Ciências ULisboa
Faculdade de Ciências da Universidade de Lisboa

2019/2020

Construção de Sistemas de Software

ORM com JPA

JPA

- JPA = Java Persistence API

  API Java que define a norma para mapear modelos de objetos Java (POJOs) em modelos relacionais (aplicações Java SE e EE)

  - javax.persistence
- O mapeamento é baseado em meta-dados (*metadata*)
  - o processo é repetitivo e pode ser automatizado em função de meta-dados, nomeadamente o esquema das tabelas e das classes
  - no caso de JPA os meta-dados podem ser especificados através de anotações
- Aspetos do mapeamento
  - Estruturais: classes vs. tabelas
  - Comportamentais: criar e alterar objetos vs. persistir dados

Anotações

- São meta-dados que influenciam a análise do programa, a geração de código ou a execução do programa
- Tornaram-se uma forma popular de, nas linguagens de programação, fornecer informação adicional
  - Java, C#, Python, Swift (Objective C), ...

Em Java

- aplicam-se aos constructos Java como tipos, métodos, atributos, tipos de parâmetro, ...
- são organizadas em **tipos** e definidas com **@interface**
- cada tipo de anotação tem um conjunto de **elementos**, alguns dos quais têm definidos valores *default* e que portanto são opcionais
- são introduzidas nos programas usando o símbolo @ seguido do **tipo de anotação** e de **valores** para os seus **elementos**

Anotações

java.lang

Annotation Type SuppressWarnings

```
@Target(value={TYPE,FIELD,METHOD,PARAMETER,CONSTRUCTOR,LOCAL_VARIABLE})
@Retention(value=SOURCE)
public @interface SuppressWarnings
```

Required Elements

Modifier and Type	Required Element and Description
`String[]`	**value** The set of warnings that are to be suppressed by the compiler in the annotated element.

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
     * The set of warnings that are to be suppressed by the compiler in the
    String[] value();
}
```

```
@SuppressWarnings(value = "unchecked")
void myMethod() { ... }
```

```
@SuppressWarnings("unused")
private String description;
```

Aspetos Estruturais do ORM: Classes e Tabelas

- Apenas em casos simples vai haver uma correspondência de um para um
- Mapeamento de uma classe numa tabela (***mapped classes***):
  - a informação contida nos objetos desta classe vai ser persistida
  - estas classes (e os seus objetos) designam-se por **entidades**

- **JPA**: Anotar a classe com **@Entity**
  - por omissão é usado o nome da classe para dar o nome à tabela correspondente
  - existe uma anotação **@Table** que permite especificar outro nome
  - como o JPA segue o princípio do ***configuration by exception***, só se anota o que é diferente do assumido por omissão
  - aplicam-se algumas restrições como não ser *final*, não ter atributos *final*, ter construtor *public/protected* sem argumentos...

Aspetos Estruturais do ORM: Classes e Tabelas

- Aspetos estruturais do mapeamento de uma classe numa tabela:
  - Referência vs. Chaves Primária
  - Mapeamento de **propriedades** (*data attributes*)
    - como persistir uma propriedade de um objeto
  - Mapeamento de **relações**
    - como persistir associações, agregações e composições entre objetos
  - Mapeamento de **relações de herança**
    - como persistir objetos de classes construídas com herança

Referência vs. Chave primária

- No modelo de objetos
  - os objeto são identificados por uma referência
  - um objeto guarda a ligação a outro objeto como uma **referência**
  - as referências são **diferentes** de execução para execução
- No modelo relacional
  - os registos são identificados de forma única pela sua **chave primária**
  - as ligações são guardadas recorrendo a **chaves**

Padrão *Identity Field*

- Guardar num **atributo** a identidade da linha da tabela correspondente ao objeto
  - imutável e único
  - corresponde à **chave primária da tabela** em que a classe é mapeada
- Decisões relacionadas
  - Chaves com significado?
  - Chaves simples vs compostas?
  - Tipo das chaves?
  - Geração das chaves?

Chaves com significado?

- Chaves com significado são por exemplo os números de identificação no fisco (nif), como cidadão de um país (ncc), como aluno de uma escola,…

- A chave primária deve ser imutável e única
  - uma chave que contém dados com significado estará sempre sujeita a possíveis alterações, em particular por causa de erros humanos
  - as necessidades de negócio podem ditar que os dados com significado, que eram únicos quando o sistema foi projetado, deixem de ser únicos

- Numa chave primária sem significado é possível escolher um tipo de dados que otimize as pesquisas indexadas

Chave primária

JPA:

- Anotar um atributo de uma classe mapeada com **@Id** serve para declarar que este atributo corresponde à chave primária da tabela correspondente
- A anotação **@GeneratedValue** serve para declarar que os valores da chave primária serão gerados de forma automática
  - existem várias estratégias para gerar a chave primária e são dependentes da base de dados usada
  - se não escolhermos nenhuma (**AUTO**), será a implementação do JPA a escolher o modo de efetuar a geração da chave

```
public class Person {
    @Id @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;
```

Estratégias de Geração

javax.persistence

Enum GenerationType

AUTO
Indicates that the persistence provider should pick an appropriate strategy for the particular database.

IDENTITY
Indicates that the persistence provider must assign primary keys for the entity using a database identity column.

SEQUENCE
Indicates that the persistence provider must assign primary keys for the entity using a database sequence.

TABLE
Indicates that the persistence provider must assign primary keys for the entity using an underlying database table to ensure uniqueness.

Mapeamento de Propriedades

- As propriedades dos objetos a persistir correspondem a ***data attributes***

  T attName

- A forma de definir o mapeamento depende da natureza do tipo T
  - Tipos dados básicos, *built-in*
  - Tipos dados *built-in* que representam tempo
  - Tipos de dados com múltiplos valores
  - Tipos de dados *user-defined*
    - enumerados
    - classes

Mapeamento de Propriedades de Tipos Básicos

- O modelo de objetos assenta num catálogo de tipos de dados oferecido pela linguagem veículo
  - **int, long, double, ...**
  - **char, String, ...**
- O modelo relacional assenta num catálogo de tipos de dados suportado pelo SGBD escolhido
  - **SMALLINT, INTEGER, BIGINT, REAL, ...**
  - **CHARACTER, VARCHAR, ...**
- **JPA**:
  - Nas classes mapeadas, ao não colocar nenhuma anotação nos atributos com estes tipos, tira-se partido do ***auto-mapping*** — o mapeamento é neste caso escolhido pela implementação do JPA. Alternativamente, pode se explicitamente usar a anotação **@Basic**

Mapeamento de Propriedades Temporais

- A classe **java.util.Date** permite registar instantes no tempo
- As bases de dados têm tipos distintos para representar
  - Datas, Horas e Instantes

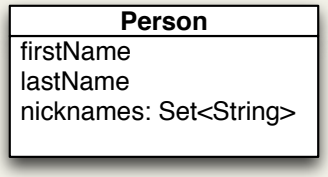
  representados no JDBC com
  - **java.sql.Date** (dia,mês,ano), **java.sql.Time** (hora,min,seg,miliseg), **java.sql.TimeStamp** (instante com precisão até nanoseg)

  todas estendem **java.util.Date**
- É necessário indicar como se pretende converter **java.util.Date** em tipos de BD, anotando com
  - **@Temporal(TemporalType.DATE)** para converter **Date** para **Datas**
  - **@Temporal(TemporalType.TIME)** para converter **Date** para **Horas**
  - **@Temporal(TemporalType.TIMESTAMP)** para converter **Date** para ***Timestamps***

Mapeamento de Propriedades Multi-valor

- No modelo de objetos, uma propriedade de um objeto pode ter **vários valores** (listas, mapas, vetores,... de dados) ou agregar objetos que não existem por si e não são partilhados
- No modelo relacional, cada campo de um registo só pode ter **um valor** (primeira forma normal)
  - para representar que o campo **c** pode ter mais do que um valor na tabela **A**
    - criar uma tabela **C** para guardar os valores de **c**
    - incluir em **C** um campo com a chave primária de **A** — chave estrangeira
- **Padrão *Dependent Mapping***: A classe dona trata do mapeamento dos seus dados/objetos que agrega.
- **JPA**: Anotar atributo que representa propriedade multi-valor com **@ElementCollection** (que se aplica também a atributos multi-valor de tipos *embeddable*)

Mapeamento de Propriedades Multi-valor



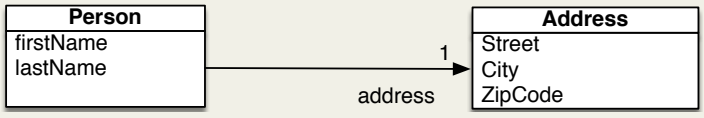

```
@Entity
public class Person {

    @Id @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;

    private String firstName;
    private String lastName;


    @ElementCollection
    private Set<String> nicknames = new HashSet<String>();
```

Mapeamento de Propriedades com Tipos Enumerados

- Um enumerado representa uma lista de opções possíveis
- Estas propriedades podem ser persistidas de duas formas
  1. Usando um **inteiro** que se refere à **ordem** da opção na lista com a anotação **@Enumerated(EnumType.ORDINAL)**
  2. Usando uma ***String*** com o **nome** da opção com a anotação **@Enumerated(EnumType.STRING)**
- Como escolher?
  - 1 falha quando se altera a ordem dos elementos da lista quer por inserção, remoção ou permutação de elementos
  - 2 ocupa mais espaço e falha quando se renomear a opção

Mapeamento de Propriedades com Outros Tipos *User-defined*

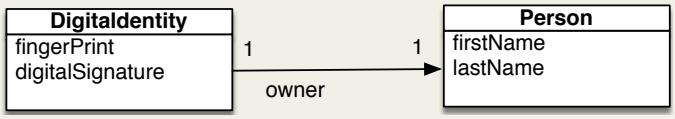
- Os objetos destas classes são apenas uma forma de organizar dados
  - São acessíveis apenas através do objeto que os contém (dono)
  - Não são partilhados, fazem sentido apenas no contexto do dono
- Não faz portanto sentido que sejam entidades, i.e., terem identidade



- **Padrão *Embedded Value*:** Mapear os valores destes objetos em atributos da classe que é dona do objeto. Ou seja, persistir estas propriedades na tabela em que a classe dona da propriedade é mapeada.
- **JPA**: Anotar com **@Embeddable** a classe que define o tipo do atributo e anotar o atributo com **@Embedded**

Relações (1 ou muitos)-para-1

- No modelo de objetos, uma relação **1-para-1** ou **muitos-para-1** entre objetos dos tipos **A** e **B** pode ser representada com um **atributo** em **A** de tipo **B**

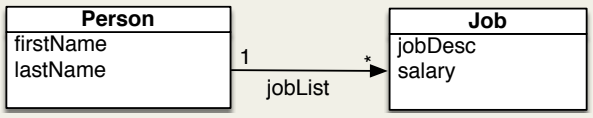


- No modelo relacional, uma relação **1-para-1** entre os registos das tabelas **A** e **B** é representada através de **chaves estrangeiras**
  - eg., uma **chave estrangeira** para **B** na tabela **A**

- **Padrão** ***Foreign Key Mapping***: Mapear estas relações entre objetos numa referência a uma chave estrangeira

Relações 1-para-muitos

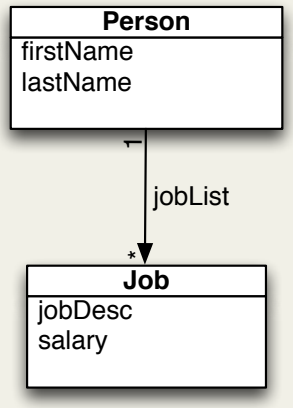
- No modelo de objetos, uma relação **1-para-muitos** entre objetos dos tipos **A** e **B** pode ser representada com **atributos de tipos multi-valor**
  - um atributo em **A** que tem uma coleção, vetor, mapa, etc, de objetos **B**

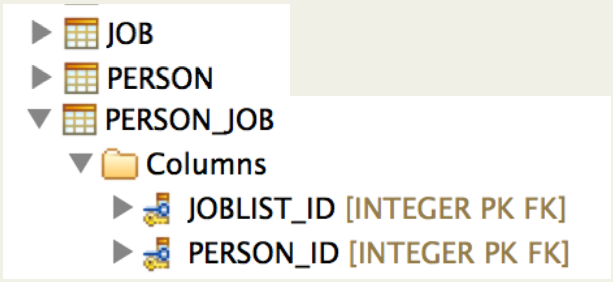


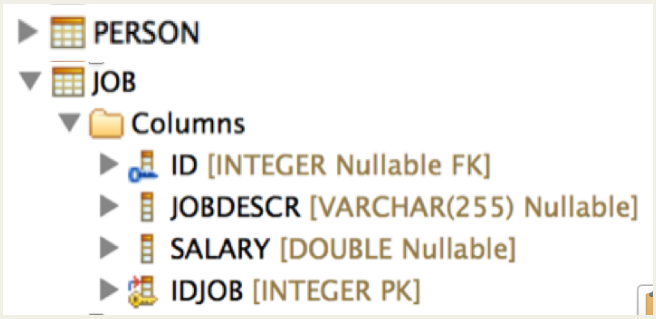
- No modelo relacional, uma relação **1-para-muitos** entre os registos das tabelas **A** e **B** é representada através de **chaves estrangeiras**
  - uma **chave estrangeira** para **A** na tabela **B** ou
  - uma tabela extra **C** com as chaves primárias de **A** e de **B**

- Dois **padrões** para o mapeamento desta relações:
  - ***Foreign Key Mapping*** ***Association Table Mapping***

Relações 1-para-muitos

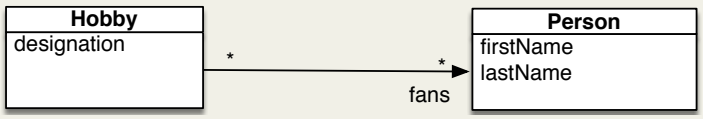






Relações muitos-para-muitos

- No modelo de objetos, uma relação **muitos-para-muitos** entre objetos dos tipos **A** e **B** pode ser representada com **atributos de tipos multi-valor**
  - eg. um atributo em **A** que tem uma coleção de objetos do tipo **B**



- No modelo relacional, uma relação **muitos-para-muitos** entre os registos das tabelas **A** e **B** é representada através de tabelas adicionais e de **chaves estrangeiras**
  - uma tabela extra **C** com as chaves de **A** e de **B**
- **Padrão** para o mapeamento desta relações:

  Association Table Mapping

Mapeamento de Relações

- A partir da informação existente nas classes não é sempre possível determinar a natureza das relações
  - eg., um atributo na classe A com o tipo Collection<B> tanto pode representar uma relação 1-para-muitos como muitos-para-muitos entre A e B
- E portanto, não é possível determinar automaticamente como deve ser feito o mapeamento
  - eg,. determinar se é preciso ter uma tabela extra ou não
- Para efeitos de mapeamento da relação subjacente, é necessário dar informação adicional sobre a relação

- **JPA**: Anotar atributos declarando o tipo da relação @OneToMany @ManyToOne @ManyToMany

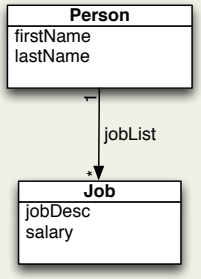
Mapeamento de Relações

- Anotar um atributo do tipo B da classe A com @ManyToOne indica que o atributo é o lado *many* de uma relação 1-para-muitos

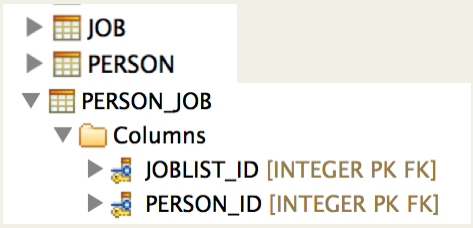
- Anotar um atributo do tipo Collection<B> da classe A com @OneToMany indica que este atributo é o lado *one* de uma relação 1-para-muitos
- A anotação @JoinColumn indica que a relação 1-para-muitos vai ser materializada no modelo relacional através de uma chave estrangeira na tabela B.
- Por omissão o mapeamento é feito com uma tabela extra para descrever a relação

- Anotar um atributo do tipo Collection<B> da classe A com @ManyToMany indica que este atributo é um dos lados de uma relação muito-para-muitos

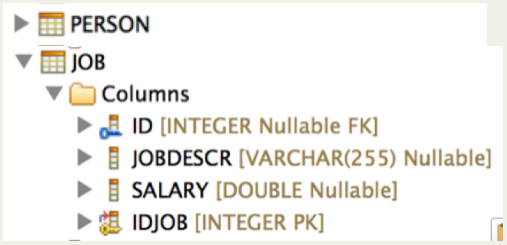
Mapeamento de Relações




```
@Entity
public class Person {

    @Id @GeneratedValue
    private int id;
    private String firstName;
    private String lastName;

    @OneToMany
    private List<Job> jobList;
```

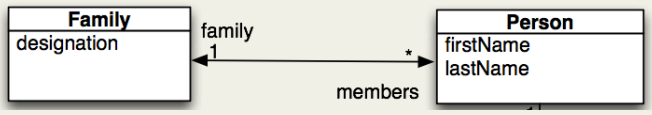



```
@Entity
public class Person {

    @Id @GeneratedValue
    private int id;
    private String firstName;
    private String lastName;

    @OneToMany @JoinColumn
    private List<Job> jobList;
```



Direcionalidade das Relações

- No modelo de objetos, uma relação em que os objetos de A conhecem os objetos de B com que estão relacionados, mas o contrário não acontece, diz-se **unidirecional**
- Uma **relação bidirecional** tem de ser **programada** de forma explícita, nomeadamente com a definição de um atributo em cada uma das classes



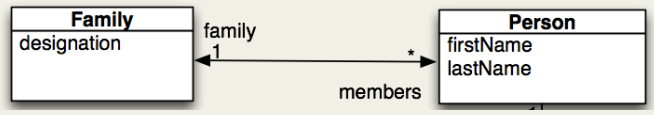
- No modelo relacional as associações são bidirecionais já que são implementadas com chaves estrangeiras, as quais podem ser atravessadas em qualquer direção
- Assim é necessário dar informação adicional
  - se a relação é unidirecional ou bidirecional
  - quem é reponsável por atualizar a relação (de forma a que não seja feita de modo duplicado) — **o dono da relação**

Mapeamento de Relações

JPA:

- Os pares possíveis para a definição de relações bidirecionais são:
  - **OneToOne** <-> **OneToOne**
  - **OneToMany** <-> **ManyToOne**
  - **ManyToMany** <-> **ManyToMany**
- Anotar com **@MappedBy** o atributo que não é dono da relação indicando o nome do atributo que é o dono
- Nas relações **1-para-muitos**, o lado do **ManyToOne** é sempre o dono da relação

```
@OneToMany(mappedBy = "family")
private final List<Person> members
```



Mapeamento de Relações de Herança

- No modelo de objetos, existe a possibilidade de definir uma relação **herança** entre objetos dos tipos **A** e **B**
  - eg. a classe **B** pode estender diretamente a classe **A** e definir mais atributos (entre outras coisas)
- No modelo relacional (puro) não existe a relação de herança e portanto a maioria das bases de dados relacionais não a suporta nativamente
- **Padrões** de mapeamento de relações de herança:
  - ***Single Table Inheritance***: Mapear toda a hierarquia de classes numa **única tabela**
  - ***Class Table Inheritance:*** Mapear **cada classe** numa tabela
  - ***Concrete Table Inheritance***: Mapear cada classe **concreta** numa tabela
- A solução deve ser escolhida tendo em conta as características da hierarquia pois influencia o desempenho (espaço e tempo)

Mapeamento de Relações de Herança: *SINGLE TABLE*

- Mapear toda a hierarquia numa única tabela
  - os atributos introduzidos nas várias classes da hierarquia são todos incluídos nesta tabela
  - esta tabela tem ainda uma coluna que descrimina o tipo do objeto a que o registo se refere, essencial para fazer a criação dos objetos
- Vantagens/Desvantagens:
  - Simples
  - Estável face a mudanças nos donos dos atributos
  - Acesso aos dados é rápido porque estão todos numa tabela
  - Potencial desperdício de espaço na base de dados
  - Tabela pode crescer rapidamente com hierarquias grandes

Mapeamento de Relações de Herança: *JOINED/CLASS TABLE*

- Mapear cada classe da hierarquia numa tabela diferente
  - esta tabela só tem colunas correspondentes aos atributos introduzidos na respetiva classe (ou seja, não tem para os atributos herdados)
  - mas tem coluna que permite encontrar na tabela correspondente à superclasse o resto da informação
- Vantagens/Desvantagens:
  - Fácil de entender porque mapeamento é 1-1
  - Não há espaço desperdiçado nas tabelas
  - Muito sensível a mudanças nos donos dos atributos
  - Acesso mais lento pois é sempre necessário fazer *joins* para se reconstruir a informação de um objeto

Mapeamento de Herança: *TABLE PER (CONCRETE) CLASS*

- Mapear cada classe **concreta** da hierarquia numa tabela diferente
  - esta tabela tem colunas correspondentes aos atributos introduzidos na respetiva classe e os atributos herdados
- Vantagens/Desvantagens:
  - Não há espaço desperdiçado nas tabelas
  - Não é necessário fazer *joins* para se reconstruir objetos de uma subclasse
  - Problemas com chaves
  - Suporte fraco para relações polimórficas com classes da hierarquia que não são folhas
  - Exige fazer interrogações *union* ou interrogações separadas para diferentes classes

Mapeamento de Relações de Herança

- **JPA:**
  - Anotar todas as classes com **@Entity**
  - Colocar o atributo com a chave na classe raiz da hierarquia e anotar com

    @Inheritance(strategy=…)

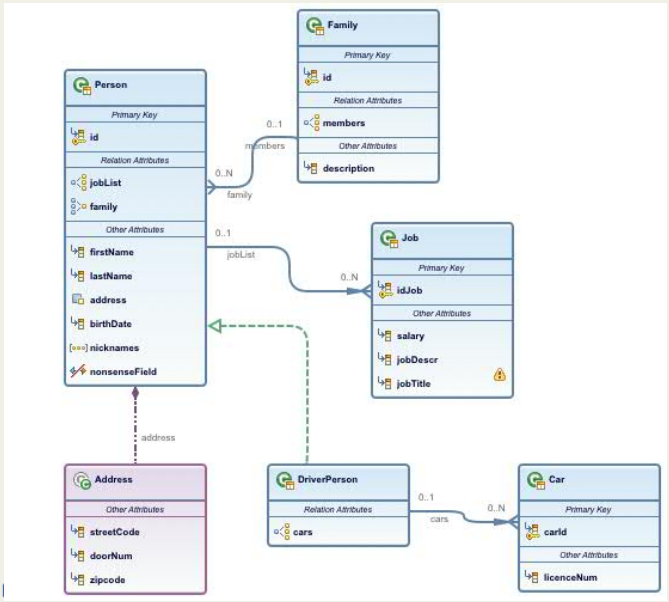
    escolhendo a estratégia apropriada
    - **@InheritanceType.SINGLE_TABLE**
    - **@InheritanceType.JOINED**
    - **@InheritanceType.TABLE_PER_CLASS**
  - Por omissão do **@Inheritance** é usado **SINGLE_TABLE**
  - O suporte à estratégia **TABLE_PER_CLASS** é opcional pelo que nem todas as implementações a suportam

Outros Aspetos Estruturais do Mapeamento com JPA

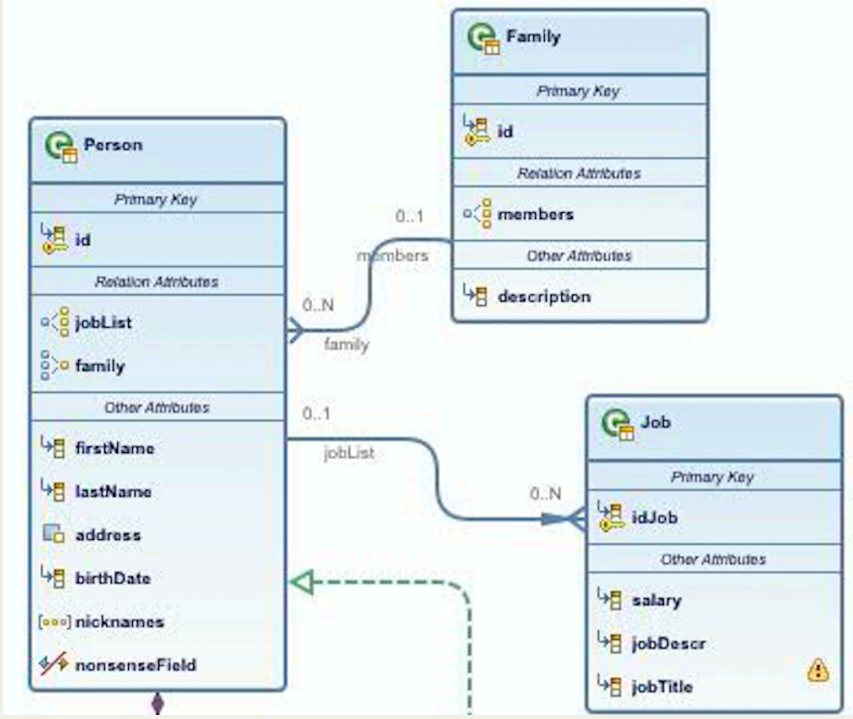
- E se necessitarmos indicar algumas restrições da coluna da DB associadas ao atributo?
  - @Column(name="name", nullable=false, length=512)
- E se não quisermos persistir uma propriedade ou relação ?
  - Por omissão os atributos são persistidos, se não necessitarem de meta-informação adicional
  - Anotar o atributo com **@Transient** para evitar persistência
- E persistir mapas?
  - **@MapKeyEnumerated** e **@MapKeyTemporal** para definir mapas com chaves *Enums* e *Date/Calendar*
- E persistir atributos como *large object (CLOB/BLOB)* ?
  - **@Lob** – serve para tipos baseados em caracteres (como Strings) ou binários (como byte[])

Aspetos Estruturais do Mapeamento: Diagramas

- Podem ser usados diagramas para representar e visualizar as entidades e relações definidas por uma solução de desenho em que o ORM é realizado recorrendo ao JPA

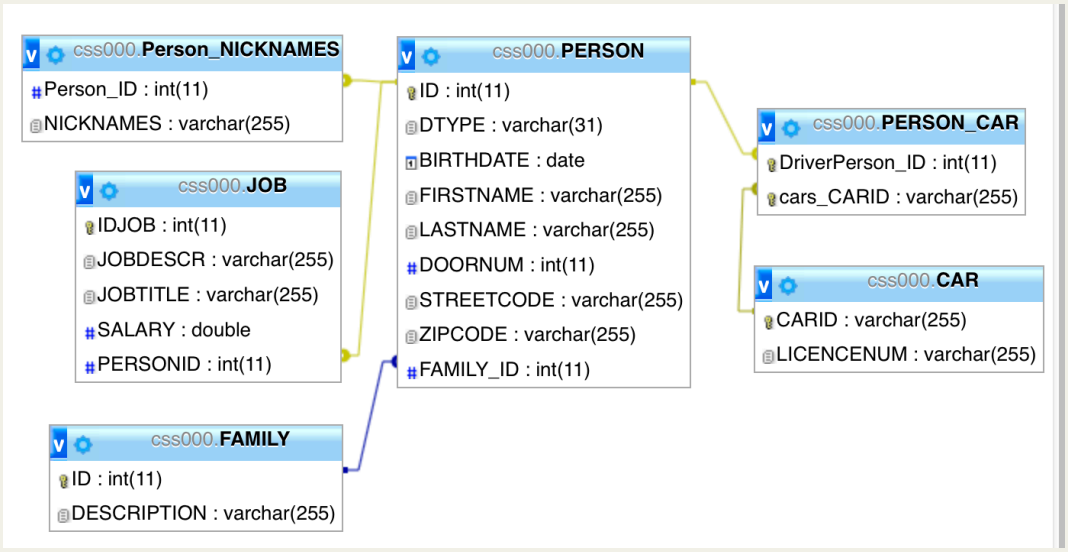


Aspetos Estruturais do Mapeamento: Diagramas



Aspetos Estruturais do Mapeamento: Diagramas

Estes diagramas tendem a ser mais úteis do que os diagramas só com os modelos de dados



Aspetos Comportamentais do Mapeamento com JPA

- Até agora vimos como, em **tempo de desenho**, pode ser definido o mapeamento entre as classes Java e suas relações em tabelas da base de dados através de meta-dados de persistência

- Falta ver como, em **tempo de execução**, é realizada a "sincronização" entre os dois mundos
  - carregar objetos
  - guardá-los na base de dados

Aspetos Comportamentais do Mapeamento com JPA

Em particular, vamos ver como o JPA

- permite ao programador **criar, aceder e modificar** as entradas de uma base de dados relacional como se fossem simples objetos
- permite definir **caraterísticas que afetam o carregamento** e semântica das operações sobre entidades
- permite lidar e gerir a **concorrência**
- oferece uma linguagem de interrogações tipo SQL — **JPQL** que lida diretamente com objetos

Padrão *Unit of Work*

- Fazer refletir cada pequena mudança no modelo de objetos imediatamente na base de dados é pouco eficiente…
- Assim, à medida que se vão carregando objetos da base de dados e os vamos modificando, é preciso manter informação relativamente ao que se mudou para garantir que esses dados são escritos na base de dados. O mesmo se passa com as inserções e remoções.

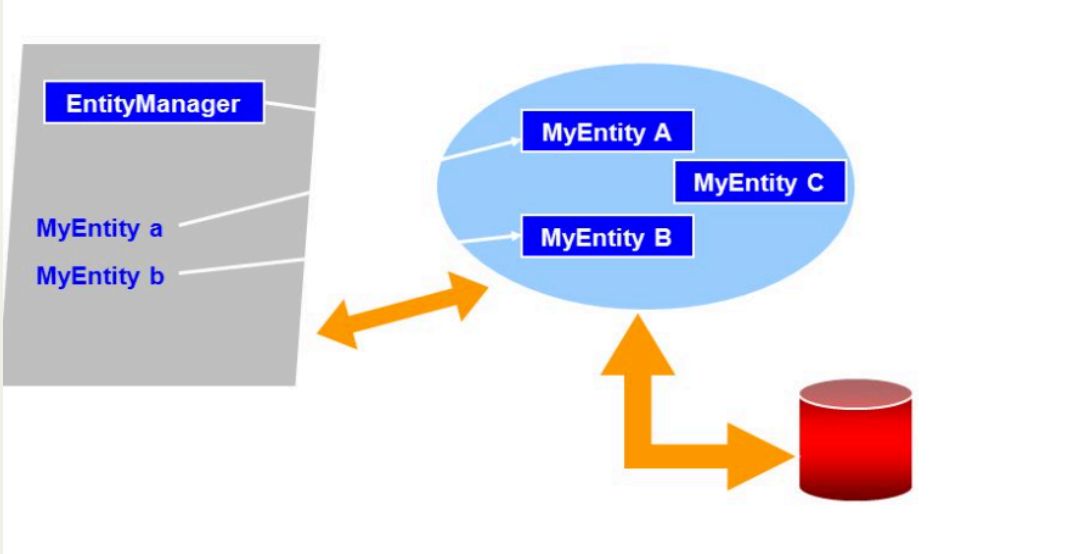
- **Solução:**
  - Ter um objeto que representa a **Unidade de Trabalho** que mantém uma lista de **objetos afetados por uma transação de negócio**, coordena a escrita das mudanças na base de dados e a resolução de problemas de concorrência.
  - Com métodos para **registar** novos objetos, objetos apagados, objetos modificados (sujos) e fazer ***commit***

Padrão *Identity Map*

- À medida que se vão carregando objetos em memória é preciso ter em atenção que não se carrega **o mesmo objeto duas vezes**, pois isso permite alterações inconsistentes de dois objetos que representam o mesmo registo de uma tabela
- Carregar os mesmos dados para diferentes objetos seria também pouco eficiente.

- **Solução:**
  - Ter um **Mapa de Identidades** que guarda os objetos que foram lidos da base de dados numa transação de negócio. Sempre que é preciso um objeto, primeiro é procurado no mapa se o objeto não se encontra já carregado.
  - Método para procurar objeto por chave

JPA: Entity Manager & Persistent Context

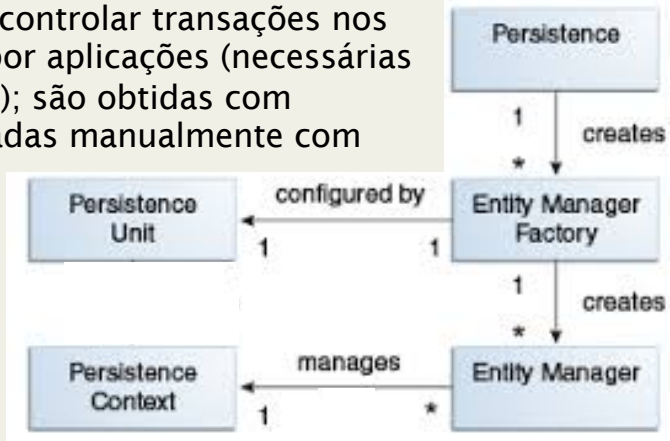


EntityManager: usado numa única transação de negócio e única unidade de trabalho; descartado depois de usado.

Aspetos Comportamentais do Mapeamento com JPA

Principais interfaces da API do JPA que permitem interagir com as entidades

- **EntityManagerFactory** – Permitem construir *EntityManagers* para uma **unidade de persistência**
- **EntityManager** – Permitem gerir entidades, interagindo com um **contexto de persistência**; podem ser geridos pelas aplicações (***non-managed***) ou por *containers* (***container-managed***)
- **EntityTransaction** – Permitem controlar transações nos gestores de entidades geridos por aplicações (necessárias para as comunicações com a bd); são obtidas com **em.getTransaction()** e demarcadas manualmente com **begin()** e **commit()**



Entity Manager

- Gere um conjunto de entidades, as quais são obtidas e manipuladas através de operações como
  - procura de entidades

    find(Class entityClass**, Object** key**)**
  - persistência de entidades

    persist(Object entity**)**
  - remoção de entidades

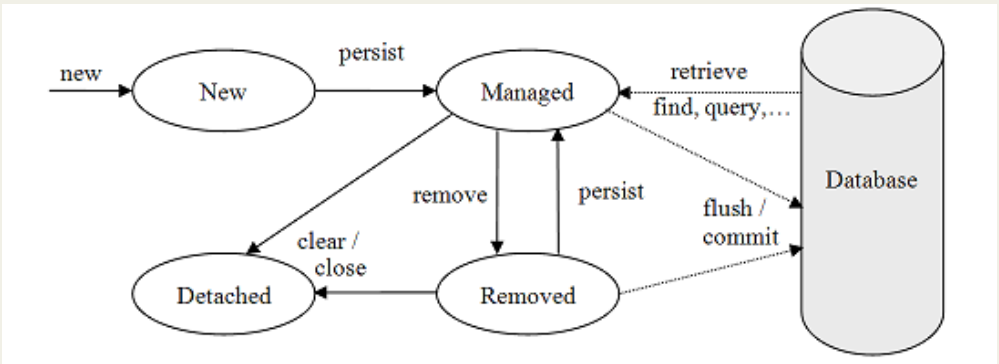
    remove(Object entity**)**
  - *merge* de entidades

    merge(Object entity**)**

  que devem ser realizadas **dentro de uma transação**
- Propaga as mudanças sobre as entidades geridas que foram realizadas dentro de uma transação, na altura em que é feito o **commit** da transação ou quando é invocado o **flush**

EntityManager: Contexto de Persistência

- Associado a um ***EntityManager*** existe um **contexto de persistência**, o qual
  - mantém o conjunto de entidades geridas pelo gestor, em que a identidade (chave) é usada para gerir a unicidade no conjunto
  - funciona como uma "cache" local de entidades geridas pelo *Entity Manager*
- As instâncias de uma entidade podem estar em 1 de 4 estados. **Ciclo de vida das entidades**:



Gestão de *Entity Managers*

- A gestão dos *Entity Managers* pode ser feito pela aplicação
- Estes *Entity Managers* são chamados ***non-managed*** (por oposição a ***container-managed***)

- Os *Entity Managers* não podem ser usados num contexto de múltiplos fios de execução (a classe não é *thread safe*)
- O padrão mais comum é arranjar um *Entity Manager* para servir cada pedido de execução de uma *business transaction* (***entitymanager-per-request***)
- Quando se terminar a utilização de um *Entity Manager* este deve ser fechado para os seus recursos serem libertados

- Num contexto em que o gestor de entidades é gerido pela aplicação, usa-se a classe **EntityManagerFactory** (que é *thread safe)* para criar *Entity Managers*

Gestão de *Entity Managers*

- Uma fábrica **EntityManagerFactory** é sempre relativa a uma **unidade de persistência** identificada pelo seu nome
- A **unidade de persistência** define várias configurações, nomeadamente:
  - as classes entidade que são geridas pelos *Entity Manager* da aplicação
  - os dados da ligação à base de dados como *driver*, *user* e *password*
- Porque a criação do **EntityManagerFactory** é tipicamente cara, deve ser feita em tempo de inicialização; a fábrica deve ser fechada apenas no fecho da aplicação.

Transações

- As transações são um elemento fundamental na persitência

  o *Entity Manager* propaga as **mudanças** sobre as entidades geridas que foram realizadas dentro de uma **transação**, na altura em que é feito o ***commit*** da transação

- Tratam-se não de transações de base dados mas de **transações ao nível dos objetos**, i.e., transações em que um conjunto de mudanças realizadas sobre um conjunto de objetos são propagadas para a base de dados **como um todo**.

Resource Local Transactions

- O tipo de transações suportado pelo JPA apropriado quando o gestor de entidades é gerido pela aplicação
- Cada *Entity Manager* tem associado uma transação
- Implementado pelo tipo ***EntityTransaction***

Resource Local Transactions

- O gestor de entidades tem o método **getTransaction()** que permite aceder à transação, uma instância de **EntityTransaction**
- A transação tem de ser demarcada manualmente pelo programador com os métodos **begin()** e **commit()**
- No caso da transação não ser bem sucedida — o que é sinalizado com a **rollbackException** — deverá ser chamado o **rollback()** se a transação ainda estiver activa (é invocado o *clear* sobre o entitymanger, que leva a que os objetos geridos fiquem *detached, e* a transação na base de dados é *rolledback*)

Exemplo: Gestão do *EntityManager*

```
public class CustomerCatalog {

    /**
     * Entity manager factory for accessing the persistence service
     */
    private EntityManagerFactory emf;
```

```
public void addCustomer (int vat, String designation, int phoneNumber, Discount discountType)
        throws ApplicationException {
    EntityManager em = emf.createEntityManager();
    try {
        em.getTransaction().begin();
        Discount mergedDiscountType = em.merge(discountType);
        Customer c = new Customer(vat, designation, phoneNumber, mergedDiscountType);
        em.persist(c);
        em.getTransaction().commit();
    } catch (Exception e) {
        if (em.getTransaction().isActive())
            em.getTransaction().rollback();
        throw new ApplicationException("Error adding customer", e);
    } finally {
        em.close();
    }
}
```

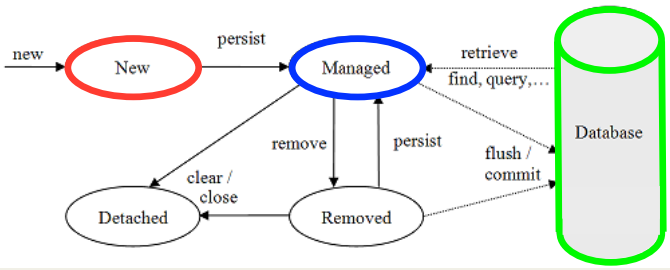

Exemplo: Gestão de Transações

```
public class CustomerCatalog {

    /**
     * Entity manager factory for accessing the persistence service
     */
    private EntityManagerFactory emf;
```

```
public void addCustomer (int vat, String designation, int phoneNumber, Discount discountType)
        throws ApplicationException {
    EntityManager em = emf.createEntityManager();
    try {
        em.getTransaction().begin();
        Discount mergedDiscountType = em.merge(discountType);
        Customer c = new Customer(vat, designation, phoneNumber, mergedDiscountType);
        em.persist(c);
        em.getTransaction().commit();
    } catch (Exception e) {
        if (em.getTransaction().isActive())
            em.getTransaction().rollback();
        throw new ApplicationException("Error adding customer", e);
    } finally {
        em.close();
    }
}
```

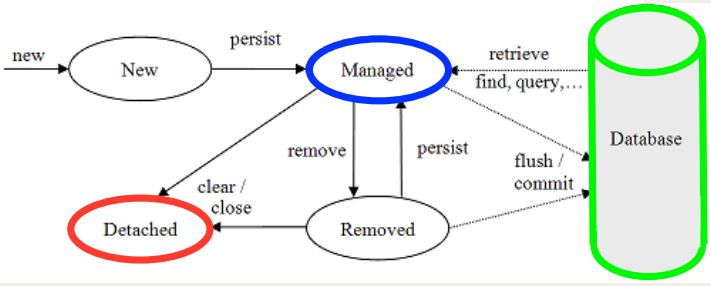
Exemplo: Ciclo de Vida das Entidades

```
public void addCustomer (int vat, String designation, int phoneNumber, Discount discoun
        throws ApplicationException {
    EntityManager em = emf.createEntityManager();
    try {
        em.getTransaction().begin();
        Discount mergedDiscountType = em.merge(discountType);
        Customer c = new Customer(vat, designation, phoneNumber, mergedDiscountType);
        em.persist(c);
        em.getTransaction().commit();
    } catch (Exception e) {
        if (em.getTransaction().isActive())
            em.getTransaction().rollback();
        throw new ApplicationException("Error adding customer", e);
    } finally {
        em.close();
    }
}
```



Exemplo: Ciclo de Vida das Entidades

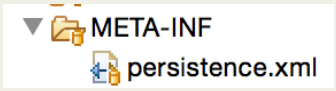
```
public Sale addProductToSale (Sale sale, Product product, double qty)
        throws ApplicationException {
    EntityManager em = emf.createEntityManager();
    try {
        em.getTransaction().begin();
        Sale mergedSale = em.merge(sale);
        mergedSale.addProductToSale(product, qty);
        em.merge(product);
        em.getTransaction().commit();
        return mergedSale;
    } catch (Exception e) {
        if (em.getTransaction().isActive())
            em.getTransaction().rollback();
        throw new ApplicationException("Error adding product to sale", e);
    } finally {
        em.close();
    }
}
```



Exemplo: Gestão do *EntityManagerFactory*

```
public class SaleSys {

    private CustomerService customerService;
    private EntityManagerFactory emf;
    private CustomerCatalog customerCatalog;

    public void run() throws ApplicationException {
        // Connects to the database
        try {
            emf = Persistence.createEntityManagerFactory("domain-model-jpa");
            customerCatalog = new CustomerCatalog(emf);
            customerService = new CustomerService(
                    new AddCustomerHandler(customerCatalog, new DiscountCatalog(emf)));
            // exceptions thrown by JPA are not checked
        } catch (Exception e) {
            throw new ApplicationException("Error connecting database", e);
        }
    }

    public void stopRun() {
        // Closes the database connection
        emf.close();
    }
```


Exemplo: Configuração da *Persistence Unit*



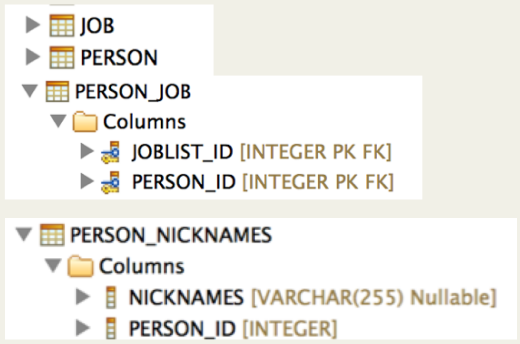

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLS
    <persistence-unit name="domain-model-jpa" transaction-type="RESOURCE_LOCAL">
        <class>business.Customer</class>
        <class>business.Discount</class>
        <class>business.EligibleProductsDiscount</class>
        <class>business.ThresholdPercentageDiscount</class>
        <class>business.NoDiscount</class>
        <class>business.Unit</class>
        <class>business.Product</class>
        <class>business.SaleProduct</class>
        <class>business.Sale</class>
        <shared-cache-mode>NONE</shared-cache-mode>
        <!-- para permitir fazer testes de integração usando o DBSetup -->
        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:derby:data/derby/css000;create=true"/>
            <property name="javax.persistence.jdbc.user" value="css000"/>
            <property name="javax.persistence.jdbc.password" value="css000"/>
            <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
        </properties>
    </persistence-unit>
</persistence>
```

Persistência Transitiva

- O que acontece aos *nicknames* de um objeto *Person* que é apagado?
- E ao Jobs a que está associado?
- E se um objeto *Person* é persistido, são persistidos os seus nicknames?
- E os seus jobs?

```
@ElementCollection
private Set<String> nicknames
```

```
@OneToMany
private List<Job> jobList;
```



Persistência Transitiva

- Quando se lida com grafos de entidades, quando se persiste ou se apaga uma entidade é preciso perceber o que vai acontecer às entidades e dados com que ela está associada
- É muito complexo gerir individualmente as mudanças

- **Solução:**
  - Ter um mecanismo que permita exprimir
    - a propagação transitiva de mudanças nas entidades através de cada relação
    - *if you want an operation to be **cascaded** along an association*

Operações em Cascata

- O JPA permite declarativamente dizer como deve ser realizada a propagação de mudanças das entidades origem para as entidades destino dos arcos do grafo de objetos
- Para cada operação básica do *Entity Manager* — *persist(), merge(), remove(), refresh()* — existe o correspondente *tipo* cascade
- Para dizer que queremos que uma operação seja *cascaded* a uma entidade a que está associada (ou coleção de entidades), basta juntar a correspondente anotação à declaração da associação

  cascade=CascadeType.

  PERSIST, MERGE, REMOVE, DETACH, REFRESH, ALL

- Existe ainda a opção para **orphanRemoval** para **OneToMany** e **OneToOne**

Exemplo: Ciclo de Vida das Entidades

```
public Sale addProductToSale (Sale sale, Product product, double qty)
        throws ApplicationException {
    EntityManager em = emf.createEntityManager();
    try {
        em.getTransaction().begin();
        Sale mergedSale = em.merge(sale);
        mergedSale.addProductToSale(product, qty);
        em.merge(product);
        em.getTransaction().commit();
        return mergedSale;
    } catch (Exception e) {
        if (em.getTransaction().isActive())
            em.getTransaction().rollback();
        throw new ApplicationException("Error adding product to sale", e);
    } finally {
        em.close();
    }
}
```

Exemplo: Ciclo de Vida das Entidades

```
/**
 * The products of the sale
 */
@OneToMany(cascade = ALL) @JoinColumn
private List<SaleProduct> saleProducts;
```

```
public void addProductToSale(Product product, double qty)
        throws ApplicationException {
    if (!isOpen())
        throw new ApplicationException("Cannot add products to a closed sale.");

    // if there is enough stock
    if (product.getQty() >= qty) {
        // adds product to sale
        product.setQty(product.getQty() - qty);
        saleProducts.add(new SaleProduct(product, qty));
    } else
        throw new ApplicationException("Product " + product.getProdCod() + " has stock
                        product.getQty() + ") which is insuficient for the current sal
}
```


Lazy Load

- É importante ter a possibilidade de carregar os dados de uma forma ***lazy***, i.e., adiar o seu carregamento até ao momento em que sejam necessários
- O JPA permite definir se queremos ou não adiar o carregamento da informação sobre as associações de um objeto através da anotação **fetch=FetchType.LAZY** e **fetch=FetchType.EAGER**
- No caso de ser adiada, é apenas até a ligação ser acedida. Para isso é preciso que na altura que é feito o acesso o objeto esteja *managed*. Caso contrário, o atributo poderá estar a *null*. Atenção que acessos no *toString* não contam.

Controlo de Concorrência

- Na presença de múltiplas transações a modificarem uma entidade JPA é preciso usar algum mecanismo de controlo de concorrência para manter a consistência dos dados

- **Exemplo**:
  1. A transação 1 carrega os dados da entidade
  2. A transação 2 altera os dados e faz *commit*
  3. A transação 1 altera os dados com base no que tinha lido (dados entretanto tornados obsoletos)

Locking

- O ***locking*** é uma técnica para lidar com a concorrência que pode ser usada para gerir a concorrência nas transações

- Há geralmente duas estratégias de ***locking***
  - **Otimista**
    - assume que os conflitos são pouco frequentes e dá às transações liberdade para prosseguir em paralelo
    - deteta e previne conflitos, verificando que não foram feitas modificações desde que os dados foram lidos
  - **Pessimista**
    - assume que as transações vão colidir frequentemente
    - a transação que precisa dos dados coloca-lhes um *lock* que só é libertado quando é feito o *commit*

JPA: *Version-based Concurrency Control*

- O JPA implementa o conceito de ***version-based concurrency control***, que é uma forma de ***optimistic locking***
- ***Version-based Concurrency Control***
  - Pode se definir que uma entidade tem versões (***versioned entity***) incluindo na entidade um atributo simples (tipo numérico ou *timestamp*) anotado com **@Version**
  - É a implementação do JPA que trata de gerar *updates* que concretizam a atualização deste atributo e tratam da verificação da existência de alterações concorrentes na entidade
  - Quando é detetado um conflito — por exemplo quando é invocado o **commit** ou **flush** que envolvem alterações a uma entidade alterada — é levantada a exceção **javax.persistence.OptimisticLockException**
- A utilização de outras estratégias de *locking* pode ser necessária como complemento à utilização de entidades com versões

JPA: *Version-based Concurrency Control*

- Versões nas linhas das tabelas é muito mais fácil do que versões em objetos...
  - Quando é considerado que uma entidade muda?
  - O que acontece se alterar as associações do objeto?
- O JPA define quando a **versão** de uma entidade **é alterada**

The version attribute is updated by the persistence provider runtime when the object is written to the database.

All non-relationship fields and properties and all relationships owned by the entity are included in version checks This includes owned relationships maintained in join tables.

JPA: *Version-based Concurrency Control*

- Exemplo 1:
  1. A transação 1 muda qual a entidade y que está associada a x
  2. A transação 2 carrega a entidade x, acede a y e lê os seus dados
  3. A transação 1 faz *commit*
  4. A transação 2 altera um atributo de x com base nos dados de y que tinha lido (dados entretanto tornados obsoletos) e faz *commit*

  Diferentes cenários:

  X -> Y X *->Y vs X ->* Y

JPA: *Version-based Concurrency Control*

- Exemplo 2:
  1. A transação 1 carrega uma entidade y que está associada a x e altera os seus dados
  2. A transação 2 carrega a entidade x, acede a y e lê os seus dados
  3. A transação 1 faz *commit*
  4. A transação 2 altera um atributo de x com base nos dados de y que tinha lido (dados entretanto tornados obsoletos) e faz *commit*

  Cenários:

  X -> Y X *->Y X ->* Y

- Alteração da versão não é suficiente neste caso. Tem de se recorrer a mecanismos adicionais, nomeadamente uso de *locks*
  - eg, na transação 2 usar um *lock* optimista sobre o y lido de forma a prevenir persistir alterações do atributo de x baseada num valor obsoleto do y

JPA: *Locking*

- O JPA também permite aplicar *locks* de várias formas
  - *EntityManager*: *lock(object,mode), find(...,mode), refresh(...,mode)*
  - *Queries: setLockMode(mode)*

- Estão definidos diferentes níveis de estratégias de *locking*
  - **OPTIMISTIC**: *the version number is compared and has to match before the transaction is committed.*
  - **OPTIMISTIC_FORCE_INCREMENT**: *the version number is compared and has to match before the transaction is committed; force a version number increase as well*
  - **PESSIMISTIC_READ**: *apply a database-level read lock when the lock operation is requested: roughly concurrent readers are allowed but no writer is allowed.*
  - **PESSIMISTIC_WRITE:** *apply a database-level write lock when the lock operation is requested: roughly no reader nor writer is allowed.*

JPQL

- **Java Persistence Query Language**, uma linguagem para escrever interrogações, que é oferecida pelo JPA
- As interrogações envolvem as referências para as entidades e os seus atributos persistidos, pelo que é vista como uma **versão OO do SQL**

- Permite abstrair das variações na sintaxe SQL aceite por diferentes SGBD e com o facto de as objetos poderem ser persistidos em bases de dados com esquemas diferentes
- Podem ser usadas *Strings* ou fazer a construção dinâmica recorrendo à **JPA Criteria API** que dá mais garantias relativamente à inexistência de erros em tempo de execução

JPQL: Strings vs Criteria API

```
//Strings
criteria = “SELECT a FROM Aluno a WHERE a.numAluno = :num”
```

```
//Criteria API
CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Aluno> criteria = cb.createQuery(Aluno.class);
Root<Aluno> aluno = criteria.from(Aluno.class);
criteria.select(aluno);
criteria.where(cb.equal(aluno.get("numAluno"),
                        cb.parameter(String.class, "num")));
```

```
//make query and get result
List<Aluno> alunos= em.createQuery(criteria).getResultList();
```


JPQL

```
SELECT a FROM Aluno a WHERE a.numAluno = :num

SELECT DISTINCT a FROM Aluno a
       WHERE a.disciplinas IS NOT EMPTY

SELECT DISTINCT a FROM Aluno a, IN(a.disciplinas) AS d
       WHERE d.curso = :curso

SELECT a FROM Aluno a WHERE :d MEMBER OF a.disciplinas
```

Interrogações JPQL

- **Tipos de Interrogações**
  - Suporta interrogações dinâmicas e estáticas através dos tipos **Query**, **TypedQuery** e **NamedQuery**
  - As interrogações do tipo **NamedQuery** são mais eficientes e seguras, pelo que devem ser usadas sempre que possível
    - São suportadas através das anotações **@NamedQuery** e **@NamedQueries**
    - Definidas com parâmetros que têm valor definido programaticamente
- **Organização**
  - Faz sentido definir as **NamedQuery** nas classes que definem as entidades que são alvo da interrogação.
  - Uma outra boa prática é recorrer à definição de constantes para os nomes das interrogações e dos parâmetros que estas definem

Interrogações JPQL

```
@Entity
@NamedQuery(name=Customer.FIND_BY_VAT_NUMBER, query="SELECT c FROM Customer c WHERE c.vatNumber = :" +
        Customer.NUMBER_VAT_NUMBER)
public class Customer {

    // Named query name constants
    public static final String FIND_BY_VAT_NUMBER = "Customer.findByVATNumber";
    public static final String NUMBER_VAT_NUMBER = "vatNumber";
```

```
 */
public Customer getCustomer (int vat) throws ApplicationException {
    EntityManager em = emf.createEntityManager();
    TypedQuery<Customer> query = em.createNamedQuery(Customer.FIND_BY_VAT_NUMBER, Customer.class);
    query.setParameter(Customer.NUMBER_VAT_NUMBER, vat);
    try {
        return query.getSingleResult();
    } catch (PersistenceException e) {
        throw new ApplicationException ("Customer not found.", e);
    } finally {
        em.close();
    }
}
```

Aspetos Comportamentais do Mapeamento: Padrões

- Subjacente aos aspetos comportamentais do ORM com JPA que estudámos estão os seguintes padrões:
  - ***Identity Map*** *controls load of data so taht there are no two objetcs loaded that represent the same table row*
  - ***Unit of Work*** *maintains objects affected by a business transaction and coordinates the changes and the resolution of concurrency problems*
  - ***Lazy Load*** *An object that does not contain all the data that you need but knows how to get it*
  - ***Optimistic Offline Lock*** *validates that the changes about to be commited by one session does not conflict with the changes of another session*
  - ***Query Object*** *it is a structure of objects that can form itself into a SQL query*