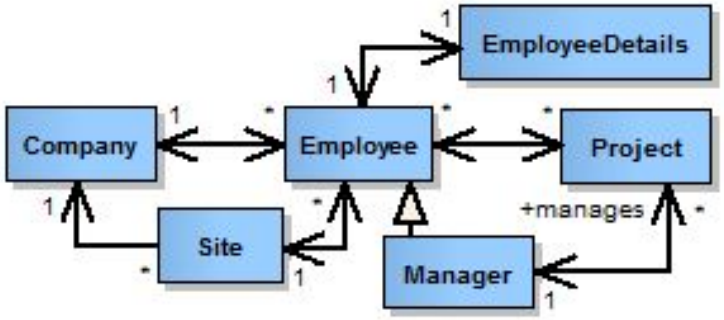
```
@Entity
public class Student {
  9
@JoinTable(name="
",
joinColumns="
"
  10
private List<Course> courses;
inverseJoinColumns="
")
  11
  12
private StudentHistory history;
}
```

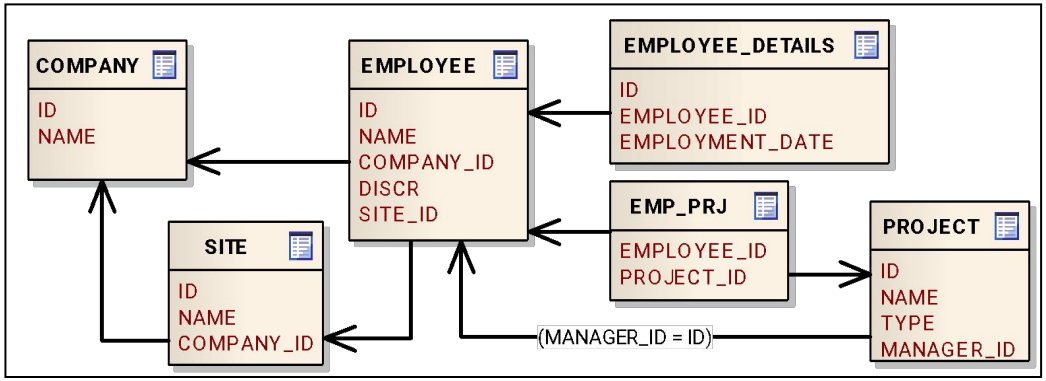
```
@Entity
public class Room {

  16
(EnumType.STRING)
  private RoomType type;
}
```

```
public enum RoomType {
  AULA, CLASSROOM, LIBRARY
}
```

```
@Entity
  13
public class
{
  14
  15
(TemporalType.DATE)
  private Date enrollmentDate;
}
```

ORM Quiz #2 (25 min)






```
@Entity
public class Company {
  @Id
  private Integer id;
  private String name;

  @OneToMany(mappedBy = "company")
  private List<Employee> employees;
```

```
@Entity
(strategy=
(name="DISCR")
@DiscriminatorValue("EMPLOYEE")
Employee
{
  @ManyToOne
  @JoinColumn(name = "COMPANY_ID")
  private Company company;
  @OneToOne (mappedBy =
"employee")
  private EmployeeDetai
  @ManyToMany(mappedBy="employees"
)
  private List<Project> projects;
  @ManyToOne
```

```
@Entity
"EMPLOYEE_DE
EmployeeDetails
loyeeDetails
(TemporalType.DATE)
  @Column
(name="EMPLOYM
  private Date startDate;
  @OneToOne
  @JoinColum
```

```
public class Site {
  private Integer id;
  private String name;
  @JoinColumn(name=
  private Company company;
mappedBy=
  private List<Employee>
employees;
```

```
@Entity
("MANAGER")
class Manager extends
Employee{
  @OneToMany(mappedBy = "manager")
```

```
@Entity
public class Project {
  private Employee employee;
  @JoinTable(name="
",
joinColumns=@JoinColumn(name="
"),
inverseJoinColumns=@JoinColumn(name="
EMPLOYEE_ID"))
<Employee>
employees;
(enum)
type;
  @ManyToOne
```

Set<> vs List<> of children ?

for @OneToMany and @ManyToMany

If it's a HashSet, how about the hashCode/equals?

- Set = the default option
  - Implementing hashCode/equals correct in an Entity is surprisingly HARD
    - Be very careful about @Id (link)
    - Don't implement them at all (link) and rely on the 1st level caching to give you == instances (more on that later)
- List, if order matters
  - But the default row order in ResultSet is not reliable without ORDER BY, so:
  - **A) User-controlled child order**

in UI: 

```
@OneToMany
@OrderColumn
private List<Address> addresses;
```

adds a hidden column managed by JPA: ADDRESS.INDEX

  - **B) Pre-sort the children**

```
@OneToMany
@OrderBy("type ASC, valueStr ASC")
private List<Address> addresses;
```

- No link at all. Instead:
  (eg: if too many children)

```
childRepo.getByParent(parentId, page);
```

I have a large table

(40 columns)

so I must have a large @Entity

(40 fields)

Move groups of related fields
into separate @Embedded objects

👌

Persistence Concerns should not be decisive in the way you model your Domain Objects

Embeddable

```
@Entity
public class TeachingActivity {
  @Enumerated(EnumType.STRING)
  @Column(name = "DAY")
  private DayOfWeek day;

  @Column(name = "START_HOUR")
  private int startHour;
  …
}
```

- The same group of fields repeats in 2+ entities — DRY
- Break-down a long entity — SRP
- You can then add brains & constraints to those new smaller classes

extract

```
@Embeddable
public class TimeSlot {
  @Enumerated(EnumType.STRING)
  @Column(name = "DAY")
  private DayOfWeek day;

  @Column(name = "START_HOUR")
  private int startHour;
  …
}
```

Identical DB Tables

```
@Entity
public class TeachingActivity {

  @Embedded
  private TimeSlot timeSlot;
  …
}
```

Any @Entity embedding it will have these columns in its table

If you want the column to be named TIME_SLOT_DAY, set

- spring.jpa.hibernate.naming.implicit-strategy=org.hibernate.boot.model.naming.ImplicitNamingStrategyComponentPathImpl

@ElementCollection of @Embeddable

```java
@Entity
public class TeachingActivity {

  @ElementCollection
 [@CollectionTable(<customize>)]
  private List<TimeSlot> timeSlots;
  …
}
```

```java
@Embeddable
public class TimeSlot {
  @Enumerated(EnumType.STRING)
  @Column(name = "DAY")
  private DayOfWeek day;

  @Column(name = "START_HOUR")
  private int startHour;}
```

No PK

ACTIVITY	
ID	…
7	…

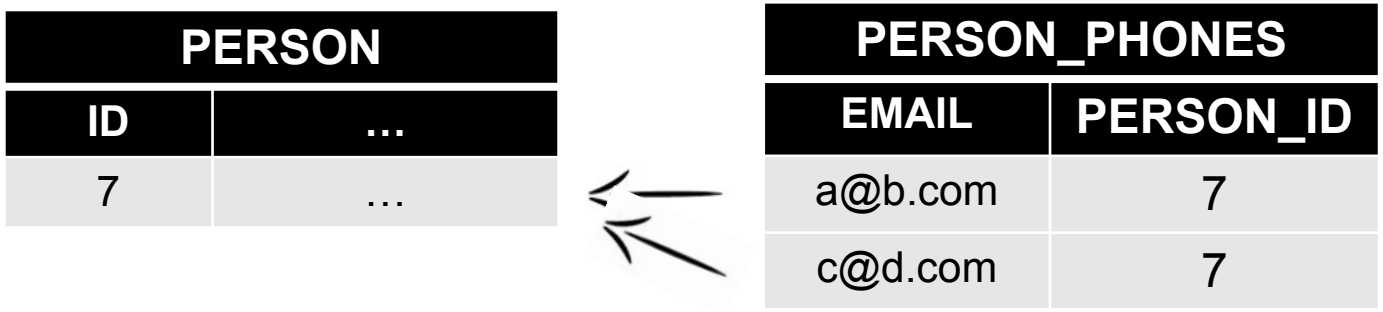
ACTIVITY_TIME_SLOT		
TA_ID	DAY	START_HOUR
7	MONDAY	8
7	FRIDAY	14

vs. @OneToMany

- No PK □ No FK to it □ *private children*
- Cannot be directly persisted, queried, removed
- Any update on any child causes DELETE+INSERT of all
  (performance hit if many children)

@ElementCollection of primitives

```java
@Entity
public class Person {

  @ElementCollection
  private List<String> email;
  …        Set<Long>
}          Map<Integer,String>
```



Inheritance

Remember:

Favor Composition over Inheritance

Inheritance

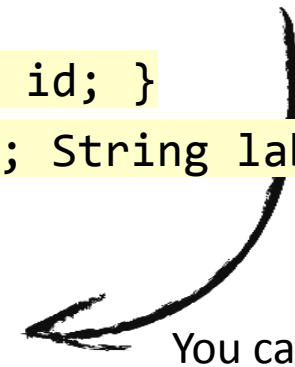
An @Entity …

- Can extend a regular class **A**
  - Any fields in **A** will not be persisted to DB

`{id, label}`

- Can extend a `@MappedSuperclass` **B**
  - And inherit persistent fields from **B**
  - eg: `@MappedSuperclass BaseEntity { @Id Long id; }`
  - eg: `@MappedSuperclass BaseRef { @Id Long id; String label; }`

- Can extend another `@Entity` **C**
  - **C** should be **abstract** 👌



You can
.find(), query, persist(), …
a C

Where to persist the subclass data?

- All in a ***SINGLE_TABLE***
  - Simplest. Best Performance
  - When many *shared* columns
  - Lots of NULLs
  - Subtype columns could collide (eg. age)

RECORD				
ID	TYPE	NAME	SEX	FLAG
1	PERSON	John D	M	-
2	SHIP	Marina	-	RO

shared *specific*

- ***TABLE_PER_CLASS***, one per subtype
  - When many *specific* columns
  - Can use child-specific FKs, NOT NULL
  - SELECT FROM Record requires UNION
    - Not needed? MappedSuperclass

PERSON		
ID	NAME	SEX
1	John D	M

ADDRESS

SHIP		
ID	NAME	FLAG
2	Marina	RO

CREW

- Separate ***JOINED*** tables
  - Lots of joins

PERSON	
ID	SEX
1	M

RECORD		
ID	~~TYPE~~	NAME
1	~~PERSON~~	John D
2	~~SHIP~~	Marina

SHIP	
ID	FLAG
2	RO

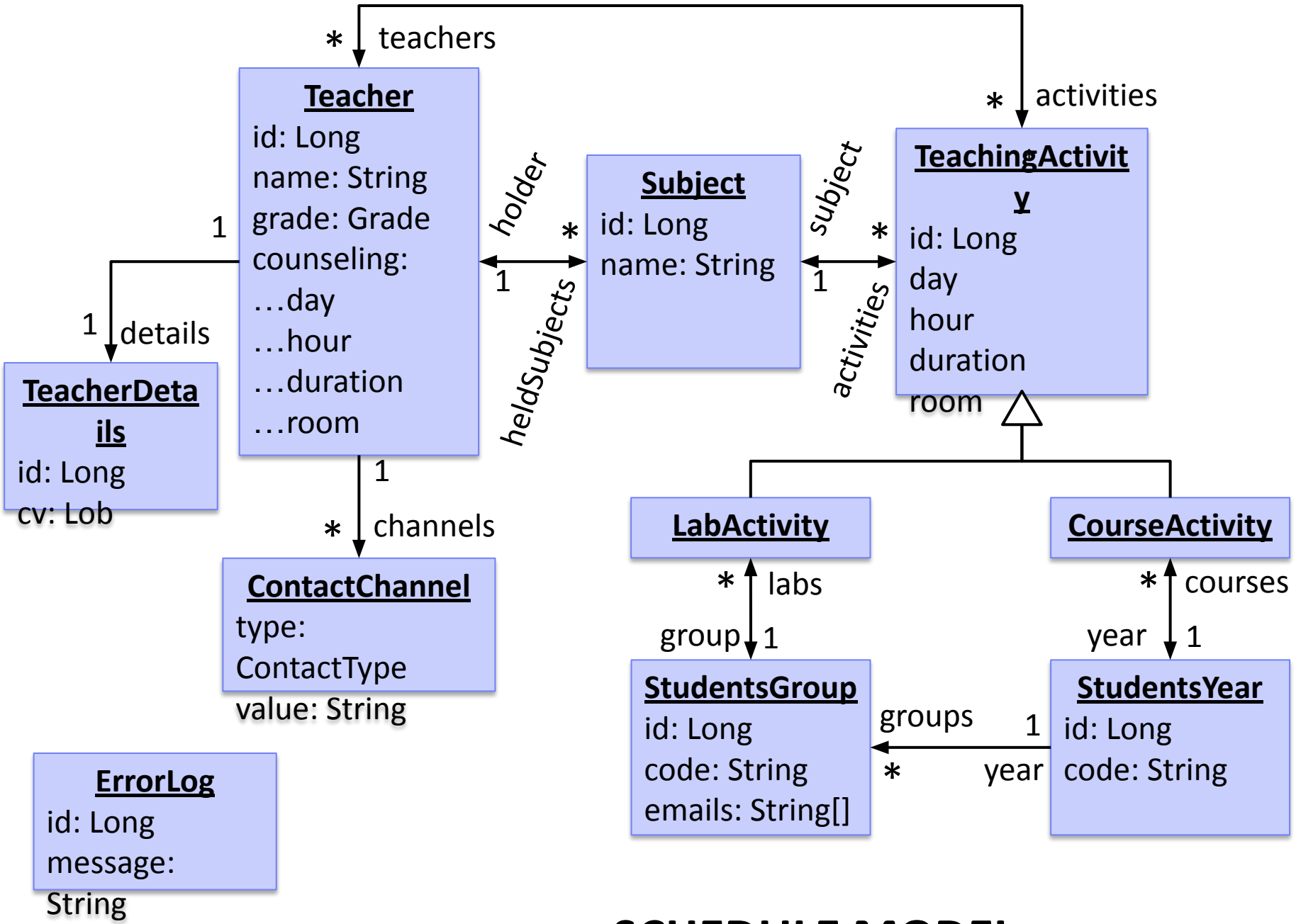
Inheritance Discriminator

RECORD				
ID	TYPE	NAME	SEX	FLAG
1	PERSON	John D	M	-
2	SHIP	Marina	-	RO

```
@Entity
@Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorColumn(name="TYPE")
public abstract class Record {
   private String name;
}
```

```
@Entity
@DiscriminatorValue("PERSON")
class Person extends Record {
  private Sex sex;
}
```

```
@Entity
@DiscriminatorValue("SHIP")
class Ship extends Record {
  private String flag;
}
```

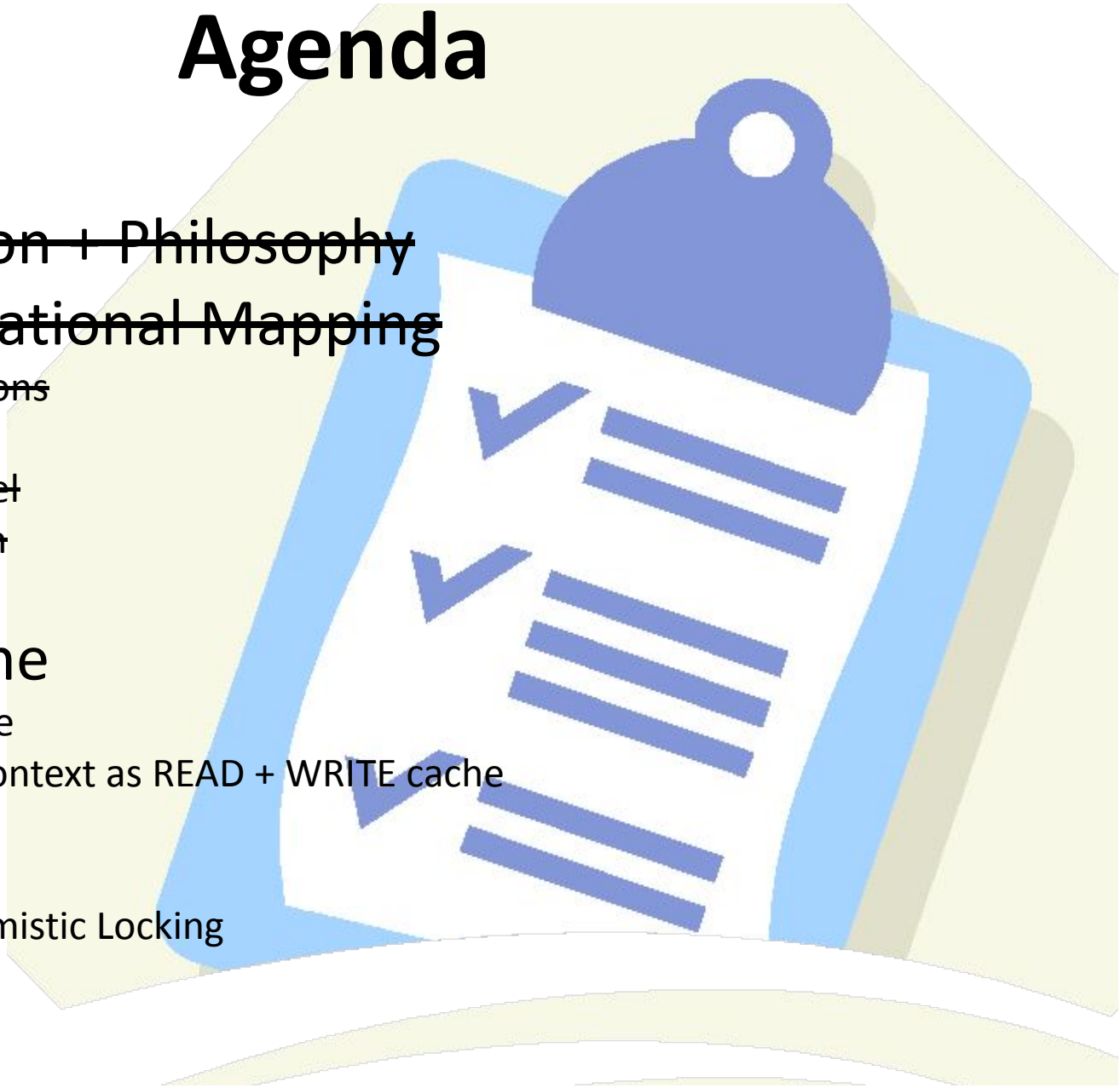


SCHEDULE MODEL

Modeling Rich👑 @Entities

- Encapsulate collections
  - 👌 Guard bidirectional links [avoid]
  - ☐ expose unmodifiable collections via getter
  - ☐ provide add/remove child methods in parent @Entity that sets both ends <->
- Enforce constraints
  - 👌 Validating constructor (+a hidden protected one for Hibernate)
  - ☐ Guard fields against null, min size, or more complex consistency rules
  - 👌 Smart "mutator" methods (instead of setters)
  - ☐ Guard a multi-field Domain Invariant: Contract.status=APPROVED +.approvedBy
- "Immutable" @Embeddable
  - 👌 Decompose a large @Entity into smaller immutable parts
  - ☐ no setters on @Embeddable (JPA read/writes private fields via reflection anyway)

Agenda



- ~~Introduction + Philosophy~~
- ~~Object-Relational Mapping~~
  - ~~Storing Relations~~
  - ~~Embeddable~~
  - ~~Rich JPA Model~~
  - ~~Polymorphism~~
  - Large Objects
- JPA Runtime
  - Entity Lifecycle
  - Persistence Context as READ + WRITE cache
  - Cascading
  - Lazy Loading
  - Merge + Optimistic Locking

Concepts

- **Persistence Unit**

  = a set of @Entity classes

  - Student, Course, and Teacher

- **Persistence Context**

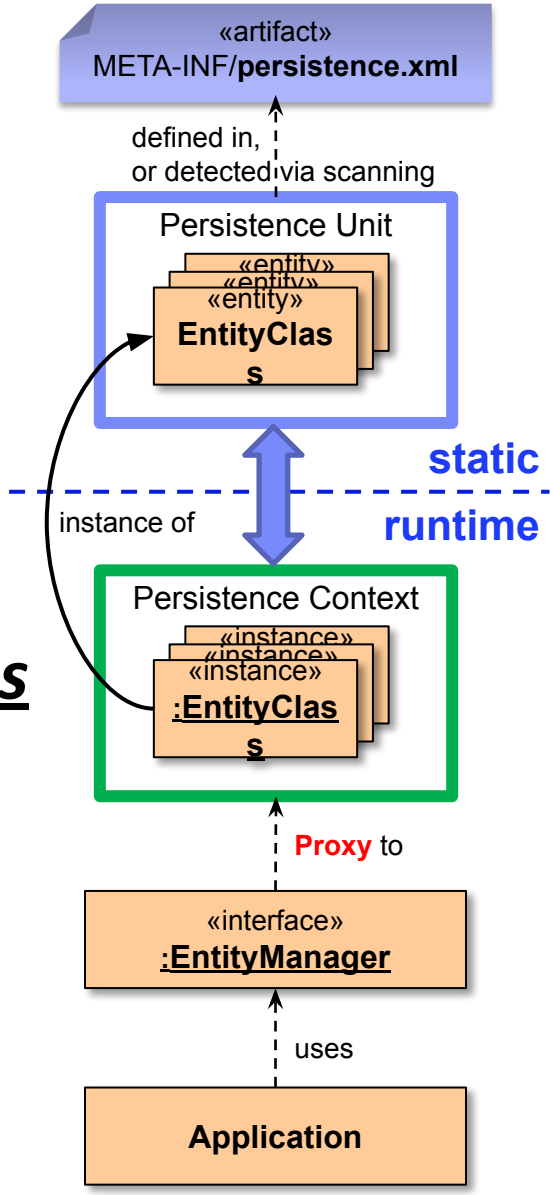
  = a pool of managed @Entity instances

  - The employee "John", "Mike", the Project "Colibri"

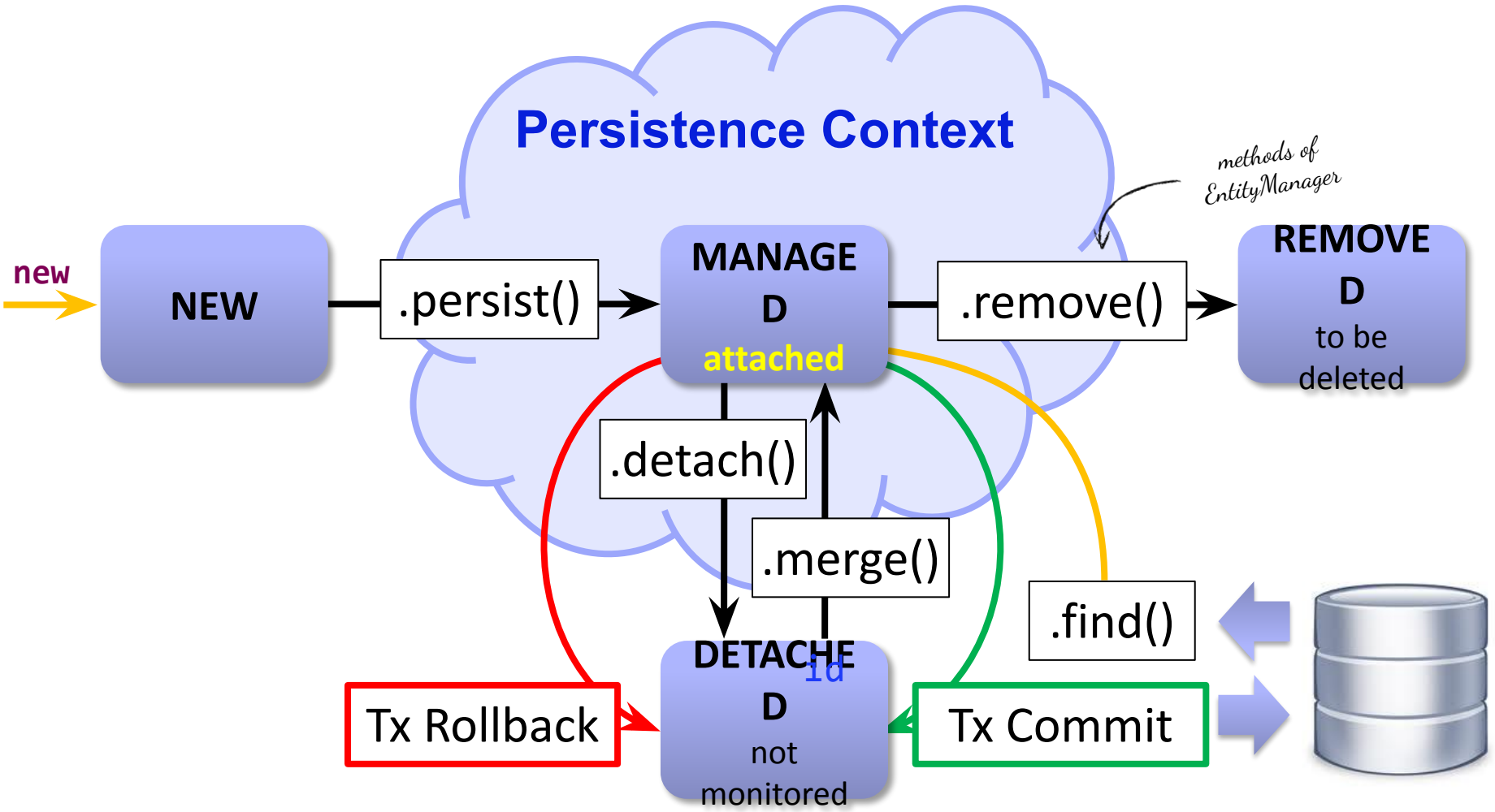
- **Entity Manager**

  = the JPA API

  - A proxy to the current Persistence Context



Entity Lifecycle



Inserting

- springDataRepo.save(**e**)
  - Raw JPA: entityManager.persist(***e***) within an open @Transactional
  - If the @Id is @GeneratedValue:
    - Then **e**.id has to be null,
    - It **will be set by JPA during the save() method**

@Id

- Number
  - Faster DB (smaller index, partition-friendly)

  or

- String
  - Natural Key (eg VAT code), meaningful ☺️
  - UUID (or similar): 694aab7e-0506-11e8-ba89-0ed5f89f718b

Strategies to Generate an @Id

```
@Id
@GeneratedValue(strategy=...)
private Long id;
```

Numeric

- Sequence 👌

```
@SequenceGenerator(name="MyGenerator", allocationSize=50)
```

  - Fastest: can be allocated in blocks (eg INCREMENT BY 50) □ less DB round-trips
- Identity
  - Each INSERT returns the assigned PK □ No BATCHING is possible
- ~~Table~~
  - Portable but slow
- Manually assigned string (UUID/natural)
  - Natural Key or Composite PK □ turn it into Unique Key and create a numeric PK 👌
  - UUID: ⚠️ repo.save() of a new entity does a prior SELECT before the INSERT
    □ can be avoided with custom ID generator (see GeneratedUUIDTest.java)

Inserting

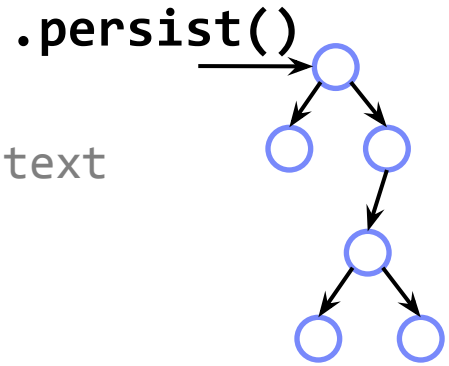
- springDataRepo.save(**e**)
  - Raw JPA: entityManager.persist(***e***) within an open @Transactional
  - If the @Id is @GeneratedValue:
    - Then **e**.id has to be null,
    - It **will be set by JPA during the save() method**

- Entities referenced by **e** can be:
  a) Attached entities, got via .find, .persist, …
  b) Proxies to entities in database, created via b=em.getReference(id, B), or just b=new B(id) □ at INSERT, ***e*** will be FK-ed to a b with that ID
  c) Unattached entities, if you cascade PERSIST to them

Cascading Operations

- The methods of the EntityManager:

```
.persist(entity)
.remove(entity)
.detach(entity) // ... from Persistence Context
.merge(entity) // you'll see…
.refresh(entity) // = revert all changes
```



- … can be cascaded on the relationships:
  - Best use from "Parent to Child"
  - ⚠️ Gets tricky with merge (later on…)
  - Handy for persisting data in unit tests

```
@Entity
public class Student {
  @OneToOne(cascade = PERSIST)
  private StudentHistory history;
  ...
}
```

orphanRemoval=true

- On @OneToOne
  - Set the field to null □ the previously referenced entity is DELETED from DB
- On @OneToMany
  - Remove a child from the collection □ that child will be DELETED from DB

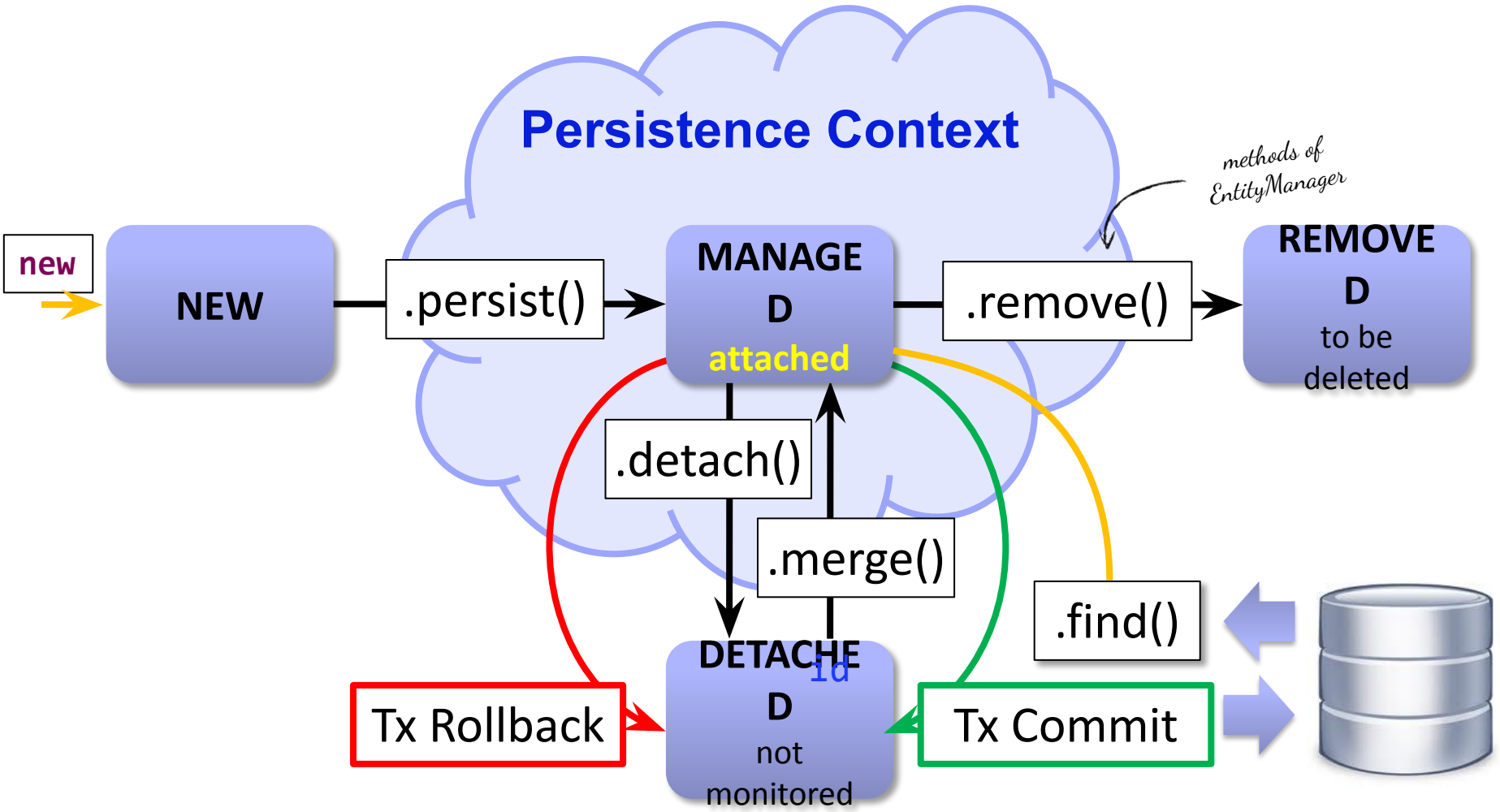
👌 Use for exclusive children

Deleting

- em.remove(*e*)
  - Requires a surrounding Tx
  - *e* must be attached

- Can be cascaded on relations
  - eg. for deleting parents + dependent children

Entity Lifecycle



.find()

Load an entity from DB by PK:

```
Customer customer = entityManager.find(Customer.class, id);
```

```
= customerRepo.findById(id);
```
(Spring Data JPA)

Persistence Context = 1st level cache

- In the same Tx, an entity is never loaded twice
  - The second time, JPA will give you the first instance (==)
- ...anyway you request it:
  - Find by ID
  - JQPL query
  - A relation of another loaded entity,...
  - em.merge()
- Persistence Context acts as a **read cache** withing the current transaction
  - bound to the current thread

The need for Lazy Loading

```
Customer customer = customerRepo.findById(id);
```

What would you expect to find in the returned entity?

1) customer.name: String ✓
2) customer.country: @Entity Country ✓
3) customer.phones: List<@Entity Phone> 🤔

***Remember**: JPA tries to give you the illusion that the objects are persisted in memory (=Repository Pattern).*

Lazy Loading

- **EAGER** = loaded before JPA gives you the entity

```
Course course = courseRepo.findById(courseId);
        // or courseRepo.search(...)
String name = course.getName(); //already loaded
```

- **LAZY** = loaded from DB first time you access it

```
for (Student student : course.getStudents()) {
  ...
}
```

when entering the loop, a query is ran:

```
SELECT ... FROM STUDENT s .... WHERE s.COURSE_ID=?
```

How the hack?... *Exactly! A hack! A hacked Set<> implem.*

  - Triggered by calling any method on the students collection: .size(), .iterator(),…
  - Happens only the first time: data is then kept in memory
  - Only works transaction is still open

Lazy vs Eager Loading
Defaults

- Direct fields = eager
  - All columns in the entity's table
- @...ToOne = eager:
  - a) findById □ + JOIN Teacher
  - b) SELECT ... (JPQL) □ +1 SELECT ⚠

```
SELECT ... FROM COURSE ...
SELECT ... FROM TEACHER t WHERE t.COURSE_ID=?
```

can be avoided using LEFT JOIN FETCH course.teacher

- @...ToMany = lazy
  - Loaded on-demand,
    e.g. the first time you iterate over it
  - ? *Custom* Set/List implem: PersistentXxx

```
@Entity
public class Course {

  private String name;

  @ManyToOne
  private Teacher teacher;

  @OneToMany
  private Set<Student> students;

  ...
}
```

Lazy vs Eager Loading

Changing the Defaults

usually a BAD idea

- Direct fields = ~~eager~~ lazy
  - ? How: *Change bytecode of* Course.getName()
    - ⚠ requires @Entity bytecode enhancement
      `@EnableLoadTimeWeaving on @Configuration`
  - 💡 Defer loading of @Lob fields
- @...ToOne = ~~eager~~ lazy
  - ? How: *proxies the Teacher* class, so calling any method (except getId()) triggers its full load
  - 💡 Defer loading large entities (# fields, blobs)
- @...ToMany = ~~lazy~~ eager
  - 💡 If parent has no meaning w/o children
  - – Heavy: for each Course, **always** +1 query ⚠
  - – 👉 prefer LEFT JOIN FETCH on use-cases that later NEED the children

```
@Entity
public class Course {

   @Basic(fetch = LAZY)
   private String name;


   @ManyToOne(fetch = LAZY)
   private Teacher teacher;


   @OneToMany(fetch = EAGER)
   private Set<Student> students;

   ...
}
```

Changing Data

Changes to attached entities are automatically saved to DB!


ACME

How do you Update an Entity ?

Changes to attached entities are automatically saved to DB!

WHAAT?

```
course.setName("Java");
```

Then, later, automatically...

```
UPDATE COURSE SET NAME=? ... WHERE ID=?
```

(Just before the Transaction COMMIT)

When ?

The JPA Transaction

Includes:

Already loaded entities (Persistence Context)

```
em.persist(course);
```

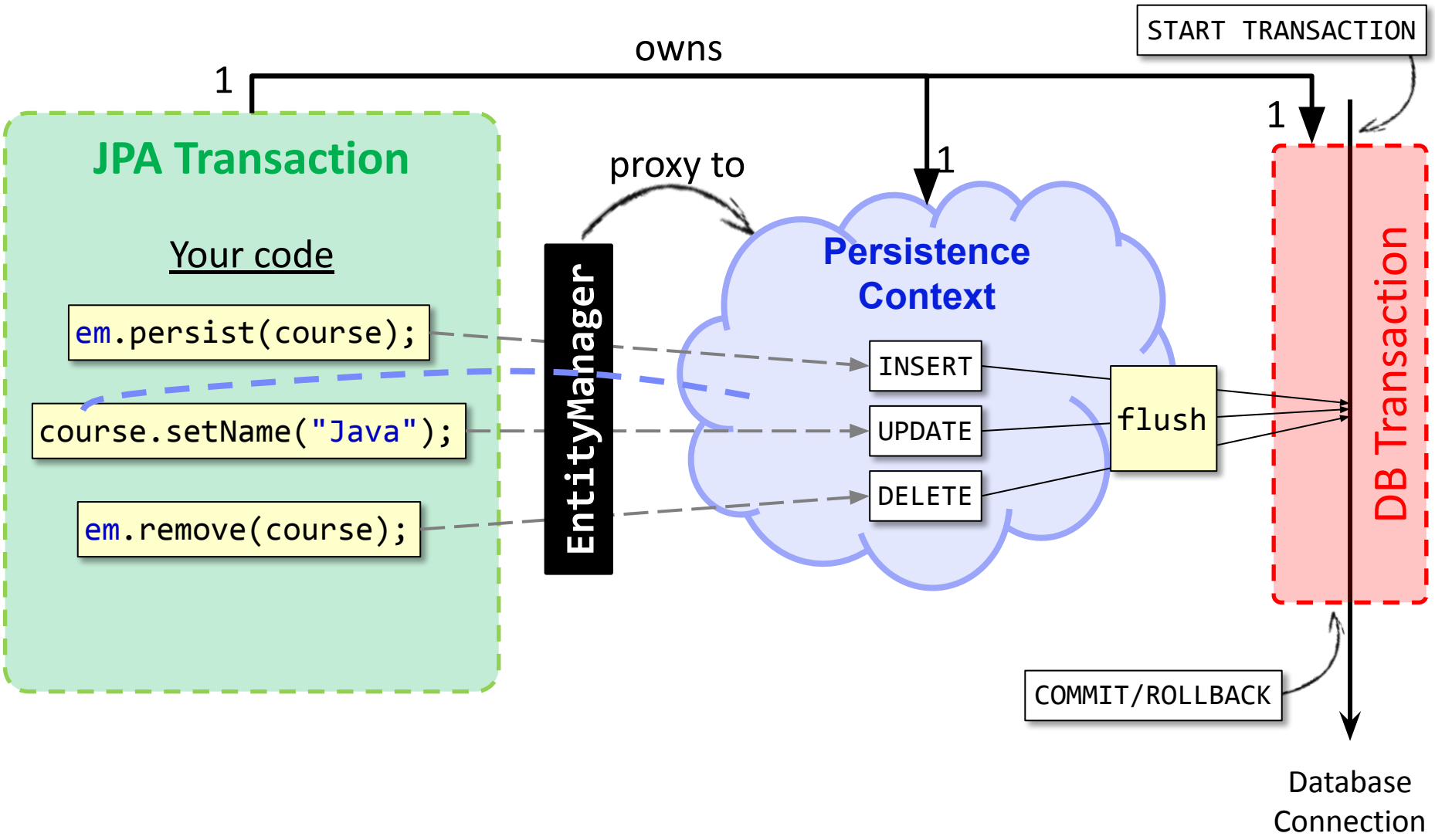
```
course.setName("Java");
```

```
em.remove(course);
```

Row locks

...

Persistence Context = read/write cache



Persistence Context is a read + write cache

Flushing

= Send all pending changes to DB

- Run the INSERTs, UPDATEs, DELETEs within the current DB Transaction
  - **Note**: later on, this Tx can be commited or rolledback

- Happens:
  - Automatically, right before the JPA Transaction Commit
  - Automatically, before running any query in the DB
    - Because the pending changes might influence the query results
  - `em.flush();  // code smell - you shouldn't need to do that !`

    (except when calling PL/SQL or in some dark corner cases)

At JPA Transaction Commit...

(eg end of @Transactional)

1. Flushes all pending changes to DB
2. COMMITs the underlying DB Transaction
3. Clears the Persistence Context

Controlling the JPA Transaction

- Programmatic — `entityManager.getTransaction()` (plain Java)
  - .begin()
  - .setRollbackOnly() □ can only ROLLBACK at the end
  - .commit() == end Tx (commit or rollback)
  - ⚠️ Prone to error in closing
- Declarative:
  - `@TransactionAttribute` (EJB)
  - `@Transactional, TransactionTemplate` (Spring)
  - Before entering the method □ start Tx
  - Thrown exception □ mark for rollback
  - After exit method □ end Tx (commit or rollback)

See more about transaction in my other webinar:

Transactions in Spring

propagation

rollback conditions

under-the-hood stuff

performance issues

best practices

When using JPA,

if you change the state of an @Entity instance,

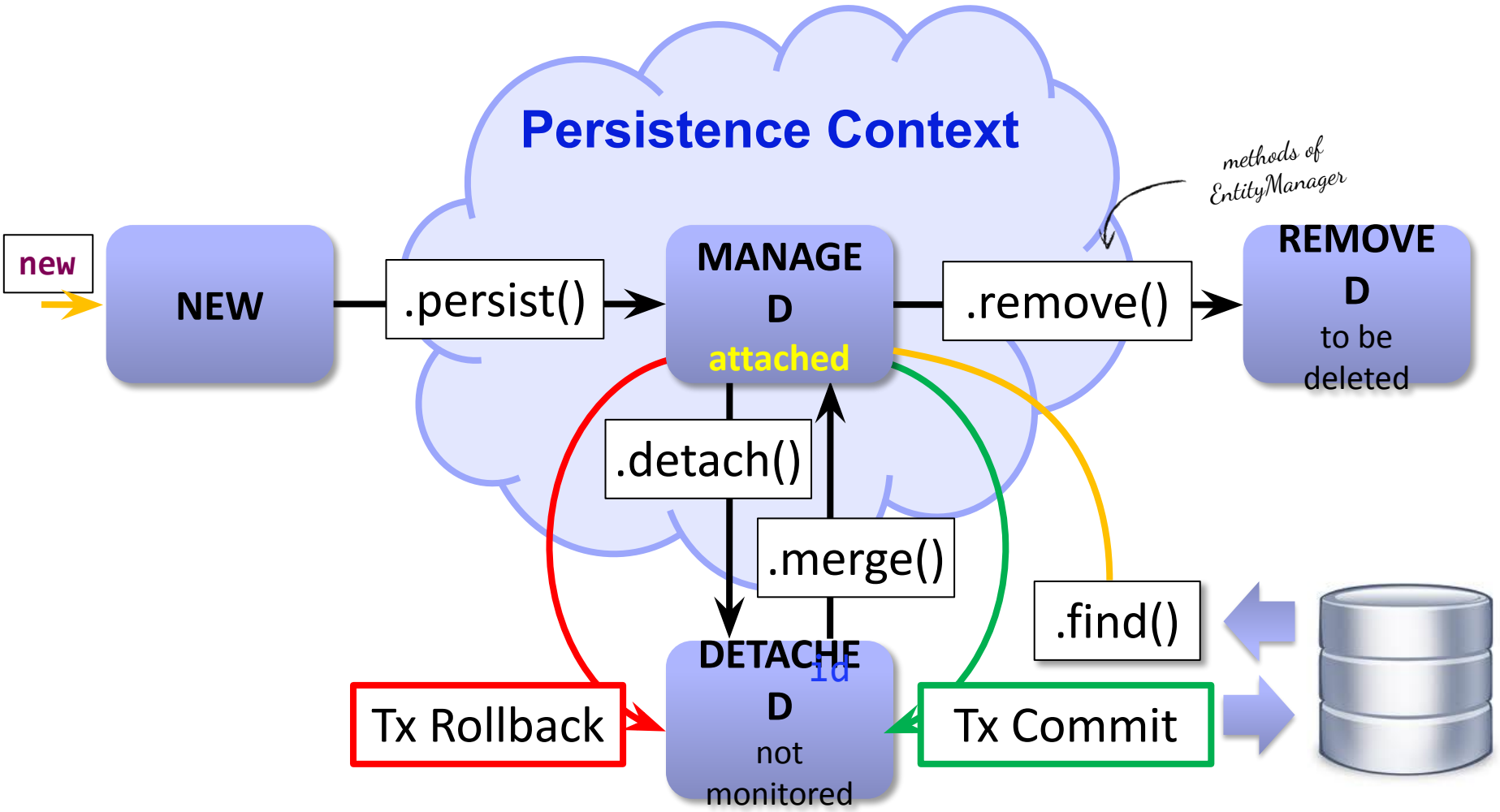
that change CAN get saved to DB

When it DOESN'T ?

Detached entities are NOT Monitored for Changes:

- Just-created instances
- After `em.detach(e)`
  - Or got from methods with `NOT_SUPPORTED` propagation (Tx ⏸)
- After `em.clear()`
  - Detaches all entities in Persistence Context ⚠️
- After Transaction end
  - Automatically does `entityManager.clear()`
- Entities attached in other outer Tx ☹️☹️
  - Best practice: don't pass entity instances to a `REQUIRES_NEW` method
- In transactions started with `readOnly=true`

Entity Lifecycle



merge

entityManager.merge(entity);

repo.save(entity); // with id≠null

`.merge(e)`

Overwrites an entity in DB

- **e** must be detached
  - Usually instantiated from outside data (UI, files, XML, JSON…)
- **e** must have all fields set ⚠️
  - @Id – the PK of row to overwrite
  - Any unmodifiable fields (eg createDate) must be copied in **e** DB before merging

□ Alternative: update an attached entity (later auto-flushed)

- What happens under the hood:
  1. JPA loads that Entity by id from DB (into **e'**)
  2. Copies all persistent fields from **e** □ **e'**
  3. Returns **e'**
     - Note: **e** is never attached ⚠️

Cascading `.merge()` on children

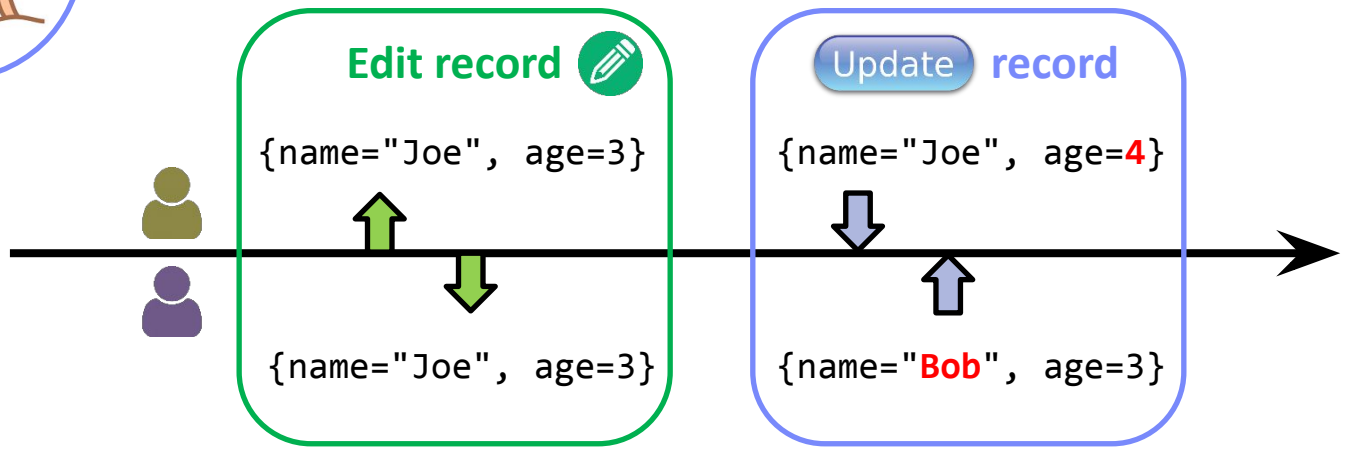
A child **c** will be:

- INSERT - if **c**.id = null
- UPDATE - if **c**.id ≠ null
- DELETE - IF **c** is NOT in the new children list AND orphanRemoval=true

Concurrency Control

Concurrency Control

No Protection



- What is the final age?
  - 3 => ⚠️ Changes from first user were silently lost (overwritten)
- Biz must accept this behavior
  - Explain them the risk

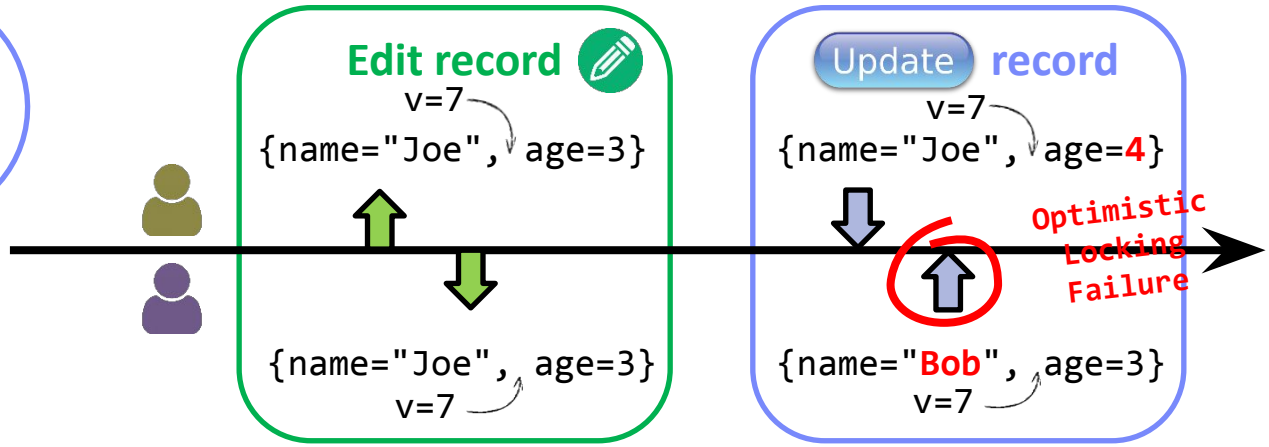
EASY WITH JPA

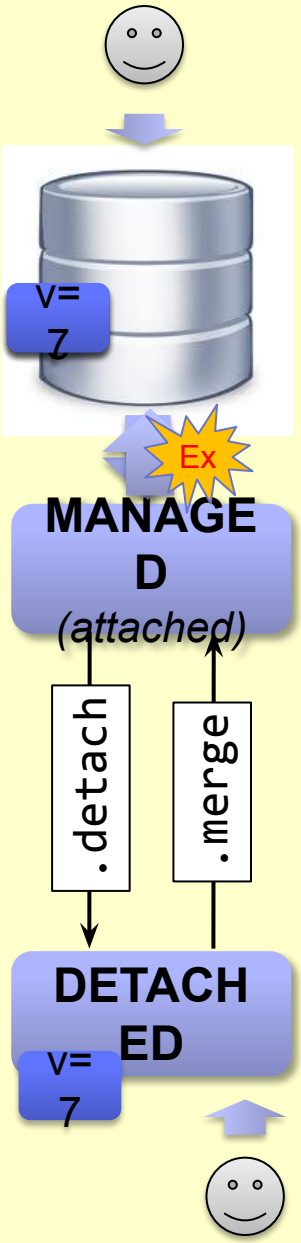
MOST USED

Hopes 🙏 concurrent edit won't happen

Concurrency Control

Optimistic Locking

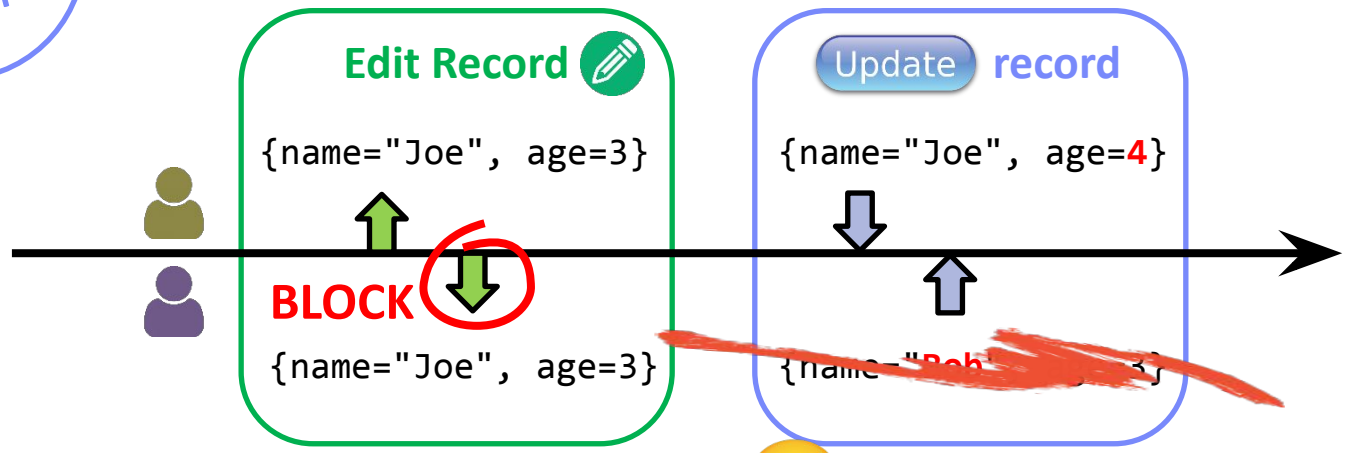




- Add a `@Version` field to your Entity
  - of type: long or Date
  - Send Receive the version number to your client
  - JPA auto-increments it at each update, after checking it's the same
- You `.merge(entityChangedByClient)`
  - Before saving it, JPA checks the version to be = the one in DB

MOST CONSIS TENT

Pessimistic Locking



- Manual implementation ☹
  - Using a custom "Lock" Entity+Table to save the user locks
  - On open edit screen: INSERT INTO LOCKS, on "Save" click -> check LOCKS + DELETE
    - Concurrency on that? **SELECT FOR UPDATE**: `em.lock(e, PESSIMISTIC_WRITE);`
- Decisions:
  - Lock auto-expiration: fixed timeout or session expiration?
  - "Your session will expire in 5 minutes" ? Wizz wizzair.com
  - 'force unlock' feature? Fallback to open as read-only?
  - …
    - JPA – Runtime

Entity Listeners

```
@PrePersist
@PreUpdate
public void automaticUpdateTrackingColumns() {
    lastModifiedDate = LocalDateTime.now();
    lastModifiedBy =
MyUtil.getUserOnCurrentThread();
}
```

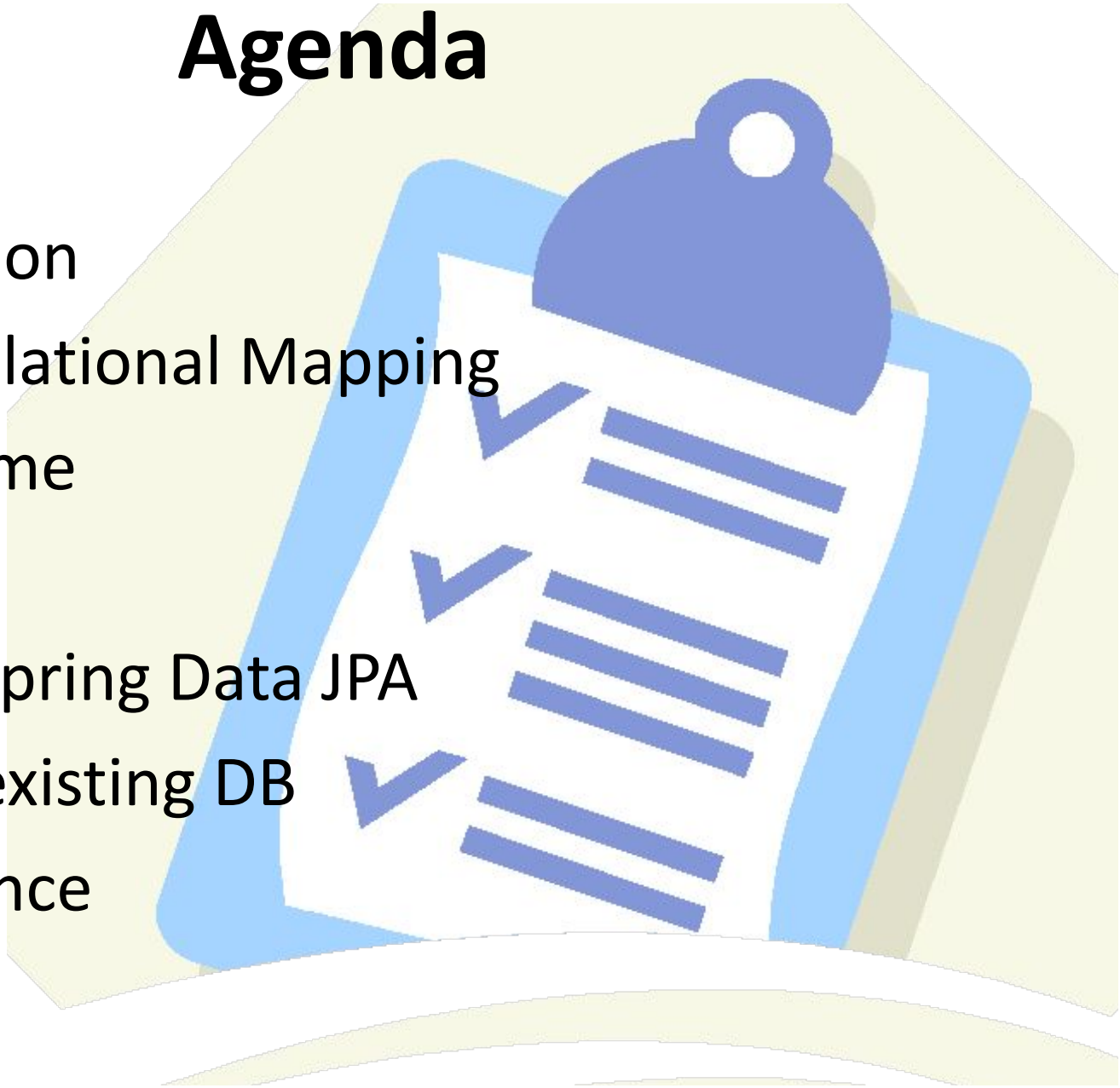
Also:
@PostLoad,…

- Inside the Entity or
- In a separate @EntityListener

  - Psst! Spring Data would help you with that !

KEEP CALM AND CODE

Agenda



- Introduction
- Object-Relational Mapping
- JPA Runtime
- **JPQL**
- Setup & Spring Data JPA
- ORM on existing DB
- Performance

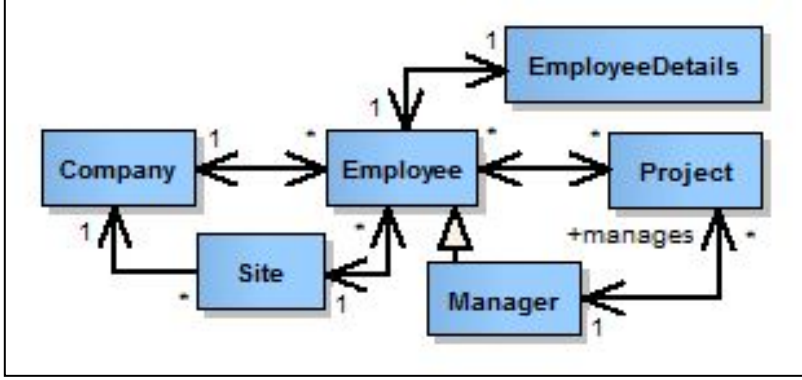
Java Persistence Query Language

= query language over entity model (not DB)

- Similar to SQL
  - Actually, it's translated to SQL and ran in DB

1. Inline: `em.createQuery("SELECT...");`
2. Declared `@NamedQuery` or Spring's `@Query`
   - Are automatically validated at application startup
   - Can use :named or positional (?1) parameters
3. Native SQL queries

JPQL in 1 Slide 😊



Extract `@Entity` (with all fields)

- `SELECT e FROM Manager e`
  `LEFT JOIN FETCH e.projects`
  `WHERE e.firstName LIKE '%J%'`
  `AND e.site.company.name = 'Kepler'`

Polymorphic queries

EAGER load list *

Using fields, not columns

"." traverses `@...ToOne` links

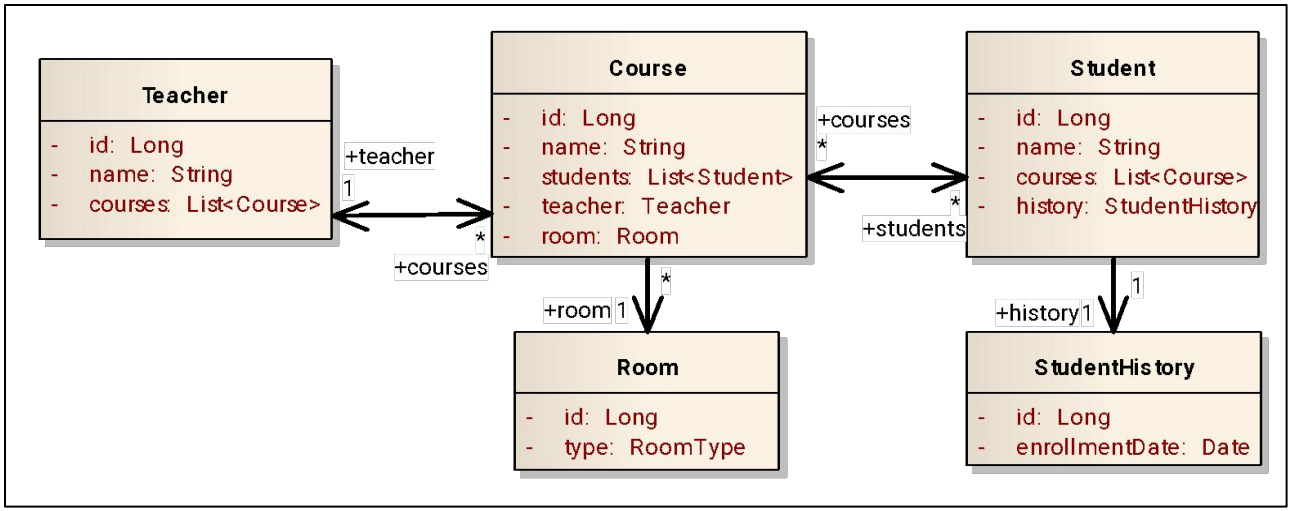
- `SELECT p.name FROM Project p`
  `INNER JOIN p.employees e`
  `WHERE e.name = 'John'`

"JOIN" traverses `@...ToMany` links

You don't need ON ... condition. (JPA KNOWS)

Students enrolled this year

```
SELECT s
FROM Student s
WHERE s.history.enrollmentDate
        >= '2013-01-01'
```



Courses held by prf. Mike loading students

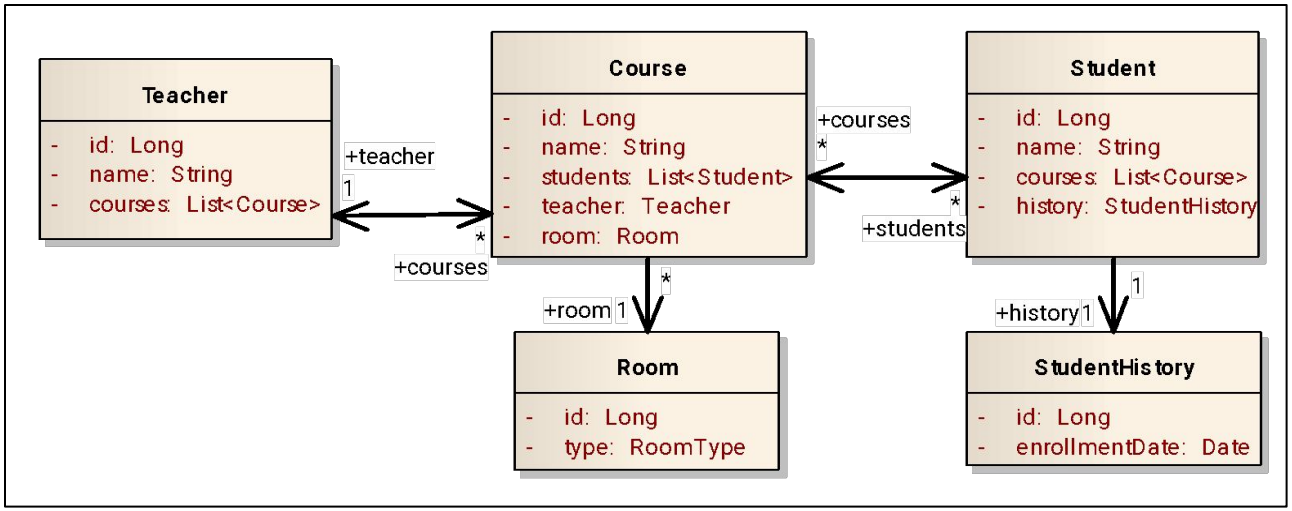
works even without implementing hashCode/equals() because in the same Tx, two entities with the same ID are ==

```
SELECT c
FROM Course c
LEFT JOIN FETCH c.students
WHERE c.teacher.name = 'Mike'
```

Returns the same course two times in a List ?!

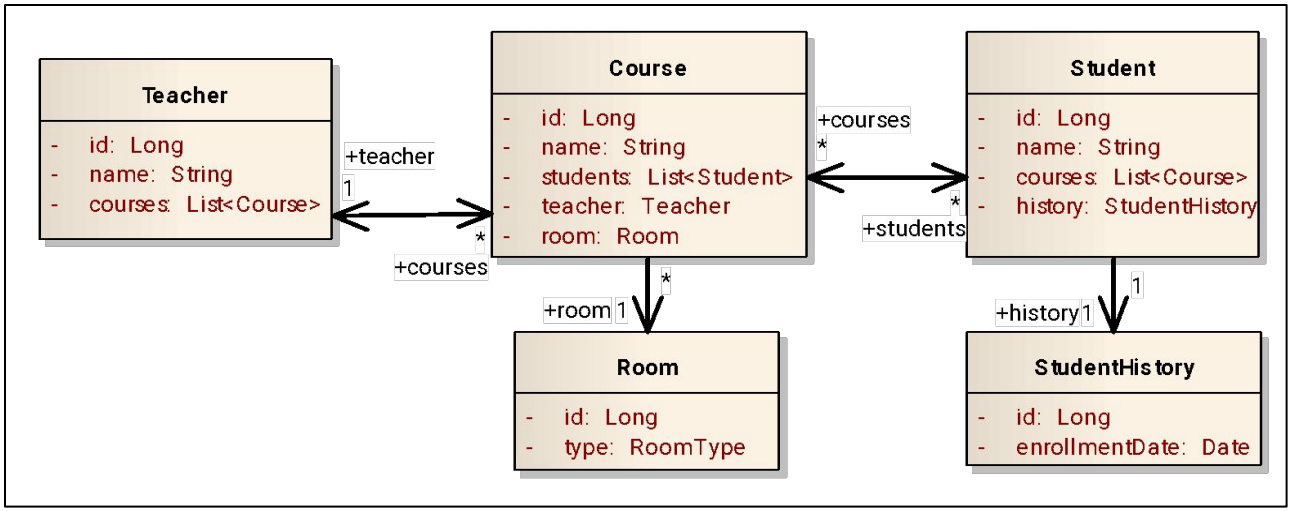
a) `Set<>`

b) `DISTINCT`



of courses prof. Mike has in an AULA

```
SELECT COUNT(c)
FROM Course c
WHERE c.teacher.name = 'Mike'
  AND c.room.type = 'AULA'
```



The courses student John has in an AULA

```
SELECT c
FROM Course c
INNER JOIN c.students s
WHERE s.name = 'John'
   AND c.room.type = 'AULA'
```

Duplicated course?

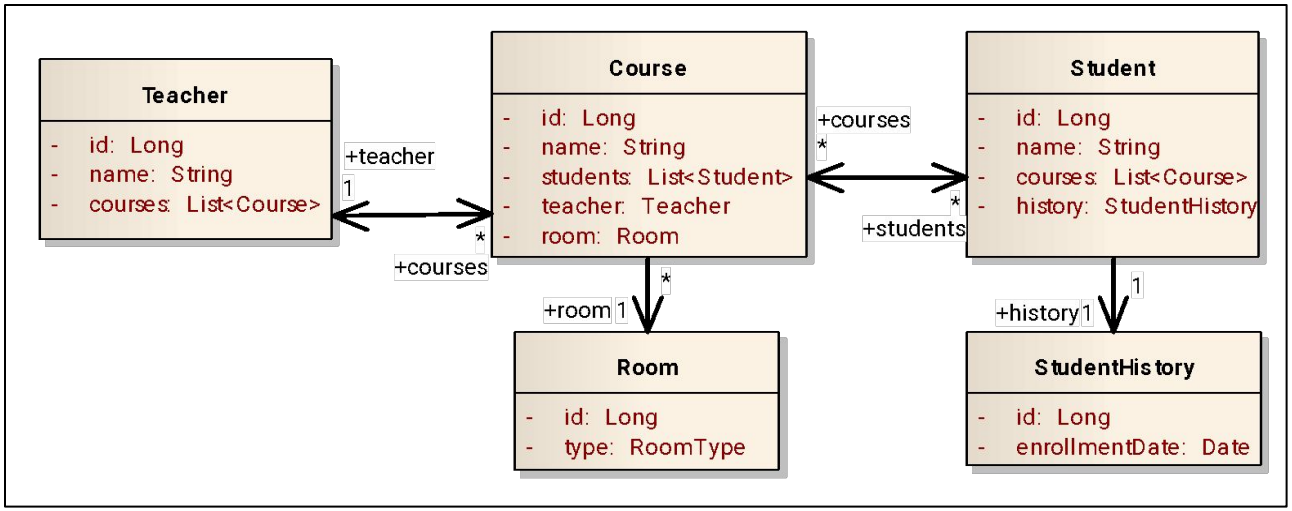
equals() ? ==

a) Set<>

b) DISTINCT

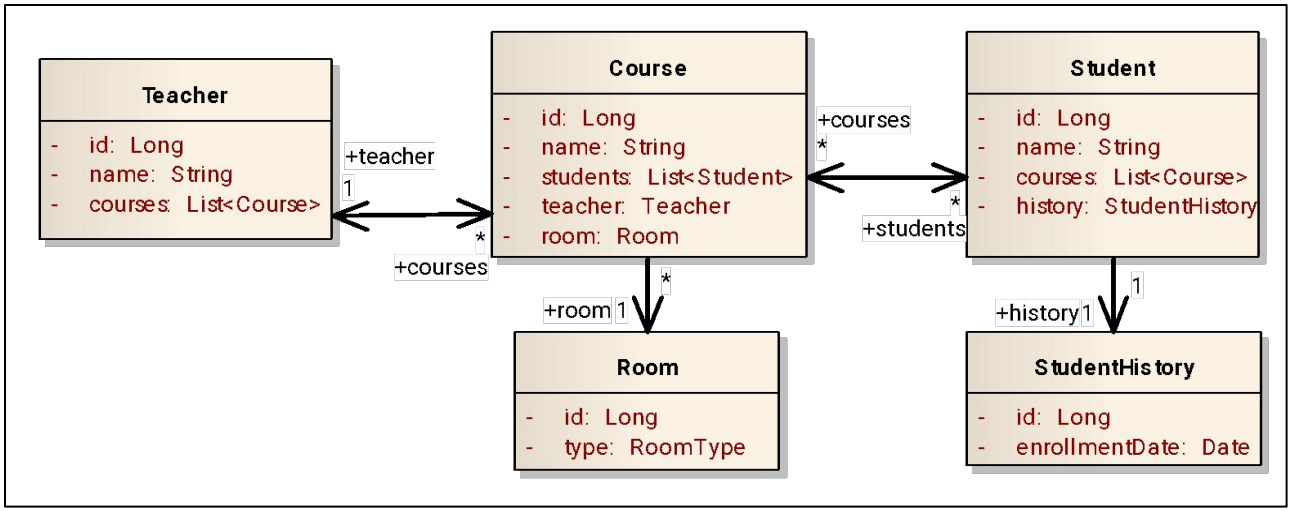
Performance?

CLOB?



Teachers of the last enrolled student

```
SELECT c.teacher
FROM Student s JOIN s.courses c
WHERE s.history.enrollmentDate =
    (SELECT max(ss.history.enrollmentDate)
     FROM Student ss)
```



(Student name – enrolled course name) pairs

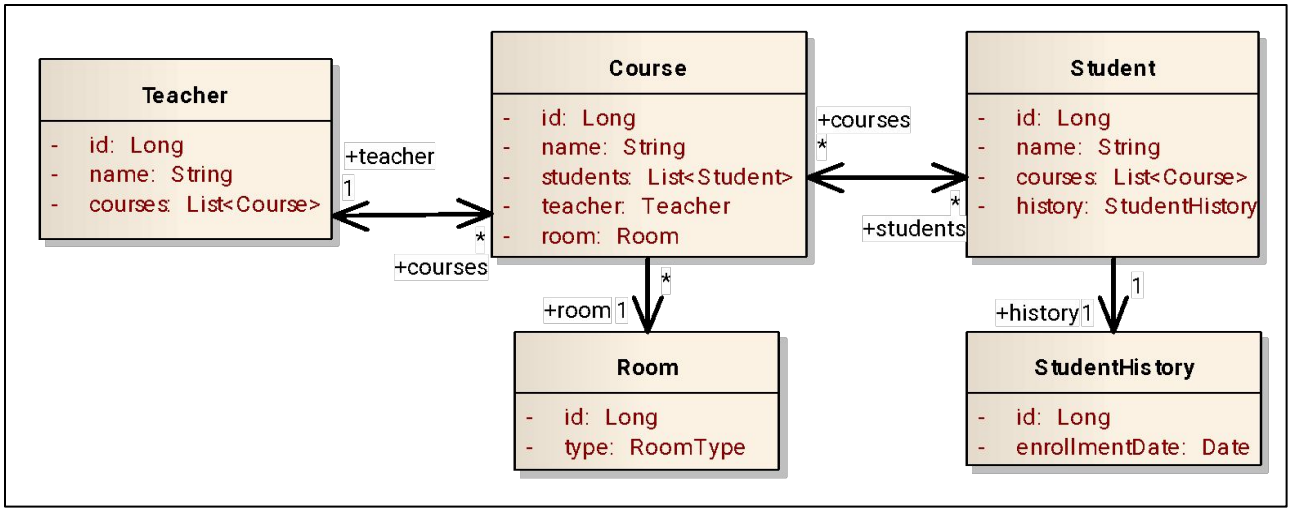
`List<Object[]>`

1. `SELECT s.name, c.name` or
2. `SELECT new com...StudentCourseDto(s.name, c.name)`

`List<StudentCourseDto>`

```
FROM Student s JOIN s.courses c
```

Lomb ok?



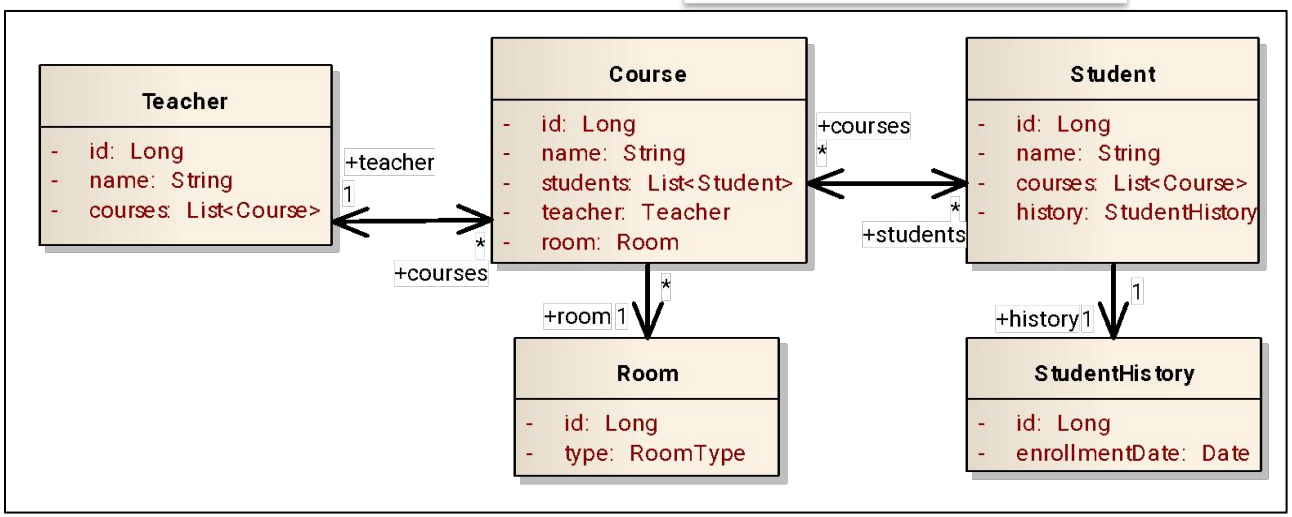
Courses with the same name as those attended by Stud X, but held by a different teacher

```
SELECT c2
FROM Student s JOIN s.courses sc,
   Course c2
WHERE sc.name = c2.name
AND c2.teacher.id != sc.teacher.id
AND s.id = :studentId
```

Cross Join

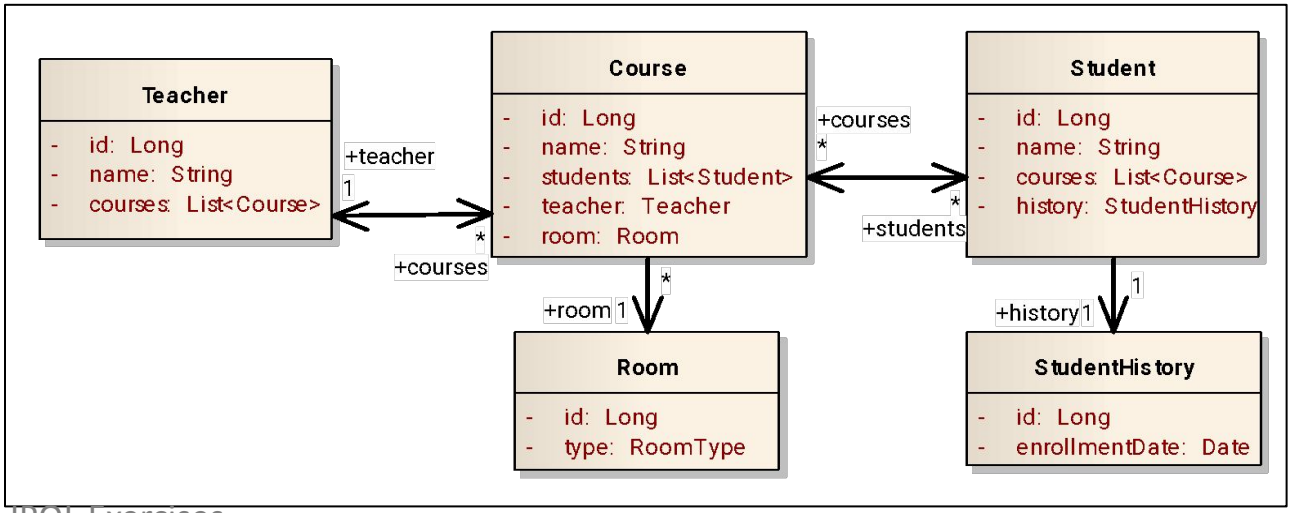
... ON sc.name = c2.name (Hibernate)

Named parameter



Courses with the same name as those attended by X, but held by a teacher not known by X

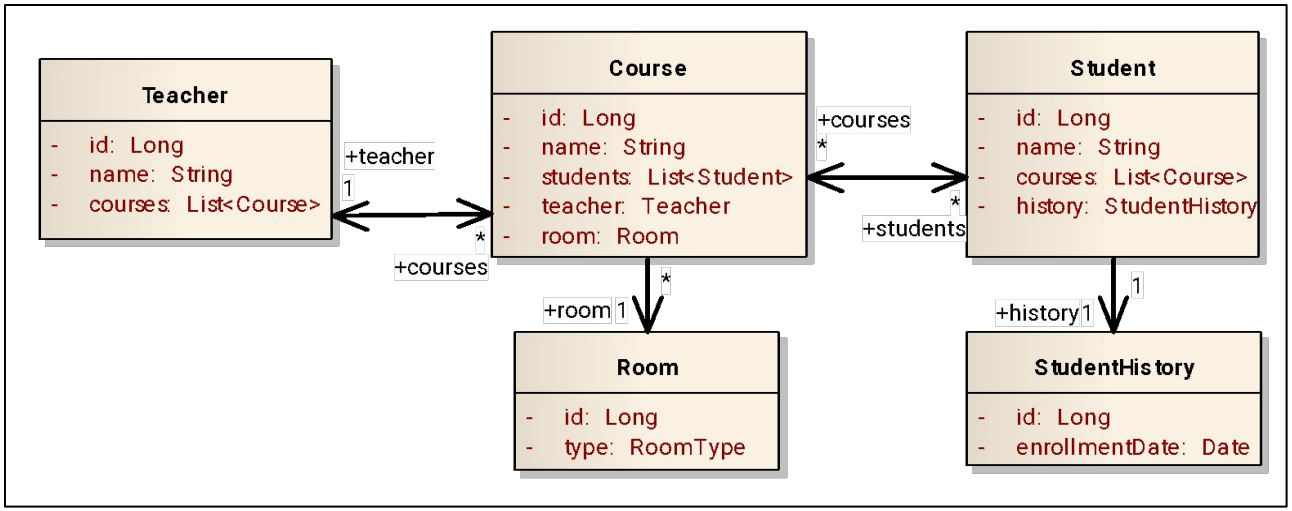
```
SELECT tc FROM
FROM Student s JOIN s.courses sc,
     Teacher t JOIN t.courses tc,
WHERE s.id = :studId
  AND sc.name = tc.name
  AND t.id NOT IN (SELECT ssc.teacher.id FROM Student ss
                   JOIN ss.courses ssc where ss.id =:studId)
```



Courses in an AULA that Student X does NOT attend

```
SELECT ca
FROM Student s,
   (SELECT c FROM Course c
     WHERE c.room.type = 'AULA') ca
WHERE s.id = :studentId
AND ...
```

JPQL Limitation:
No subqueries in FROM clauses



Courses in an AULA that Student X does NOT attend

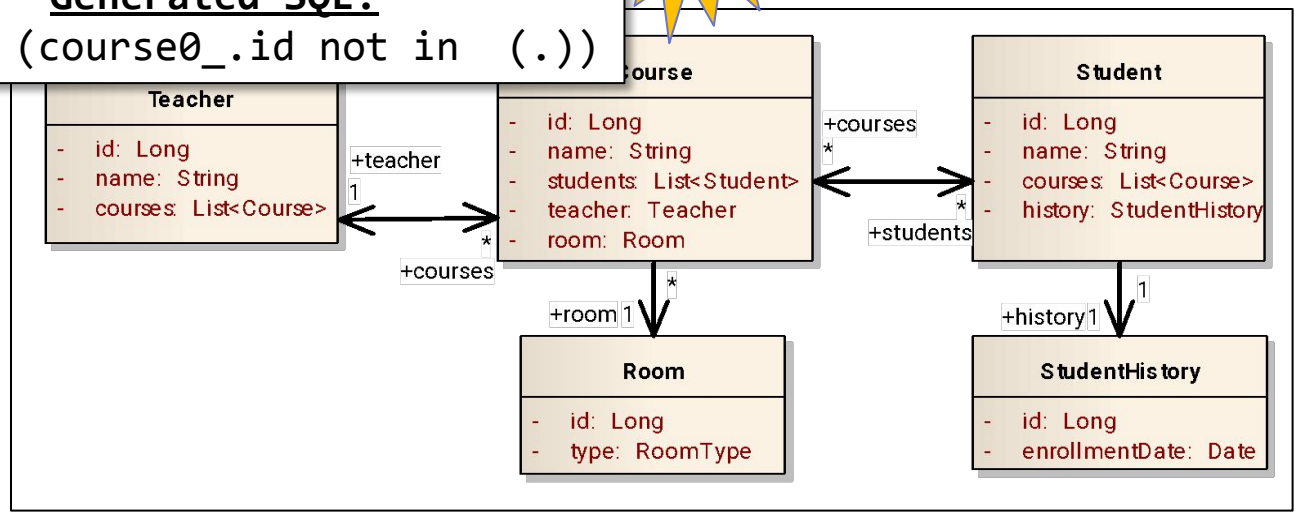
```
SELECT c
FROM Course c, Student s
WHERE s.id = :studentId
AND c.room.type = 'AULA'
AND c NOT IN s.courses
```

JPQL Limitation:
You CAN'T use the IN operator to test Entity child collections

Ex

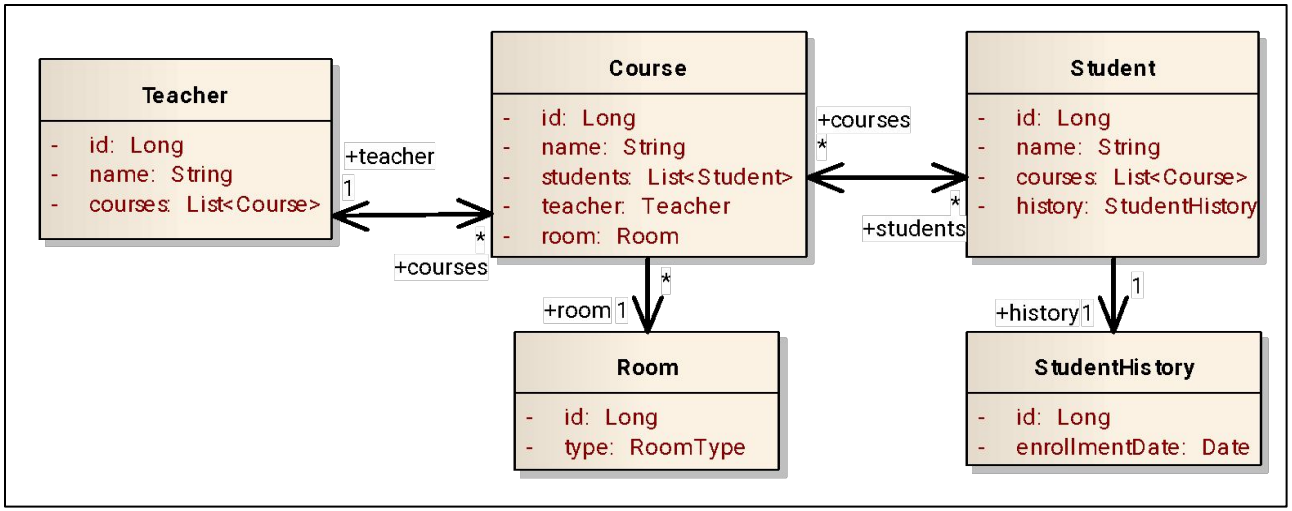
Generated SQL:

```
... AND (course0_.id not in  (.))
```



Courses in an AULA that Student X does NOT attend

```
SELECT c
FROM Course c, Student s
WHERE s.id = :studentId
AND c.room.type = 'AULA'
AND
```



No UNION in JPQL

```
X1 UNION X2
```

JPQL Limitation:
No UNION

- Can be worked around:

```
SELECT a FROM A a
WHERE a.ID IN (X1)
   OR a.ID IN (X2)
```

Search Query: Concatenation

```
public List<Course> search(CourseSearchCriteria criteria) {
    String jpql = "SELECT c FROM Course c WHERE 1=1 ";
    Map<String, Object> params = new HashMap<>();

    if (StringUtils.isNotBlank(criteria.namePart)) {
        jpql += " AND UPPER(c.name) LIKE UPPER('%' || :name || '%') ";
        params.put("name", criteria.namePart);
    }

    if (criteria.teacherId != null) {
        jpql += " AND c.teacher.id = :teacherId";
        params.put("teacherId", criteria.teacherId);
    }

    if (criteria.studentId != null) {
        jpql += " AND c.id IN (SELECT cc.id FROM Student s JOIN s.courses cc WHERE
s.id=:sId)";
        params.put("sId", criteria.studentId);
    }

    TypedQuery<Course> query = em.createQuery(jpql, Course.class);
    for (String key : params.keySet()) {
        query.setParameter(key, params.get(key));
    }
    return query.getResultList();
}
```

Search Query: Criteria API

```
public List<Course> search(CourseSearchCriteria searchCriteria) {
    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<Course> q = cb.createQuery(Course.class);
    Root<Course> r = q.from(Course.class);
    Map<String, Object> params = new HashMap<>();
    List<Predicate> predicates = new ArrayList<>();

    if (StringUtils.isNotBlank(searchCriteria.namePart)) {
        predicates.add(cb.like(
                cb.upper(r.get("name")), cb.upper(cb.parameter(String.class,
"nameParam"))));
        params.put("nameParam", "%" +searchCriteria.namePart + "%");
    }

    if (searchCriteria.teacherId != null) {
        predicates.add(cb.equal(
                r.get("teacher").get("id"), cb.parameter(Integer.class, "teacherId")));
        params.put("teacherId", searchCriteria.teacherId);
    }

    q.select(r).where(predicates.toArray(new Predicate[0]));
    TypedQuery<Course> query = em.createQuery(q);
    for (String key : params.keySet()) {
        query.setParameter(key, params.get(key));
    }
    return query.getResultList();
}
```

Search Query: Criteria + Metamodel

```
public List<Course> search(CourseSearchCriteria searchCriteria) {
    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<Course> q = cb.createQuery(Course.class);
    Root<Course> r = q.from(Course.class);
    Map<String, Object> params = new HashMap<>();
    List<Predicate> predicates = new ArrayList<>();

    if (StringUtils.isNotBlank(searchCriteria.namePart)) {
        predicates.add(cb.like(
                cb.upper(r.get(Course_.name)), cb.upper(cb.parameter(String.class,
"nameParam"))));
        params.put("nameParam", "%" +searchCriteria.namePart + "%");
    }

    if (searchCriteria.teacherId != null) {
        predicates.add(cb.equal(
                r.get(Course_.teacher).get(Teacher_.id),  cb.parameter(Integer.class,
"teacherId")));
        params.put("teacherId", searchCriteria.teacherId);
    }

    q.select(r).where(predicates.toArray(new Predicate[0]));
    TypedQuery<Course> query = em.createQuery(q);
    for (String key : params.keySet()) {
        query.setParameter(key, params.get(key));
    }
    return query.getResultList();
```

Generated Classes

Specifications

composable, manipulated in business logic

```java
public class TeacherSpecifications {
    public static Specification<Teacher> hasNameLike(String name) {
        return (root, query, cb) ->
            cb.like(cb.upper(root.get(Teacher_.name)), pattern: "%" + name.toUpperCase() + "%");
    }

    public static Specification<Teacher> hasGrade(Grade grade) {
        return (root, query, cb) ->
            cb.lessThan(root.get(Teacher_.grade), grade);
    }
}
```

```java
public class BusinessClass {
    private final TeacherRepo teacherRepo;

    public void method() {
        Specification<Teacher> spec = TeacherSpecifications.hasGrade(Grade.CONF)
            .or(TeacherSpecifications.hasNameLike("a"));
        List<Teacher> teachers = teacherRepo.findAll(spec);
    }
}
```

Query DSL

More fluent than Criteria API

```
public List<Teacher> searchQueryDSL(TeacherSearchCriteria searchCriteria) {
    JPAQuery<?> query = new JPAQuery<Void>(em);
    BooleanExpression pred = Expressions.TRUE;

    if (searchCriteria.grade != null) {
        pred = pred.and(teacher.grade.eq(searchCriteria.grade));
    }
    if (searchCriteria.name != null) {
        pred = pred.and(teacher.name.upper()
            .like( str: "%" + searchCriteria.name.toUpperCase() + "%"));
    }

    return query.select(teacher)
        .from(teacher)
        .where(pred)
        .fetchAll()
        .fetch();
}
```

https://spring.io/blog/2011/04/26/advanced-spring-data-jpa-specifications-and-querydsl/

Bulk Update JQPLs

- `UPDATE Employee e SET e.sal = e.sal*2 WHERE ...`
- `DELETE FROM Employee e WHERE ...`

- JPQL is converted to SQL and ran on the DB
- When executed
  - Changes are performed in DB
  - … but the Persistence Context gets out of sync
    (e.g. still em.find(id) the deleted entities)
- Results may interfere with the cached entities
  - Best Practice: run the BULK JPQL right before Tx end or .clear the em after

Hiding useless methods in JpaRepository

- Why?
- It feels wrong exposing so many methods freely to my app

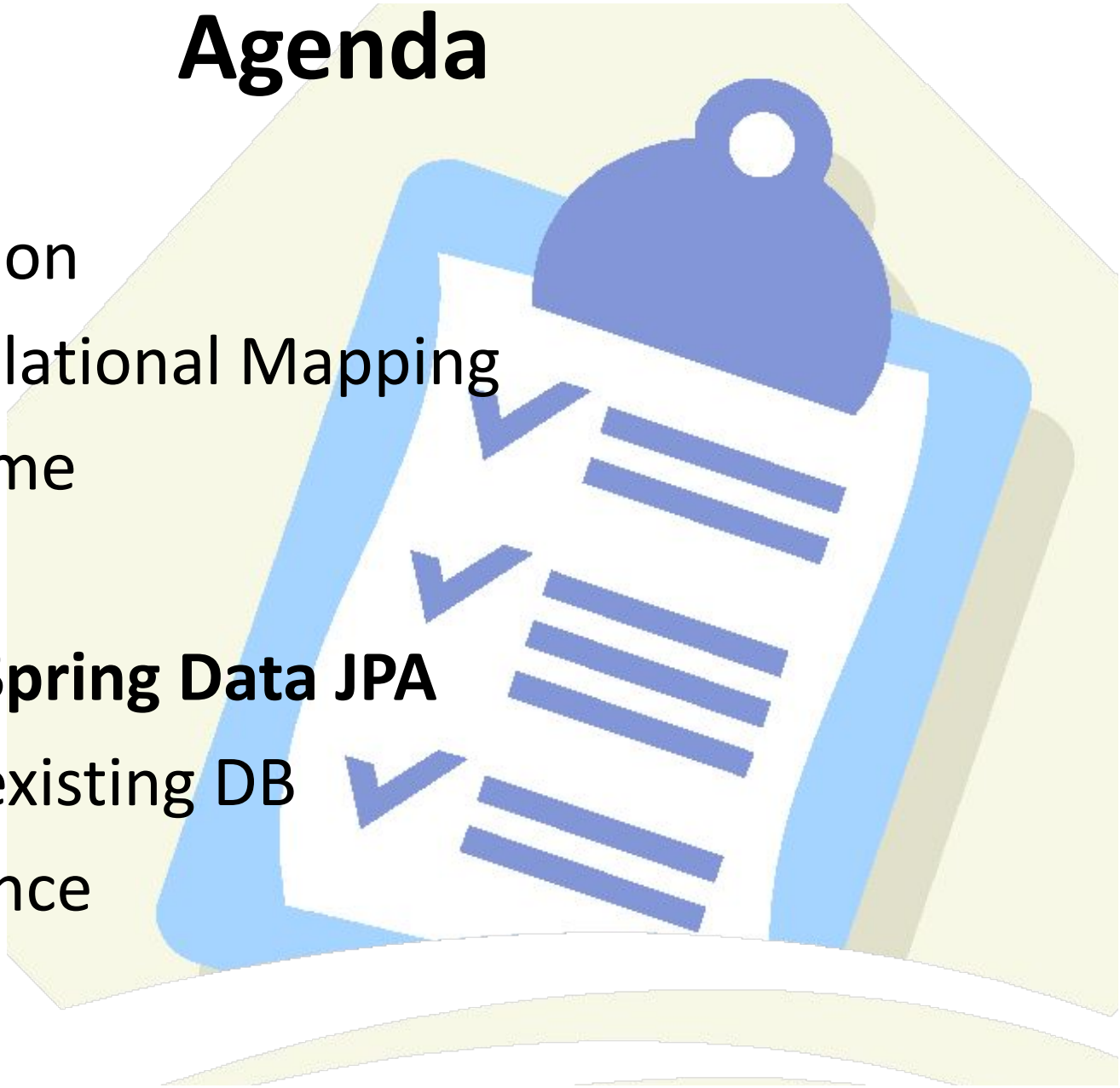
```
public interface TeacherRepo extends Repository<Teacher, Long> {

   Teacher save(Teacher teacher);

   Teacher findById(Long id);
}
```

everything works fine if the method names
match those 'standard' in the JpaRepository

https://www.amitph.com/spring-data-repository-hide-methods

KEEP CALM AND CODE

Agenda



- Introduction
- Object-Relational Mapping
- JPA Runtime
- JPQL
- **Setup & Spring Data JPA**
- ORM on existing DB
- Performance

Setting up JPA

Ingredients:

- DataSource
  - Taken from JNDI, Self-managed pool, Single Connection, In-memory DB
- Transaction Manager
  - JPA, Spring or EJB
- @Entity classes
  - ~~Listed each~~ or auto-detected in given package 
- Persistence Provider Config
  - hibernate.hbm2ddl.auto, hibernate.show_sql, …

JPA in Spring Boot

- DataSource
  - If HSQLDB in classpath -> in memory DB
  - Explicit DataSource @Bean ?
  - Explicit spring.datasource.url, … ?
  - Explicit spring.datasource.jndi-name= ?
- Transaction Manager
  - @Transactional all the way
- @Entity classes
  - Auto-detected in any subpackage under the @SpringBootApplication
- Persistence Provider Config
  - `spring.jpa.hibernate.*` entries in application.properties

https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-sql.html

JPA in Spring XML

```
<jee:jndi-lookup id="dataSource" expected-type="javax.sql.DataSource"
  jndi-name="jdbc/regliss"/>

<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="jpaVendorAdapter">
    <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
      <property name="database" value="ORACLE" />
    </bean>
  </property>
  <property name="jpaProperties">
    <props>
      <prop key="hibernate.use_sql_comments">true</prop>
      <prop key="hibernate.format_sql">false</prop>
      <prop key="hibernate.id.new_generator_mappings">true</prop>
      <prop key="hibernate.hbm2ddl.auto">validate</prop>
      <prop key="hibernate.jdbc.fetch_size">100</prop>
      <prop key="hibernate.jdbc.batch_size">500</prop>
      <prop key="databasePlatform">org.hibernate.dialect.Oracle10gDialect</prop>
    </props>
  </property>
  <property name="packagesToScan" value="com.bnpp.regliss.entity" />
</bean>
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-sql.html

Other Setups ?

- Standalone JSE + persistence.xml
- Standalone JSE + hibernate.cfg.xml
- EJB 3.x + persistence.xml
- EJB 3.x + hibernate.cfg.xml
- Spring XML + persistence.xml
- Spring XML + hibernate.cfg.xml
- Spring @Configuration + Hibernate Adapter
- +/- orm.xml
- etc.....

A la carte ☺️

Spring Data JPA

Queries from Method Names

Spring Data JPA generates the appropriate query

```
Customer findByCnp(String cnp);

List<Customer> findByActiveTrue();

Customer getByNameAndActiveTrue(String name);

Customer findFirstByOrderByNameAsc();

List<Customer> findByAddressCity(String city);
```

```
SELECT c FROM Customer c
WHERE c.address.city=?
```

Explicit @Query

```
@Query("SELECT c FROM Customer c WHERE c.id!=?2 AND c.phone=?1")
List<Customer> findOthersWithSamePhone(String phone, long id);

@Query(native=true, value="SELECT ... FROM (SELECT ...) ...")
List<Object[]> runNativeQuery();

@Query("SELECT c.id, c.name FROM Customer c")
List<Object[]> findAllCustomerIdAndName();

@Query("SELECT new com..MyDto(c.id, c.name) FROM Customer c")
List<MyDto> findAllCustomerIdAndNameAsDtos();

@Query("SELECT c FROM Customer c LEFT JOIN FETCH c.orders")
List<Customer> getCustomersFetchOrders();
```

Duplicate rows ?!!!?!#@# Argh!!

@NonNullApi

Return Values

- Must be consumed in the same Tx
- Don't forget to .close() it!!

```
Optional<Customer> findByCNP(String cnp);

Stream<Customer> findByActiveTrue();

Page<Customer> findByActiveTrue(Pageable pageable);
```

.getTotalPages()
.getContent():List
.getSort() …

.sort
.pageNumber
.pageSize

Spring+Hibernate can fill automatically the Create/Update date + user (from Spring Security)

@LastModifiedBy
@LastModifiedDate

https://www.baeldung.com/database-auditing-jpa

Bulk Update/Delete

(AKA DMLs)

```
@Modifying(
    flushAutomatically=true, // em.flush() before
    clearAutomatically=true) // em.clear() after
@Query("DELETE FROM Record r WHERE r.version.id = ?1")
void clearVersion(Long versionId);
```

Native queries

```
[@Modifying]
@Query(nativeQuery = true,
  value="DELETE FROM VERSION_RECORD WHERE RECORD_ID in ?1 AND VERSION_ID =
?2")
void removeFromVersion(List<Long> recordIds, Long versionId);
```

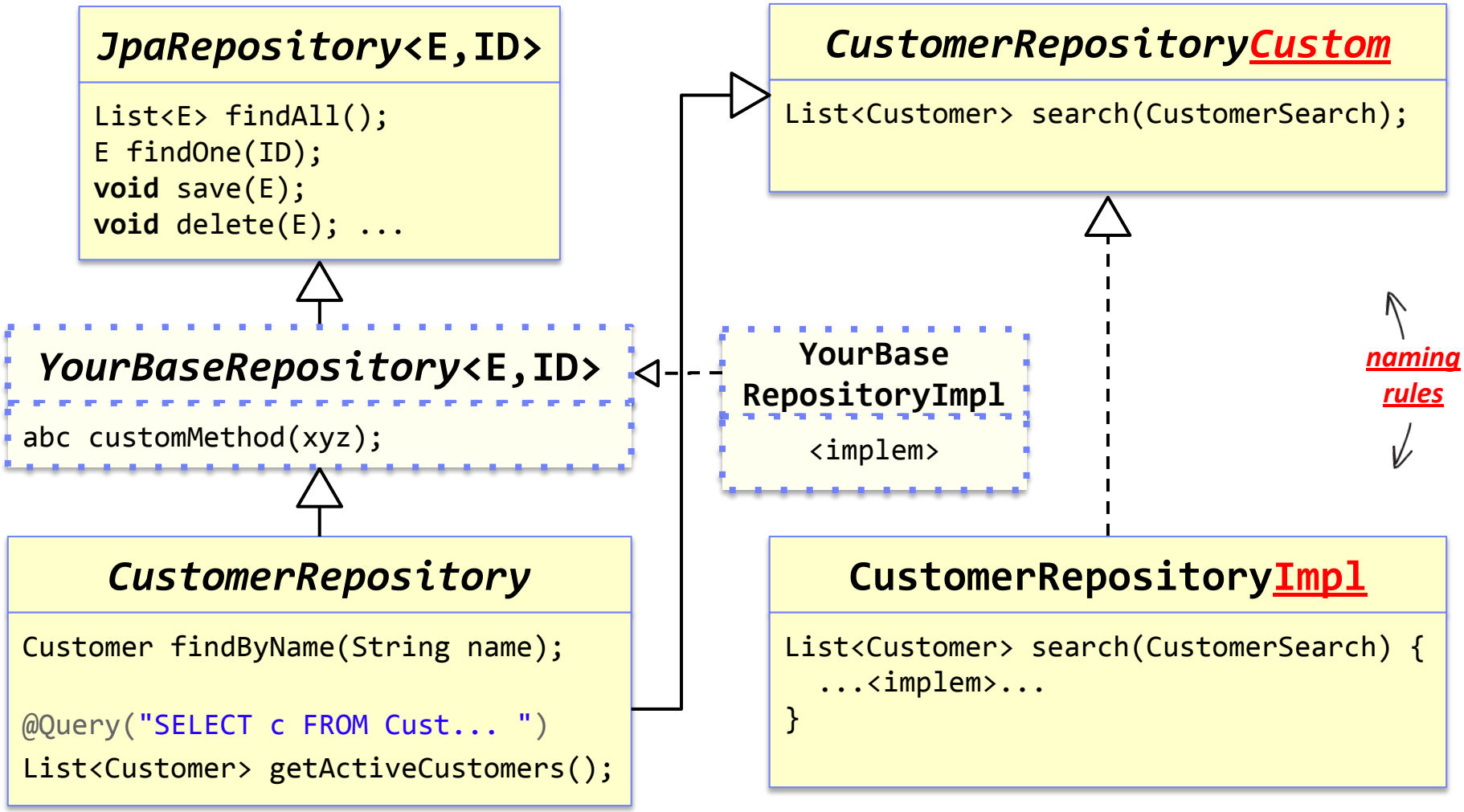
Call Stored Procedures

```
CREATE procedure plus1inout (IN arg int, OUT res int)
BEGIN ATOMIC
 set res = arg + 1;
END
```

With Spring Data JPA:

```
@Procedure(procedureName = "plus1inout", outputParameterName="res")
Integer procedure_test(@Param("arg") Integer arg);
```

Architecture



KEEP CALM AND CODE

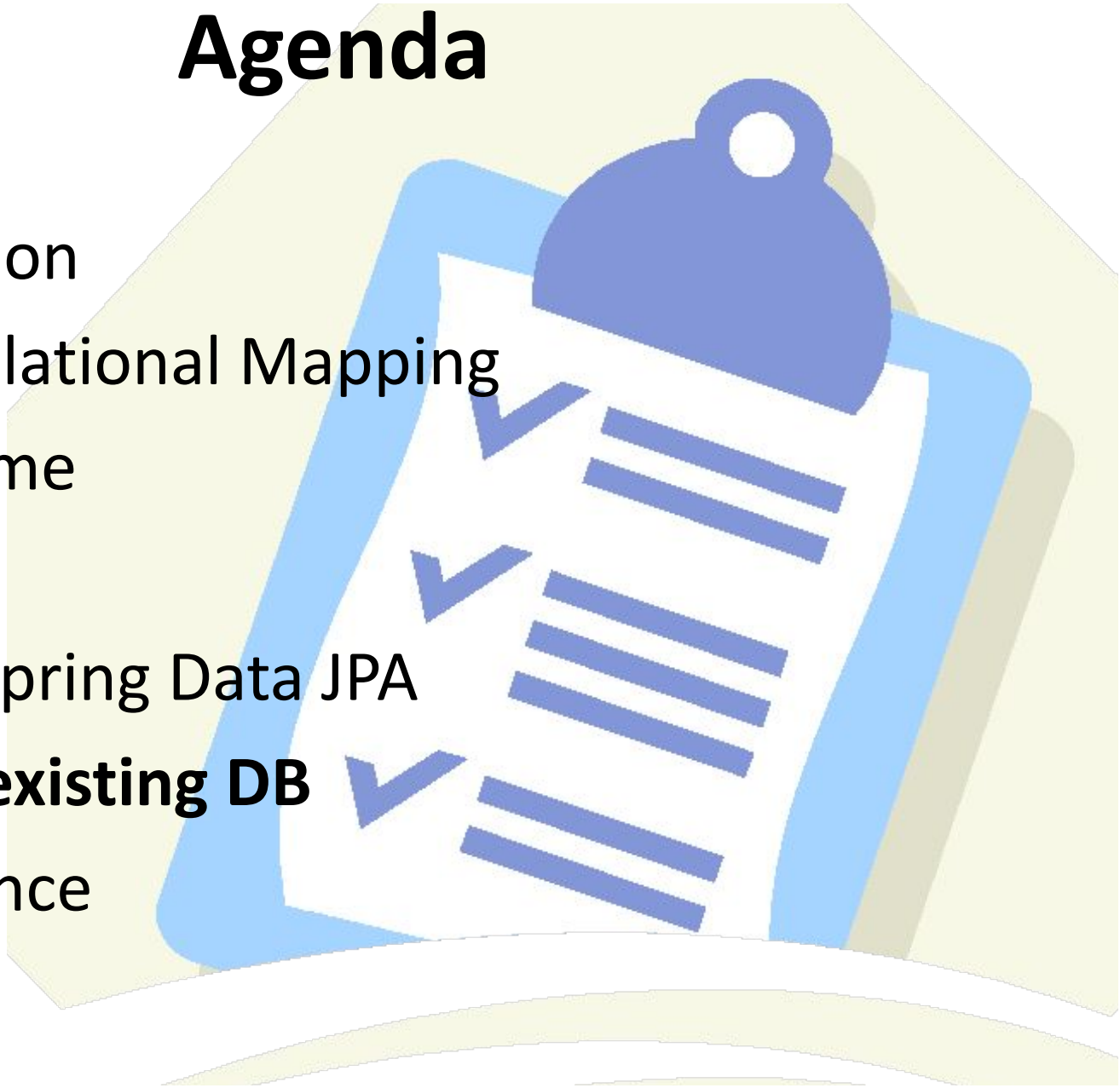
```
interface NamesOnly {

  String getFirstname();
  String getLastname();
}
```

More ?!

- Fetch Graph/EntityGraphs?
- Closed Projections
- Open Projections
- No more in/out null

```
interface NamesOnly {

  @Value("#{target.firstname + ' ' + target.lastname}")
  String getFullName();
  …
}
```

Agenda



- Introduction
- Object-Relational Mapping
- JPA Runtime
- JPQL
- Setup & Spring Data JPA
- **ORM on existing DB**
- Performance

Composite PK

```
public class PoliceFilePK
    implements Serializable
{
    @EmbeddedId(STRING)
    private FileType type;
    private Long id;
    // getters
    // hashCode, equals
}
```

```
@Entity
public class PoliceFile {
    @EmbeddedId
    private PoliceFilePK
id;
    ...
}
```


```
@Entity
public class Investigation
{
    ...
    @ManyToOne
    private PoliceFile
file;
}
```

```
PoliceFile file = em.find(
    PoliceFile.class,
    new PoliceFilePK(CONFIDENTIAL,
13L));
```

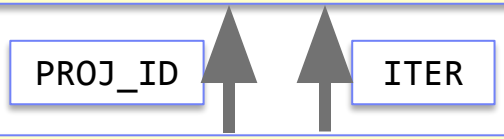
Composite PK

Parent + Child Id

```java
public class IterationPK
    implements Serializable
{
    private long project;
    private int iteration;
    // getters, hash,
equals
}
```

```java
@Entity
public class Project {
    @Id
    @GeneratedValue
    private Long id;
    ...
}
```

```java
@Entity
@IdClass(IterationPK.class)
public class Iteration {
    @Id
    private Integer
iteration;
    @Id
    @ManyToOne
    private Project project;
}
```




```java
@Entity
public class Task {
    @ManyToOne
    private Iteration
iteration;
}
```

@Column

```
@Column(
    name="CNP",
    length=13,
    unique=true,
    nullable=false,
    columnDefinition="VARCHAR2(13 BYTE)",
    insertable=false,
    updatable=false
)
```

VARCHAR2(13)

UNIQU E

NOT NULL

Don't use the value of this field for INSERT

...nor for UPDATE

Maybe another mapping of this same column is the one that actually updates it.

Entity Validation

- NOT NULL column
  - Opaque/late errors, unrecoverable
  - When DB is not safe: data imports/manual edits
- @NotNull (javax.validation)
  - Expressive: @Length, @Min, @Pattern
  - Extensible: @Constraint
- XyzValidator class - `if`
  - 100% control: truncate & warn, …
  - Can validate complex constraints in DB/external systems

- Call from @Entity constructor?
  - Trouble++ for Unit Tests

@Table

```
@Entity
@Table(
    name="MY_TABLE",
    uniqueConstraints = {
        @UniqueConstraint(columnNames={"A",
"B"}),
        …
    },
    indexes = {
        @Index(columnList="A, B"),
        …
    }
)
```

UNIQUE (A,B)

Change the Column Name

- Of inherited fields, or
- @Embedded fields:

```
@Embeddable
public class LabelVO {
    private String en;
    private String fr;
    …
}
```

```
@Entity
public class Project {
    …
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="en",
column=@Column(name="DESC_EN")),
        @AttributeOverride(name="fr",
column=@Column(name="DESC_FR"))
    })
    private LabelVO description;
}
```

DESC_EN instead of EN

Can you map an @Entity to a VIEW?

Yes!

It's just like mapping to a table

You can even make the view updatable!

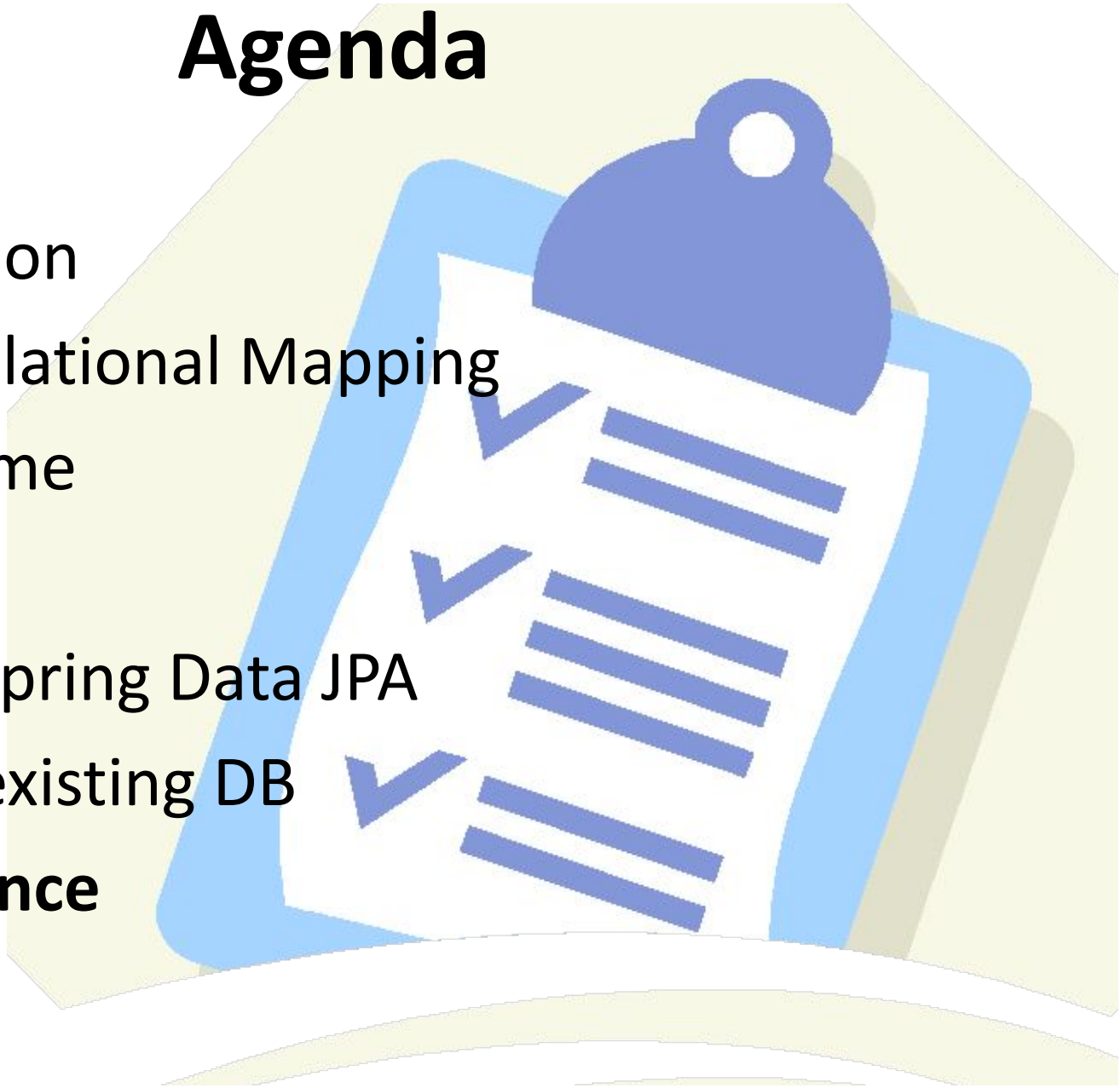
Tables/Views with no PK

- Every @Entity requires an @Id

- You can always invent an Unique Key:
  1. A natural key of 2-3 columns ?
  2. Worst case: PK={all columns}

- Composite PK in JPA:
  - @EmbeddedId
  - @IdClass

Agenda



- Introduction
- Object-Relational Mapping
- JPA Runtime
- JPQL
- Setup & Spring Data JPA
- ORM on existing DB
- **Performance**

Workshop:

JPA/Hibernate Performance Tuning

by Victor Rentea

JPA dream 

Make developers *feel* their persistent objects never leave memory (~ they are kept in a map)

In front of performance issues,
developers need to wake up from that dream.

(brutally from a deep sleep)

Myth or Reality ?

JPA is Slow

JPA Goal: **simple, maintainable code**

The truth is:

If used carelessly, JPA degrades performance!

But you <u>can</u> write high-Performance apps with JPA!

Performance

Performance ?

Measure, Don't Guess®

Premature optimization is the root of all evil

– Donald Knuth

Typical Symptoms

- Running the same simple query 1000 times
- Hundreds of useless columns or JOINs in SQLs
- Slow search
- Slow insert of bulk data
- Out of Memory at large datasets
- Awkward DB and/or Java Model

High-Performance Fetching

Fetching Data

- **Read-only**, for search or export *QUERY*
  - Select in DTOs
  - Lazy-load and N+1 Queries
  - Pagination
  - @Entity on View / @Subselect
  - Dynamic queries
  - Streaming from DB
- **Full Entities** to execute *COMMAND*
  - Lazy/Eager fetch
  - Replace @ManyToOne with numeric FK
  - Fetch Graphs
  - @Cached

High Performance JPA

You need:

- 📃 SQL Log
- 🧠 Brains
- 🏋️‍♂️ Expected volumes
- 👴 Patience to measure

https://vladmihalcea.com/14-high-performance-java-persistence-tips/

🎥 Ten SQL Tricks that You Didn't Think Were Possible (Lukas Eder)

SQL Logging 

- Ask Hibernate:
  - Verbose output of params
  - Hibernate Statistics :

```
logging.level.org.hibernate=INFO
logging.level.org.hibernate.SQL=DEBUG
```

```
spring.jpa.properties.hibernate.generate_statistics = true
```

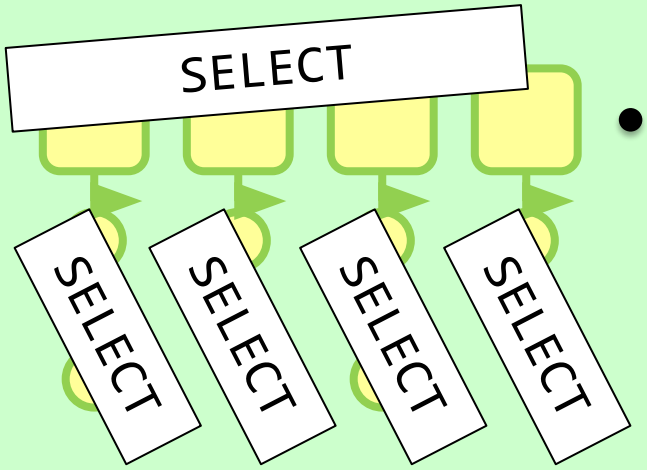
- Intercept them with **p6spy** (data source proxy)

```
spring.datasource.url=jdbc:p6spy:h2:tcp://localhost:9092/~/test
spring.datasource.driver-class-name=com.p6spy.engine.spy.P6SpyDriver
~~spring.datasource.driver-class-name=org.h2.Driver~~
... the rest stays the same ...
```

  - Can write to spy.log or to application log
  - Logs commits, execution time, actual query params (Yummy!), batches...

- Logging all SQLs hits performance
  - Enable it 'on-demand' at runtime via JMX or Spring Actuator

Lots of Identical Queries



••• x 1000 parents

=1001 queries

N+1 PROBLEM

run 1+ query for each parent

💀 Each SELECT is ran
via a dedicated network call

= network time penalty

(To tweak this default, check out Hibernate's @BatchSize)

When is the data fetched? □

Lazy Loading ?

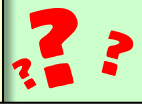
- **EAGER** = loaded before JPA gives you the entity

```
Course course = courseRepo.findById(courseId);
        // or = courseRepo.search(...)
String name = course.getName(); //already loaded
```

- **LAZY** = loaded from DB first time you access it

```
for (Student student : course.getStudents()) {
  ...
}
```

when entering the loop, a query is ran:



```
SELECT ... FROM STUDENT s .... WHERE s.COURSE_ID=?
```

How the hack?... *Exactly! A hack! A hacked Set<> implem.*

- Triggered by calling any method on the students collection: .size(), .iterator(),…
- Happens only the first time: data is then kept in memory
- Only works transaction is still open

Lazy vs Eager Loading
Defaults

- Direct attributes = eager
  - All columns in the entity's table are loaded
- @...ToOne = eager:
  - a) findById □ + JOIN Teacher
  - b) SELECT ... (JPQL) □ +1 SELECT ⚠️

```
SELECT ... FROM COURSE ...
SELECT ... FROM TEACHER t WHERE t.COURSE_ID=?
```

👌 *can be avoided using LEFT JOIN FETCH course.teacher*

- @...ToMany = lazy loaded
  - by a PersistentSet/PersistentBag first time you iterate over it

```
@Entity
public class Course {


  private String name;


  @ManyToOne
  private Teacher teacher;


  @OneToMany
  private Set<Student> students;

  ...
}
```

Lazy vs Eager Loading

Changing the Defaults

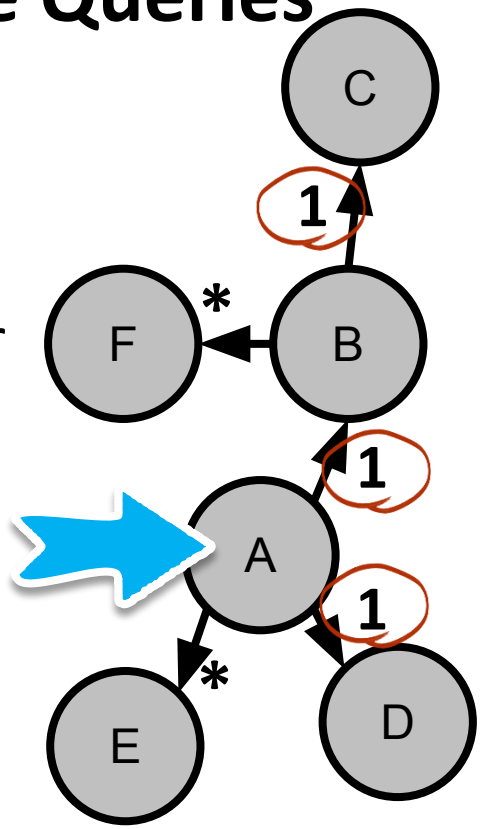
usually a BAD idea

- Direct fields = ~~eager~~ lazy
  - ? How: *Change bytecode of* Course.getName()
    ⚠ requires @Entity bytecode enhancement
    `@EnableLoadTimeWeaving on @Configuration`
  - 💡 Defer loading of @Lob fields
- @...ToOne = ~~eager~~ lazy
  - ? How: *proxies the Teacher* class, so calling any method (except getId()) triggers its full load
  - 💡 Defer loading large entities (# fields, blobs)
- @...ToMany = ~~lazy~~ eager
  - 💡 If parent has no meaning w/o children
  - – Heavy: for each Course, **always** +1 query ⚠
  - – 👉 LEFT JOIN FETCH on use-cases that NEED the children
  - – 👉 @BatchSize to load children in pages using IN in SQL

```
@Entity
public class Course {

   @Basic(fetch = LAZY)
   private String name;


   @ManyToOne(fetch = LAZY)
   private Teacher teacher;


   @OneToMany(fetch = EAGER)
   private Set<Student> students;

   ...
}
```

Symptom: Lots of Identical Simple Queries

- Becomes a problem under LOAD
  - + Loading a lists of results (eg landing page), or
  - + Doing extra queries on hot paths
- Example: SELECT from A
  - +3 queries for B,C,D (if JPQL/findAll is used)
  - +1 query for E's if lazy-loaded or fetchType=EAGER
  - +1 query for F (idem)



Avoiding extra queries

Select Parent JOIN Children

```
SELECT p FROM Person p LEFT JOIN FETCH p.addresses a
```

```
LEFT JOIN FETCH p.bio
```

Field, not collection

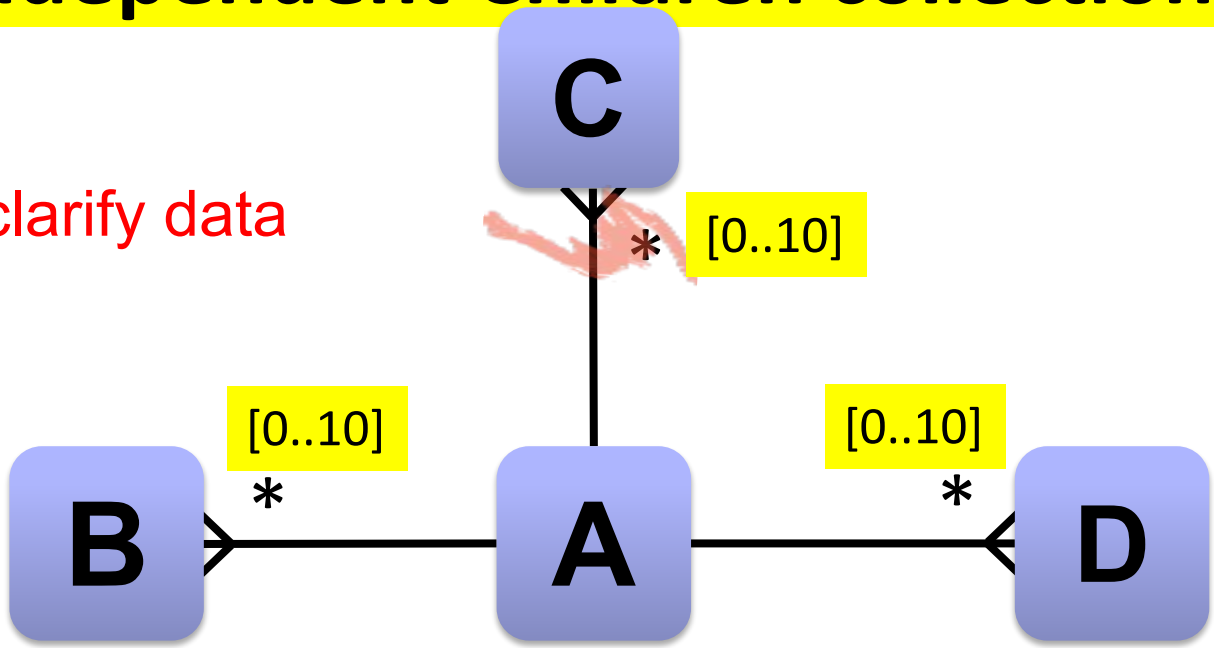
P_ID	P_NAME,....		A_ID	A_NAME,...	BIO_ID	BIRTHDATE
1	Person1	...	7	P1-Addr1	77	07-10-2012
1	Person1	...	8	P1-Addr2	77	07-10-2012
2	Person2	...	11	P2-Addr1	88	24-11-1985
2	Person2	...	12	P2-Addr2	88	24-11-1985
2	Person2	...	13	P2-Addr3	88	24-11-1985

Duplicate Rows

Many Columns

Cannot LEFT JOIN FETCH

independent Children collections



Always clarify data volumes

```
SELECT a                    jpql
FROM A a
LEFT JOIN FETCH a.bList
LEFT JOIN FETCH a.cList
LEFT JOIN FETCH a.dList
WHERE a.name LIKE '%a%'
```

CROSS JOIN

= Unrestricted combinations between A x B x C

x 10
x 10
x 10

= 1000 rows for every A!

Selecting Data Efficiently

(read-only data for search or export)

- Select individual fields using **JPQL** instead of fetching the full @Entity
  - **Scalar query**: SELECT e.id, e.name FROM Entity e … □ returns Object[] 🤢**types? order?**
  - **SELECT new** com...**MyDto**(e.id,…) FROM Entity e … □ send MyDto as JSON response
  - **SELECT e.id, e.name AS name** FROM Entity e ... □ class MyDto{ id, name + getter/setter}
  - **Spring Projections**: **interface** MyDto { long getId(); String getName(); } link
  - Select deep structures (parents with children) that match a criteria □ Driving Query
  - Fetch/Load Graphs (JPA) for deep partial fetching (not functional in recent Spring) link
- Native **SQL** Query

  = to use the full SQL power: aggregate(max..), within group .., UNION, FROM (SELECT ..), CTE
  - Spring Data @Query("<sql>", native=true) + **Spring Projections**
  - Map an @Entity Hibernate @Subselect
  - Map an @Entity on a DB VIEW
  - Materialized View (Oracle), eg updated every hour/day (= Eventual Consistency)
- "***Full-text search** over all the fields*" □ Elastic Search
  - Kept up to date eventually consistent via MQ

The usecase required reading 3 fields of a large entity

@Subselect

```
@Entity
//Mark as immutable
@Immutable
//SQL is directly in the class - easier to maintain between mult
@Subselect("select
                u.id ,
                u.last_name || ', ' ||u.first_name as fullname ,
                c.name as countryname
            from user u
            join country c on c.id = u.countrId")
public class UserWithCountryListing {

    @Id
    private String id;

    private String fullname;

    private String countryname;

    // only add getters

}
```

@Entity mapped on a DB VIEW

```
create or replace view parent_search as
select p.ID, P.NAME,
    STRING_AGG(c.NAME, ',') children_names -- native
from PARENT P
         left join CHILD C on P.ID = C.PARENT_ID
group by p.ID, P.NAME;
```

Creating the VIEW validates the SQL in DB

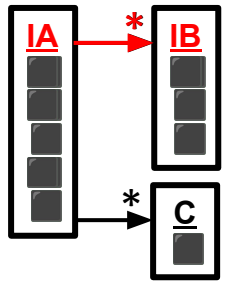
```
@Table(name = "PARENT_SEARCH")
@Entity
public class ParentSearch {
    @Id
    private Long id;
    private String name;
    private String childrenNames;
}
```

Pro Tip: join back from view to Entity via JPQL
(allows traversing the original entity links)

```
SELECT v
FROM ParentSearch ps
JOIN Parent p ON p.id=ps.id
JOIN ...
WHERE ... p.entityField
```

FetchGraph

```
@NamedEntityGraph(
   name = "post-entity-graph",
   attributeNodes = {
      @NamedAttributeNode("subject"),
      @NamedAttributeNode("user"),
      @NamedAttributeNode("comments"),
   }
)
@Entity
public class Post {

      @OneToMany(mappedBy = "post")
      private List<Comment> comments = new ArrayList<>();

      //...
}
```



Display 1.000.000 rows

User scrolling 1M rows? Really?!!

(Hint: *infinite scroll* is tricky to implement)

Pagination

1. In Browser (≤ 1000 rows): Paginate in DOM
2. In the user session on the app server – a BAD idea

   >> Sometimes used when generating html on server-side, eg with JSF, JSP, Vaadin,...
3. In DB via paginated query (SQL: LIMIT/OFFSET)

   ⚠️ A paginated query requires an ORDER BY to provide a stable row order

DB Pagination

- **Spring Data JPA**

```
Page<Parent> parentPage = repo.findByNameLike( namePart: "%ar%",
    PageRequest.of( page: 0,  size: 20, Direction.ASC,  ...properties: "name"));

interface ParentRepo extends JpaRepository<Parent, Long> {
   @Query("SELECT p FROM Parent p WHERE p.name LIKE ?1")
   Page<Parent> findByNameLike(String namePart, Pageable page);
```

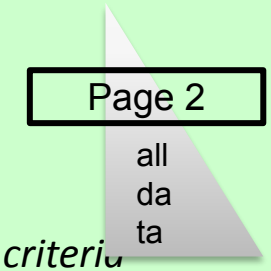
- **Bare JPA**

```
List<Parent> resultList   em createQuery(
        qSt ng  SELECT p FROM Parent p W ERE p name LIKE ?1   +
         ORDER BY p name asc   Parent class)
    setFirstResult(0)
    setMaxResults(20)
    getResultList()
```

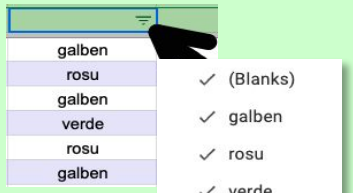
Search UX Tricks

- **Avoid N+1** queries (eg 1 + 20 rows x 2 SELECTs/row)
  - Especially for 'hot' pages (eg home-page)
  - Use: FETCH JOINs, @BatchSize, Aggregate data using native SQL query
- **Reduce #columns and JOINs** (see prev slides)
- **Sort** only indexable columns (eg not CLOB)
  - Even if you display 20 lines, database must sort the full result set

- **Total rows** (=another query that takes time) Do you really need it?



1) Display total pages *after* (async)? or
2) Don't even show total. Instead, display: "*More results available, please refine your criteria*"
   - Trick: run SELECT ... LIMIT 1001. If returns 1001 elements => show only 1000 + the message above
- **Ask User Confirmation** for queries expected to take a long time
  - eg Are you sure you want to search the employees with CV containing "e" ?
- **Display all possible values for all column** (like in Excel😇) 😱
  - WITH (SELECT heavy-SQL) AS x select **distinct** 'color', x.color from x union select **distinct** 'height', c.height from x union ...
- **Async serve** results for long-running queries/exports
  - "This query takes too long. We'll email you the results when they're ready" + UI blocker + "Loose unsaved changes" to prevent browser Refresh



select all values for columns

WITH (SELECT heavy 💀 -SQL) AS x

select **distinct** 'color' col, x.color from x union all

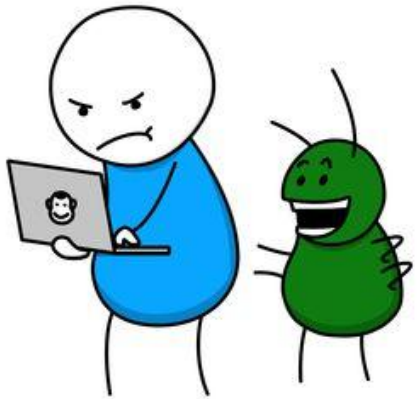
select **distinct** 'height' col, X.height from x union all

... 20 more ...

Fetch Children + Pagination

UI :

Id	Parent name	Children
1	Victor	Emma, Vlad
2	Peter	Maria, Stephan, Paul
3	Bachelor	



```
FROM PARENT
LEFT JOIN FETCH CHILDREN
R
LIMIT 4
```

allows parents without any children to remain in the result set

DB Pagination + JOIN Children is incorrect!

Solution

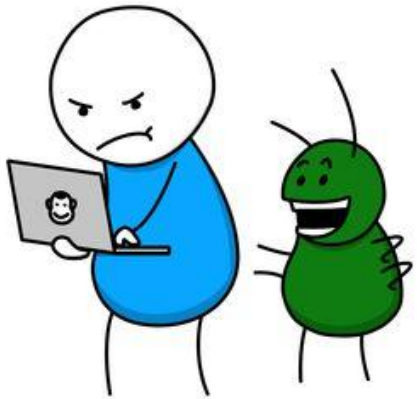
RESULT Set:

PARENT_ID	PARENT_NAME	CHILD_ID	CHILD_NAME
1	Victor	1	Emma
1	Victor	2	Vlad
2	Peter	3	Maria
2	Peter	4	Stephan
2	Peter	5	Paul
3	Bachelor	-	-

Fetch Children + Pagination

UI :

Id	Parent name	Children
1	Victor	Emma, Vlad
2	Peter	Maria, Stephan, Paul
3	Bachelor	



```
FROM PARENT
LEFT JOIN FETCH CHILDREN
R
LIMIT 4
```

allows parents without any children to remain in the result set

DB Pagination + JOIN Children is incorrect!

Solution

RESULT Set:

PARENT_ID	PARENT_NAME	CHILD_ID	CHILD_NAME
1	Victor	1	Emma
1	Victor	2	Vlad
2	Peter	3	Maria
2	Peter	4	Stephan
2	Peter	5	Paul
3	Bachelor	-	-

Driving Query

2 queries

Driving Query (identify IDs) List<Long>

```
SELECT p.id
FROM Parent p
WHERE <criteria>
[OFFSET 0 LIMIT 20]
```

or paginate in java

Fetching Data *

```
FROM Parent p
LEFT JOIN FETCH p.country
LEFT JOIN FETCH p.children ...
WHERE p.id IN (?, ?, ...?)
```

Max 1000 x ? (SQL syntax limitation)
Workaround:
a) p.id IN (0...1000) OR p.id IN (1001...2000) OR ... ∞
b) (p.id, 1) IN ((?,1), (?,1), ... ∞) --- on Oracle

Enthusiast/Naive FP use:

Let's bring all the data in Java and process it with Stream:

~~findAll().stream().filter(..)~~

- Prefer **WHERE** in DB for heavy pruning of data
  - **to save network traffic, memory, and CPU**
  - eg. WHERE EXIST (SELECT ...)

- .filter() by complex logic in Java
  - More **maintainable** and **Unit-Testable**❤️
  - 💡 Combine Spring Specification instead of Predicate 👉 example

Handle decent volumes in Java

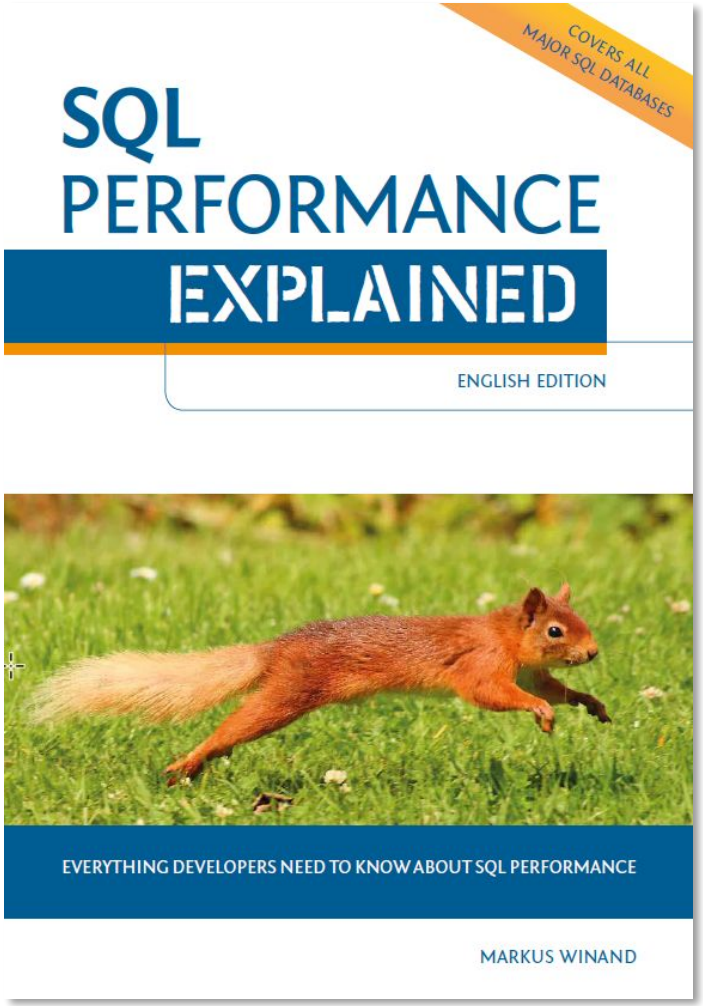
Tuning a JQPL

- **Measure it**
- **Get the native SQL**
  - COPY-PASTE from log (**p6spy**) to an SQL editor
- **Optimize SQL** □ (next slide)
- **Reverse-engineer back into JQPL**
  - that generates the SQL that you tuned

Optimizing an SQL

- Measure it (next slide)
- Analyze its Explain Plan
  - Beware of "full scans“
  - query /*+hints? */ can be dangerous
- Add an INDEX

  = persistent hash map/tree
  - Naive: add index on any column used in WHERE
  - But too many INDEXes sometime confuse the DB
  - Slower INSERTS (drop/create them at bulk imports?)
- Use partitioning for huge tables
- Break it in 2 SELECT (Driving query)
  - *(1) get ids [into temp table]*
  - *(2) load data*
- Read SQL Performance Explained



Top 20 SQL Tuning Tips

1. **Add INDEX** on (function of) cols in WHERE/ORDER BY/JOIN ON
   Better if: 🔼 high-selectivity (many unique vals); 🔽 low cardinality; 🔼 large tables (>1M rows); 🔽 redundant
2. INDEX slows down INSERT: drop INDEX before bulk insert massive data + recreate after
3. Use **EXISTS** (SELECT ...) instead of SELECT **COUNT**(SELECT ...) > 0 or WHERE id IN (SELECT id ...)
4. SELECT **only required columns**, instead of using SELECT *
5. Avoid **Correlated Subqueries:** eg SELECT .. FROM **x** WHERE (SELECT ..FROM y WHERE **x**.a=y.a)..
6. Avoid **SELECT DISTINCT a,b,c,d...g**, or * (including CLOB columns☠️)
7. Use WHERE clause instead of HAVING – filter before grouping
8. Favor **JOIN ON <cond>** over not WHERE <cond>; bonus: cleaner query
9. Use **LIMIT** to sample query results
10. Use **UNION ALL** instead of UNION (performs a DISTINCT) whenever possible
11. Avoid using **OR** in join queries
12. Use (SELECT ... WHERE **A**) **UNION** (SELECT ... WHERE **B**) vs SELECT ... WHERE **A** OR **B** -- to hit more indexes
    eg: (...where date between) UNION ALL (...where date is null) ⚔️
13. Choose **GROUP BY** over **window functions (OVER PARTITION BY)** = query-wise, not per-row
14. Use derived and temporary tables
15. Use CTE: with select ... as **A** select ... from **A** JOIN ... = cleaner
16. Defer your query during **off-peak hours** (eg midnight)
17. Use **materialized** views instead of views for slow-changing data (eg top most ordered products)
18. Avoid != or <> (not equal) and NOT IN => prefer = or IN
19. Use **less nested subqueries**; prefer to flatten them JOINing them in the main FROM clause
20. If INNER and LEFT/RIGHT JOIN produce the same results, don't use **INNER**?
21. To avoid retrieving the same dataset frequently, try to use temporary tables. (eg in

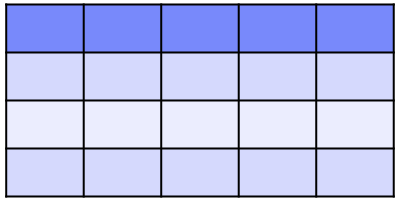
SQL Benchmarking

- Reconnect between attempts, to clear DB caches
  If you see times like: 2.5s, then 0.01s
- Be the only one on that DB machine
  Local Docker might be more consistent than a 🌧️ server
- Mind *the forgotten query*

(still running since yesterday😋)

- Iso-prod data volumes, *similar histogram*
- Up-to-date schema, indexes, up-to-date statistics

Efficient Queries

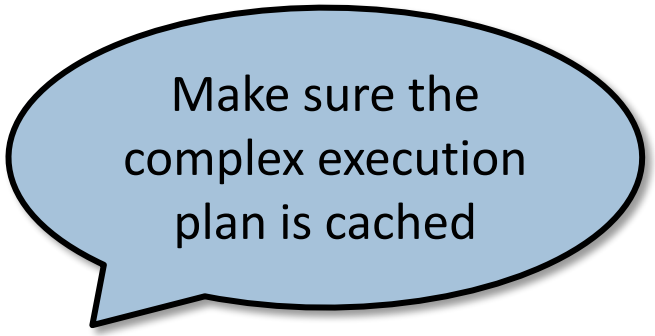


- Limit the #queries
- Limit the *height* of the result set (#rows)
  - Per web request (UI) or per processed record (batch)
  - control lazy load
- Reduce the *width* of the result set (#columns)
  - select DTOs

- Optimize the SQL query plan

Execution Plan

(some ideas)



- How is an SQL actually executed?
  - RDBMS translates the sql string into an execution plan.
- *DS* JOIN B(w/ index) JOIN C(w/o index)
  - Indexed JOIN
- A (count=10) JOIN B (count=10K) JOIN C(1M)
  - Starts with the smallest dataset
- *DS* JOIN B(2 values/join column) JOIN C(unique col)
  - JOIN C (better chance to reduce the running DS

JPA Caching

1st Level Caching

- Always ON

- aka transaction-scoped cache:
- anyway you get Entity{id=7} in the same Transaction:
  - findById
  - SELECT JPQL
  - Child of another loaded Entity

- , you will always get the same instance (==)
  - Served from memory (from Persistence Context)

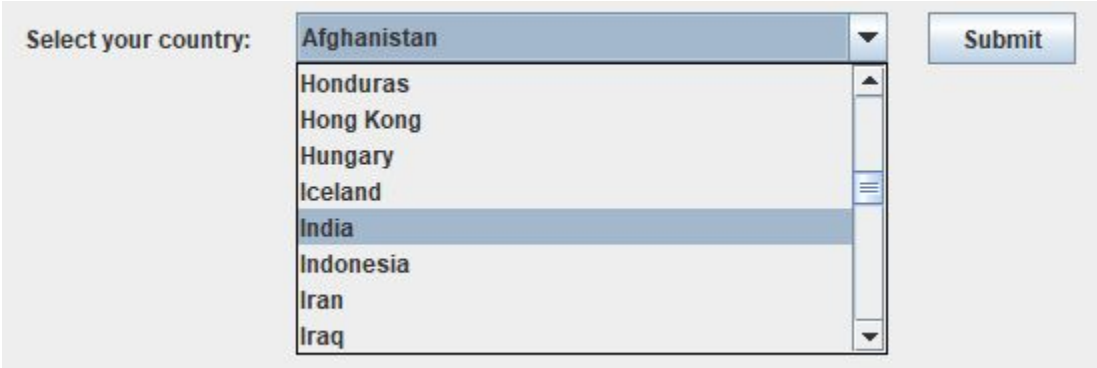
2nd Level Caching

- By default OFF
  - How to enable: link, eg backed by ehcache
- Shared cache across different requests (transactions):
- You can cache:
  1) Entity state by ID (@Cache on @Entity) 👉 findById
  2) Children of Entities (@Cache on @ManyToOne) 👉 load children w/o 2nd query
     - Stores just children IDs (uses #1 above for data)
  3) Query results for given params (@QueryHint on @Query) 👉 MyRepo.findAll
     - Usual concern: Limiting size for parameterized queries, eg via custom cache Regions
     - Only IDs are cached (uses #1 for data)
- Same problems as with any long-lived cache
  - Invalidation, entry expiration, max heap occupied

Modeling for Performance

Do all Entities really have to refer to each other?
(eg: Customer.country:Country)

Imagine you see this in a Single-Page-App:



□ countryId

countryId □

□ the FE has to know all country names

How would you send/receive the country id + name to/from backend?

The country ID is enough!

Do all Entities <u>really</u> have to refer each other?

```
@ManyToOne
private Country originCountry;
```

WHY?

Entity Reference vs Numeric Column

```
private Long originCountryId;
```

NOTE: Keep the FK for consistency!❤️

☺️- 1 JOIN/SELECT

☹️ programmatic mapping when required

in frontend: const countries = [];

in backend: Map<Long, String> countryNames = ...

or @Cacheable

RECORD_STATUS	
1	DRAFT
2	ACTIVE
3	DELETED

```
@ManyToOne
private RecordStatus status;
```

STATUS Table vs Status Enum

```
@Enumerated(STRING)
private RecordStatus status
```

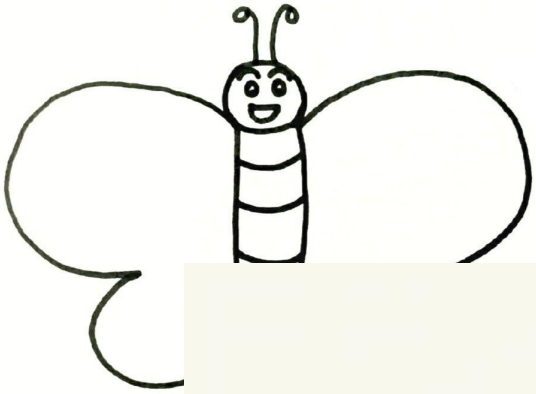
- 1 JOIN

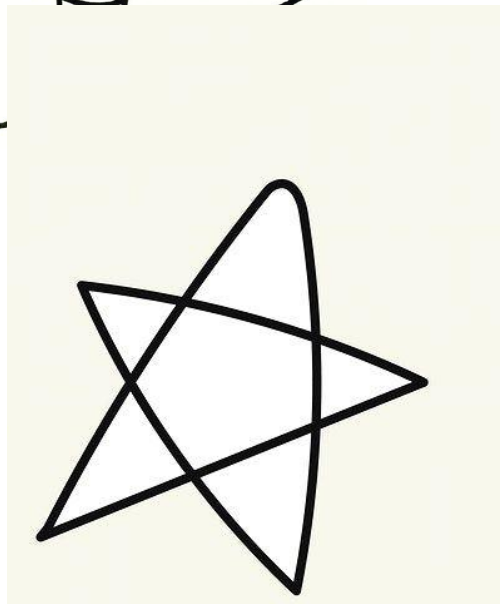
+ cleaner code
if (r.status == DRAFT)

```
enum Status {
    DRAFT,
    ACTIVE,
    DELETED
}
```

If you really want to store "D" for "DRAFT", check out this article (but first please measure the benefit)

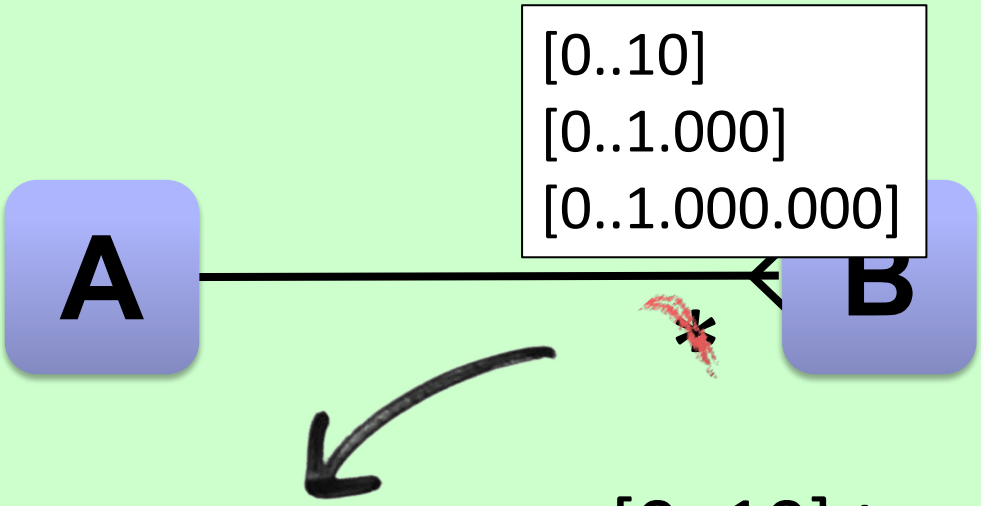
Kindergarten Time





When you're young, you like these symbols

Know Your Data Volumes!



[0..10] tags

Know expected volumes:

- At Release
- After 1 year
- In 5 years (est)

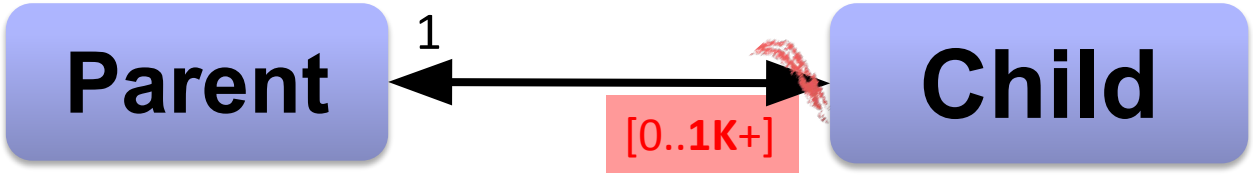
[0..1K] orderLines

[0..1M] transactions

Different @Entity Design

[0..20M]

NOSQL



@OneToMany

Only link Child → Parent (unidirectional)

```
Paginate: childRepo.findByParentId(parentId, pageable):Page<Child>
```

```
Stream: childRepo.findByParentId(parentId):Iterator<Child>
```

@ManyToMany

~~@JoinTable~~ becomes separate @Entity →
(paginate or stream from Repo)

```
@Entity
class Link {
  @Id Long id;
  Long aId;
  Long bId;
}
```

@ElementCollection

- = PK-less private children C of a parent P

Every time you change any C of a P,
All C for that P are removed + re-inserted

- Solution:
  - Make C and @Entity and
    use @OneToMany(cascade=MERGE) in P

Split Large Entities

https://vladmihalcea.com/the-best-way-to-map-a-onetoone-relationship-with-jpa-and-hibernate/

- Split large entities (to load large entities on demand)
- A ---@OneToOne---> B
- Should A have A.ID and A.B_ID ?
- A and B should share the P
  - A.B_ID is both PK, and FK to B

```
@Entity
class ProfData {
   @Id
   private Long id;

   @OneToOne
   @MapsId // shared PK with Prof
   private Prof prof;

   full-details(dozens of fields)
   clob/blob
}
```

Dynamic Queries

(implementing search forms)

Search Query: Concatenation

```
public List<Course> search(CourseSearchCriteria criteria) {
    String jpql = "SELECT c FROM Course c WHERE 1=1 ";
    Map<String, Object> params = new HashMap<>();

    if (StringUtils.isNotBlank(criteria.namePart)) {
        jpql += " AND UPPER(c.name) LIKE UPPER('%' || :name || '%') ";
        params.put("name", criteria.namePart);
    }

    if (criteria.teacherId != null) {
        jpql += " AND c.teacher.id = :teacherId";
        params.put("teacherId", criteria.teacherId);
    }

    if (criteria.studentId != null) {
        jpql += " AND c.id IN (SELECT cc.id FROM Student s JOIN s.courses cc WHERE
s.id=:sId)";
        params.put("sId", criteria.studentId);
    }

    TypedQuery<Course> query = em.createQuery(jpql, Course.class);
    for (String key : params.keySet()) {
        query.setParameter(key, params.get(key));
    }
    return query.getResultList();
}
```

Search Query: Criteria API

```
public List<Course> search(CourseSearchCriteria searchCriteria) {
     CriteriaBuilder cb = em.getCriteriaBuilder();
     CriteriaQuery<Course> q = cb.createQuery(Course.class);
     Root<Course> r = q.from(Course.class);
     Map<String, Object> params = new HashMap<>();
     List<Predicate> predicates = new ArrayList<>();

     if (StringUtils.isNotBlank(searchCriteria.namePart)) {
          predicates.add(cb.like(
                    cb.upper(r.get("name")),"%" +searchCriteria.namePart.toUpperCase() +
"%"));
     }

     if (searchCriteria.teacherId != null) {
          predicates.add(cb.equal(
                    r.get("teacher").get("id"),  searchCriteria.teacherId));
     }

     q.select(r).where(predicates.toArray(new Predicate[0]));
     TypedQuery<Course> query = em.createQuery(q);
     for (String key : params.keySet()) {
          query.setParameter(key, params.get(key));
     }
     return query.getResultList();
}
```

Search Query: Criteria + Metamodel

```
public List<Course> search(CourseSearchCriteria searchCriteria) {
     CriteriaBuilder cb = em.getCriteriaBuilder();
     CriteriaQuery<Course> q = cb.createQuery(Course.class);
     Root<Course> r = q.from(Course.class);
     Map<String, Object> params = new HashMap<>();
     List<Predicate> predicates = new ArrayList<>();

     if (StringUtils.isNotBlank(searchCriteria.namePart)) {
          predicates.add(cb.like(
                    cb.upper(r.get(Course_.name)), "%" +searchCriteria.namePart + "%"));
     }

     if (searchCriteria.teacherId != null) {
          predicates.add(cb.equal(
                    r.get(Course_.teacher).get(Teacher_.id),   searchCriteria.teacherId));
     }

     q.select(r).where(predicates.toArray(new Predicate[0]));
     TypedQuery<Course> query = em.createQuery(q);
     for (String key : params.keySet()) {
          query.setParameter(key, params.get(key));
     }
     return query.getResultList();
}
```

Generated Classes

Specifications

= hide bits of CriteriaBuilder code behind composable 'predicates', manipulated in business logic

```java
public class TeacherSpecifications {
    public static Specification<Teacher> hasNameLike(String name) {
        return (root, query, cb) ->
            cb.like(cb.upper(root.get(Teacher_.name)), pattern: "%" + name.toUpperCase() + "%");
    }

    public static Specification<Teacher> hasGrade(Grade grade) {
        return (root, query, cb) ->
            cb.lessThan(root.get(Teacher_.grade), grade);
    }
}
```

```java
public class BusinessClass {
    private final TeacherRepo teacherRepo;

    public void method() {
        Specification<Teacher> spec = TeacherSpecifications.hasGrade(Grade.CONF)
            .or(TeacherSpecifications.hasNameLike("a"));
        List<Teacher> teachers = teacherRepo.findAll(spec);
    }
}
```

Static JPQL with "OR"

deploy-time checked, awkward to read/debug

```
@Query("""
    SELECT t FROM Teacher t
    WHERE (:name is null OR
        UPPER(t.name) LIKE UPPER('%' || :name || '%') )
    AND (:grade is null OR
        t.grade = :grade )
        AND (cast(:teachingCourses as int) = 0 OR
        EXISTS (SELECT 1
            FROM CourseActivity c
            JOIN c.teachers tt
            WHERE tt.id = t.id) )
    """)

List<Teacher> searchFixedJqpl(
    String name, Grade grade, boolean teachingCourses); *
```

*Check the workshop code for a variation that uses SpEL expressions to take a structure as param, not primitives

Query DSL

More fluent than Criteria API

```
public List<Teacher> searchQueryDSL(TeacherSearchCriteria searchCriteria) {
    JPAQuery<?> query = new JPAQuery<Void>(em);
    BooleanExpression pred = Expressions.TRUE;

    if (searchCriteria.grade != null) {
        pred = pred.and(teacher.grade.eq(searchCriteria.grade));
    }
    if (searchCriteria.name != null) {
        pred = pred.and(teacher.name.upper()
            .like( str: "%" + searchCriteria.name.toUpperCase() + "%"));
    }

    return query.select(teacher)
        .from(teacher)
        .where(pred)
        .fetchAll()
        .fetch();
}
```

https://spring.io/blog/2011/04/26/advanced-spring-data-jpa-specifications-and-querydsl/

@DynamicUpdate

- By default Hibernate sends all columns at UPDATE
  - Index damage
  - Network traffic

```
Account account = accountRepository.findOne(ACCOUNT_ID);
account.setName("Test Account");
accountRepository.save(account);
```

```
update Account set active=?, name=?, type=? where id=?
```

- Solution:

```
@Entity
@DynamicUpdate
public class Account {
    // Existing data and methods
}
```

```
update Account set name=? where id=?
```

Efficient Transactions

Efficient Transactions: Summary

- Monitor connection usage in High-TPS systems
  - ↑Conn acquire waiting time starvation issue
- **Don't block DB connections**
  - Avoid @Transactional for READs (avoid lazy-load)
  - Avoid propagation=REQUIRES_NEW and NOT_SUPPORTED : they acquire +1 conn
  - Tune Transactions size:
    - **Acquire later:** `hibernate.connection.provider_disables_autocommit=true`
    - **Release earlier:** `spring.jpa.open-in-view=false`
  - Avoid launching REST calls while a @Transactional is active
  - Consider timeout for dangerous @Transaction(timeout=
  - Fragment long-running transactions: import/export in pages
  - Avoid/Minimize duration you keep DB Locks:
    - Table lock: LOCK TABLE (SQL)
    - Row lock: SELECT FOR UPDATE (SQL) or `em.lock(entity, PESSIMISTIC_WRITE);` (JPA)

Import/Export Data

Export 1.000.000 rows

e.g: to file

Iterate over rows

1. for (rs.next())
2. Iterator<Record>
3. Stream<Record>

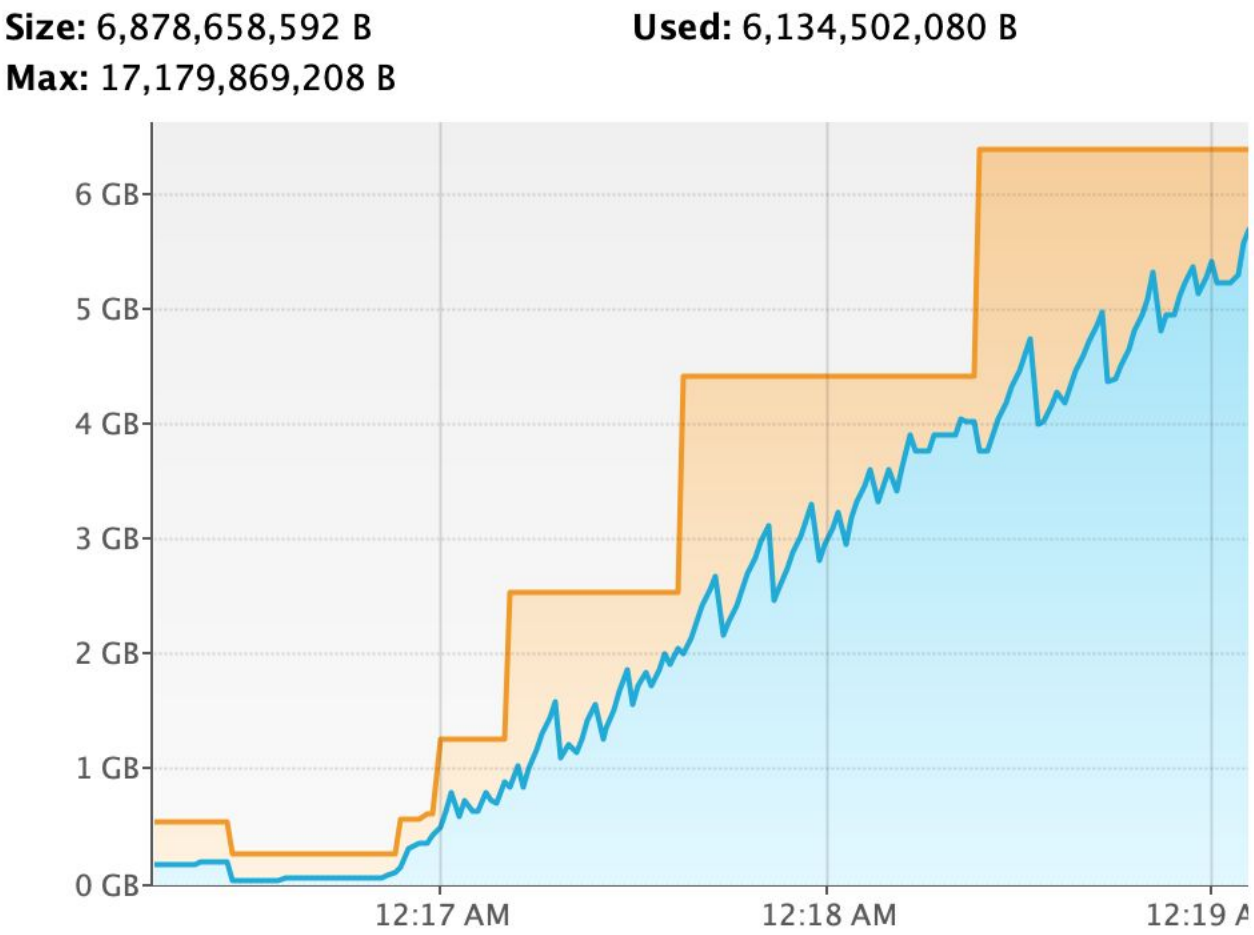
JDBC load chunks of fetch_size rows from DB

Increase fetch_size for massive reads

\+ Less network round-trips

\- More memory used

Massive Export accumulating heap



@Lob: Storing Files in DB

(see LobTest.java)

Latency

- For a file upload, start the processing @Async/@Scheduled
- Tell the user "File under processing..."

Memory

- Loading entire file in byte[] is dangerous (OutOfMemory)

  => if you do it, careful not to eagerly preload the @Entity holding the byte[]
- Better: Stream from/to a temp file into a sql.Clob/Blob
- Delete the temp file after Tx end
  - @TransactionEventListener(phase=AFTER_COMPLETION)

Import 1.000.000 rows

~~List<Record> list = readFile...~~

Streaming records from the file to DB

~~Send 1 INSERT/UPDATE per DB round-trip~~?

try 50, 100, 500

Send changes in bulks of batch_size

+ Less DB round-trips

- On error, which failed?!

Fastest:

Bulk insert into temporary tables + PL/SQL

Upload file to DB machine > SELECT FROM FILE + PL/SQL

Usually Spring Batch' speed is enough

Temporarily disable constraints, indexes

Partitioning

Temp Table +Truncate

Import 1.000.000 rows

Transaction Size

Parameterize it
and experiment close to prod

Insert all in 1 Transaction: Simple!

Speed

Smaller transactions work faster (50-100 Inserts/Tx)

Must-have: performance testing env

Multiple Smaller Transactions

What to do on failure?

a) Delete committed data: `DELETE FROM RECORD WHERE RECORD.JOB_START=?`

b) Skip errors – do you tolerate incomplete data?

c) Stop-Resume – can you ask for human intervention?

Link to an Existing Entity in DB

- Don't use find()
  - = +1 SELECT

```
country = countryRepo.getOne(13);
```

```
country = countryRepo.findById(13);
```

... if you want to FK an entity that exists in DB

eg. country_id=13

```
person.setCountry(country);
repo.save(person);
```

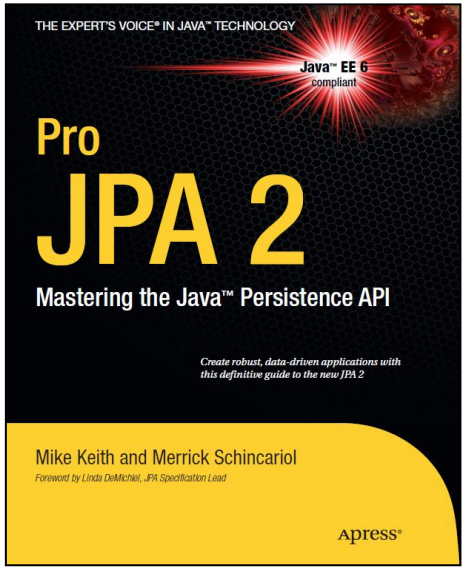
- use repo.getReferenceBy/em.getReference to avoid the SELECT:
  - Returns a proxy to an (hypothetical) entity in DB with id 13
  - No SELECT is ran, no fields are loaded
  - At insert/update, the FK will be set to person.*country_id=13*

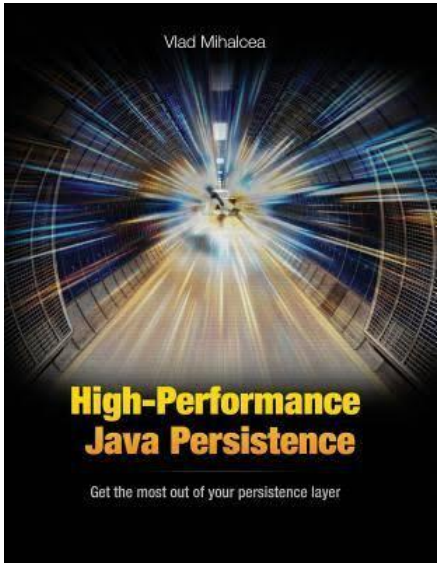
Experiment ideas

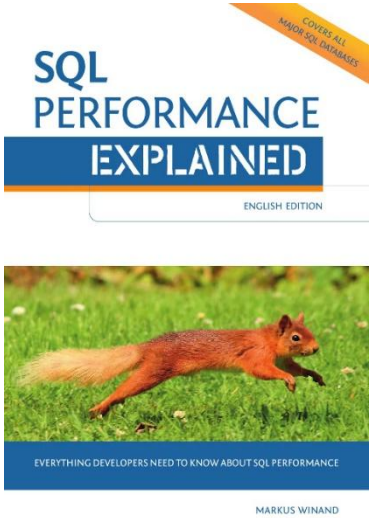
To earn a JPA Badge Achievement, define, run and test an idea:

- define and insert a bidirectional JPA link. protect it with methods.
- define and insert a unidirectional @OneToMany
- upload / download a file in a @Lob Clob field
- use @Embeddable in an @Entity. make it immutable + valid via ctor. insert+test
- demonstrate orphanRemoval
- save a new parent cascading on children list
- see an INSERT happen after the end of the method doing repo.save(new E)
- lazy loading on a children list. try: fetch=EAGER; LEFT JOIN FETCH jpql; @BatchSize
- see auto-flushing of dirty entities at end of transaction
- cause a @Transaction to rollback
- use propagation=REQUIRES_NEW; try @Tra..(readOnly=true
- call a @Transactional method locally -> proxy doesn't work! check it!
- [hard] REST API to get data then merge() it back. try @Version + 2 clients. (eg use Postman);
  then, do the updates without merge(), by e=find();e.setA();
- Write a dynamic query (search) using 1)jpql+= and 2)Criteria API; then paginate it.
- Write a "SELECT new" jpql
- Write a native query in a Spring Data Repo
- Insert 1M items in a Docker/real database in the shortest amount of time

More?







- JPA Standard: http://download.oracle.com/otndocs/jcp/persistence-2_1-fr-eval-spec/index.html
- Advanced ORM Mapping:http://en.wikibooks.org/wiki/Java_Persistence
- JPA Guru: https://vladmihalcea.com/
- JPA Guru: https://thorben-janssen.com/
- Advanced tutorials: http://baeldung.com/

1. *JPA salad, with history*
2. *Friendly field mapping*
3. *Hot relations in sweet database syrup*
4. *Inherited or stolen beans*
5. *Freshly picked primary keys*
6. *Lazy beef eagerly served*
7. *Well baked transaction*
8. *Undercover cache file*
9. *Fish .merge()d with wine*
10. *Optimistic veal calf steak*
11. *Duck queried in the forest*
12. *Spring Data Packages*
13. *Performant pancakes (discussion)*