Construção de Sistemas de Software
LTI
2022 - 2023

Parte III : Persistência

JDBC: Java & BDs

- A **J**ava **D**ata**b**ase **C**onnectivity (**JDBC**) é uma API que define como o cliente pode aceder e manipular dados com a BD.
- O foco são BDs relacionais o que a torna adequada para lidar com BDs MySQL.
- Estão incluídas no pacote **`java.sql`**.
- Um JDBC driver é uma componente de software que permite a uma aplicação Java interagir com uma BD a partir da API JDBC.

JDBC: Java & BDs

- Existem três actividades principais:

1) Conexão com a base de dados
2) Enviar comandos para a base de dados
3) Recolher e processar os resultados dos comandos enviados

- Classes/Interfaces importantes:

  Classe **`DriverManager`** – gere os drivers JDBC

  Quando o método `getConnection()` é invocado, o `DriverManager` tenta localizar um driver adequado entre os que foram carregados. A seguir estabelece a ligação com a BD referenciada.

```
String connectionUrl =
          "jdbc:mysql://192.168.56.10/cssdb?" +
          "user=css&password=css";
Connection con = DriverManager.getConnection(connectionUrl);
```

JDBC: Java & BDs

- Interface **Statement**
- Define os serviços que executam comandos SQL e retornam os resultados produzidos.
- Interface **ResultSet**
- Uma tabela que representa o resultado do comando SQL enviado.
- Por defeito deve existir um objecto **ResultSet** por cada objecto Statement. Ou seja, se for efectuado outro comando com o mesmo **Statement**, o **ResultSet** anterior é fechado.
- O objeto mantém um cursor para a linha corrente da tabela (no início ele encontra-se antes da 1ª linha). O método **next()** coloca-se na linha seguinte e devolve **true** se tiver sucesso, i.e., se existe essa nova linha.

```
   String    SQL  = "SELECT * FROM clientes";
PreparedStatement stmt = con.prepareStatement(SQL);
ResultSet rs   = stmt.executeQuery();
while (rs.next()) {
    System.out.println( rs.getString("idCliente") + " : " +
                        rs.getString("npc") );
}
```

JDBC: Java & BDs

- **`boolean execute()`** Executes the SQL *statement* in this **`PreparedStatement`** object, which may be any kind of SQL statement.
- **`ResultSet executeQuery()`** Executes the SQL *query* in this **`PreparedStatement`** object and returns the **`ResultSet`** object generated by the query.
- **`int executeUpdate()`** Executes the SQL *statement* in this **`PreparedStatement`** object, which must be an SQL Data Manipulation Language (DML) statement, such as `INSERT`, `UPDATE` or `DELETE`; or an SQL statement that returns `nothing`, such as a statement.

JDBC: Java & BDs

```
String strSQL = "DELETE FROM clientes WHERE idCliente = 2";
int rowsEffected = stmt.executeUpdate(strSQL);
System.out.println(rowsEffected + " rows effected on delete");

String strSQL = "INSERT INTO clientes(idCliente,npc,designacao,telefone," +
                "idDesconto) VALUES (2,123456789,'segundp',217500000,2)";
int rowsEffected = stmt.executeUpdate(strSQL);
strSQL = "UPDATE clientes SET designacao = 'segundo' " +
      "WHERE idCliente = 2";
rowsEffected = stmt.executeUpdate(strSQL);

strSQL  = "SELECT * FROM clientes WHERE idCliente > 1";
PreparedStatement stmt = con.createStatement();
ResultSet rs   = stmt.executeQuery(SQL);
while (rs.next()) {
  System.out.println( rs.getString("idCliente") + " : " +
                      rs.getString("designacao"));
}
rs.close();
stmt.close();
con.close();
```

JDBC: Java & BDs

Interface **PreparedStatement**

- Um objeto que representa um comando SQL pré-compilado.
- Mais eficiente quando é necessário repetir o mesmo comando várias vezes

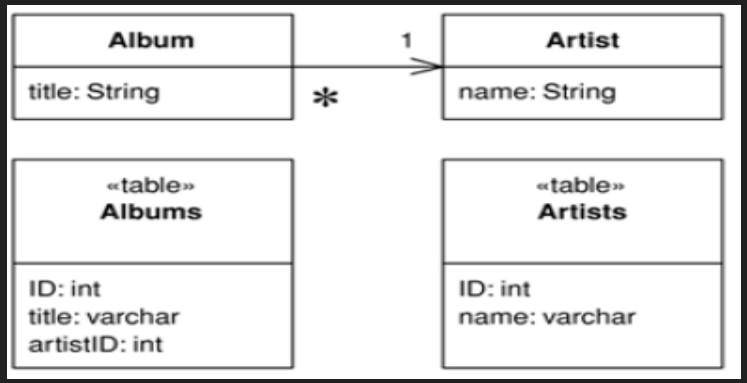
```
String commandSQL = "SELECT * FROM clientes" +
                    "WHERE idCliente > ? AND name = ?";

PreparedStatement preStmt =
                          con.prepareStatement(commandSQL);
preStmt.setInt(1, 1000);
preStmt.setString(2, "Rui");

ResultSet rs = preStmt.executeQuery(SQL);
while (rs.next())
  System.out.println( rs.getString("idCliente");
```

Mapeamento OO-BD Relacional

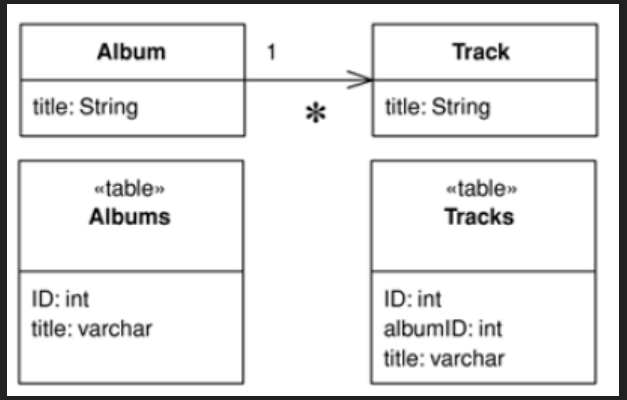
(Fowler, caps.3,12)

- A comunicação com uma BD para a persistência de informação relativa à lógica de domínio é um elemento essencial
  - O uso de BD relacionais com linguagem SQL é quase universal
- Como referido, o mundo OO é conceptualmente distinto do mundo das BD relacionais
- A representação, os tipos de associações, são distintos
- Os objetos guardam referências para outros objetos associados
- As tabelas associam-se através do uso de chaves estrangeiras
- Os objetos facilmente guardam coleções de outros objetos
- A normalização das tabelas exige que as relações entre tabelas sejam representadas por valores únicos
- É necessário definir uma estratégia de mapeamento entre estes dois mundos (ORM)

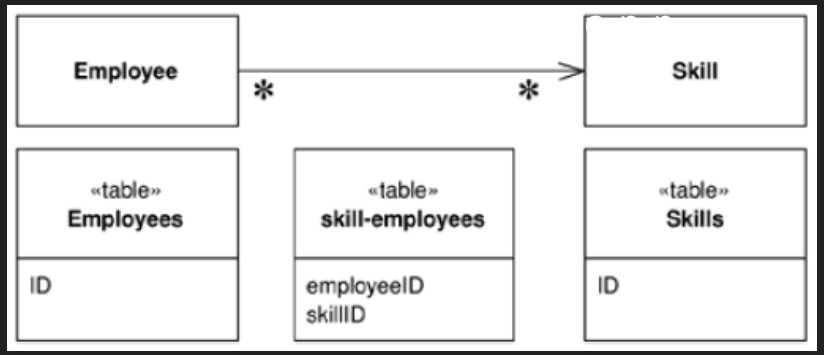
Mapeamento OO-BD Relacional



e.g., relação n-1



e.g., relação 1-n



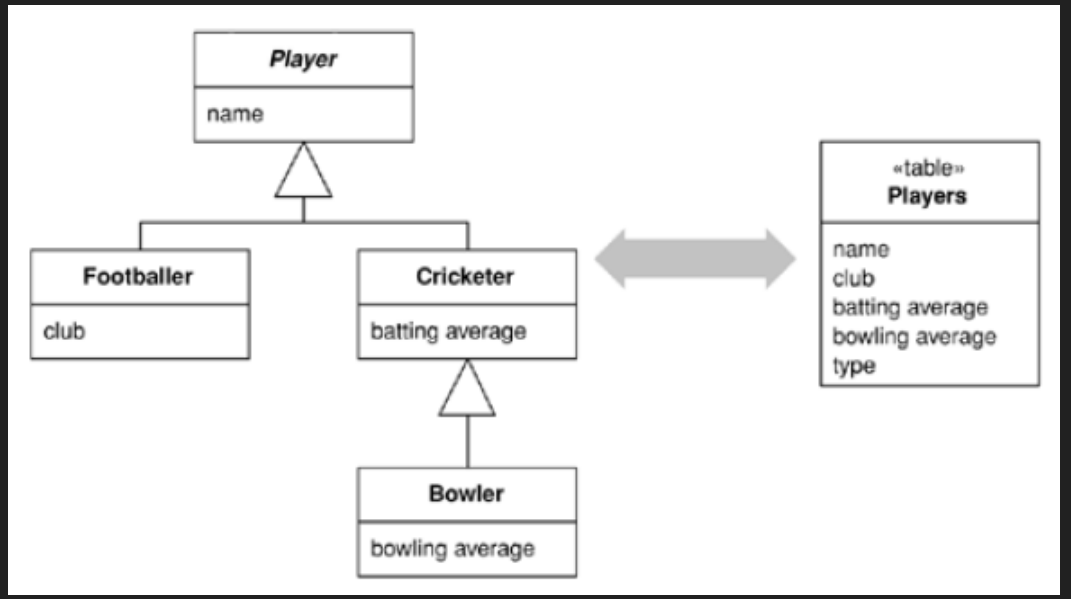
relação n-n

ORM – Herança

- Até agora apenas falámos de relações de associação como a composição. Mas o mundo OO também inclui associações por herança
- As BD relacionais não incluem a noção de herança
- Como fazer?
- A literatura propõe três alternativas que correspondem a três padrões no texto do Fowler:
  - Single Table Inheritance
  - Concrete Table Inheritance
  - Class Table Inheritance

ORM – Single Table Inheritance

- Neste padrão é apenas usada uma tabela para representar todas as instâncias de todas as classes na hierarquia da herança
- Quando se cria um objeto a partir de um registo da tabela é necessário saber a que classe pertence. Para isso existe uma coluna na tabela com essa informação (*type*) cujo valor pode ser o nome da classe

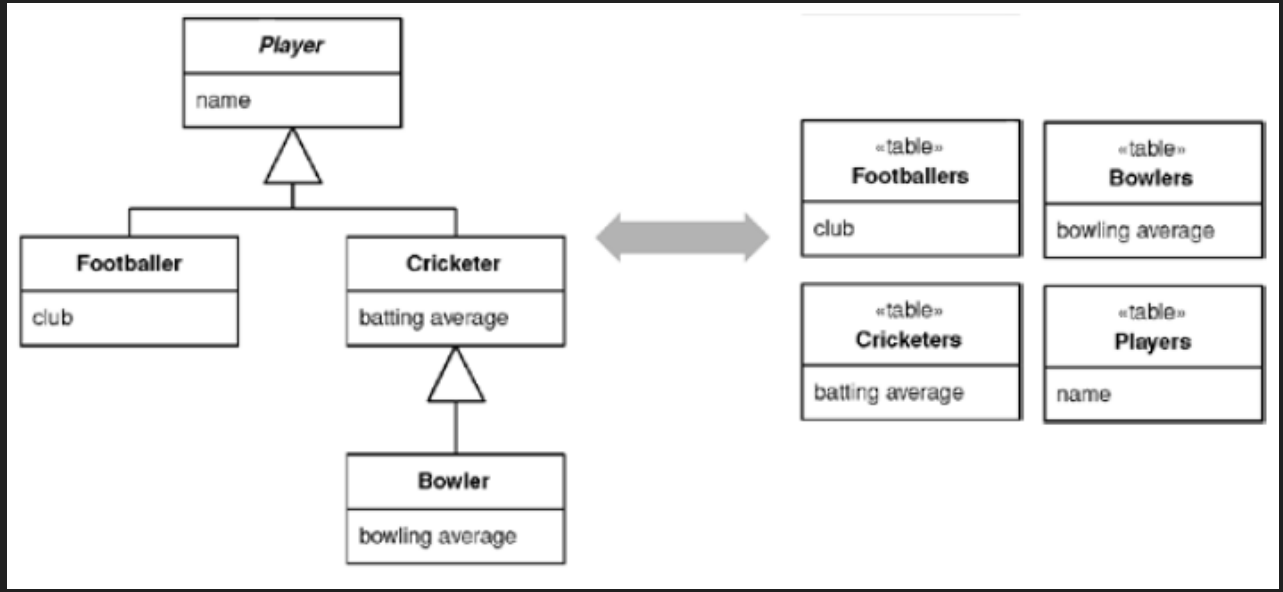


ORM – Single Table Inheritance

- Vantagens:
- Apenas existe uma tabela
- Não é preciso fazer *joins* para recolher os dados
- Um *refactoring* que transfira um atributo entre as classes não altera a BD
- Desvantagens
- Gasto de memória: muitos campos dos registos estarão vazios, cada campo só faz sentido no contexto da classe a que o registo pertence.
- O âmbito dos nomes das classes é comum: não se pode ter atributos com o mesmo nome nas classes (pode resolver-se prefixando o nome dos campos com o nome da respetiva classe)

ORM – Class Table Inheritance

- Neste padrão é usado uma tabela por classe
- Existe a questão de como ligar os diferentes registos nas várias tabelas
  - e.g., na estrutura abaixo um *bowler* terá informação em três tabelas



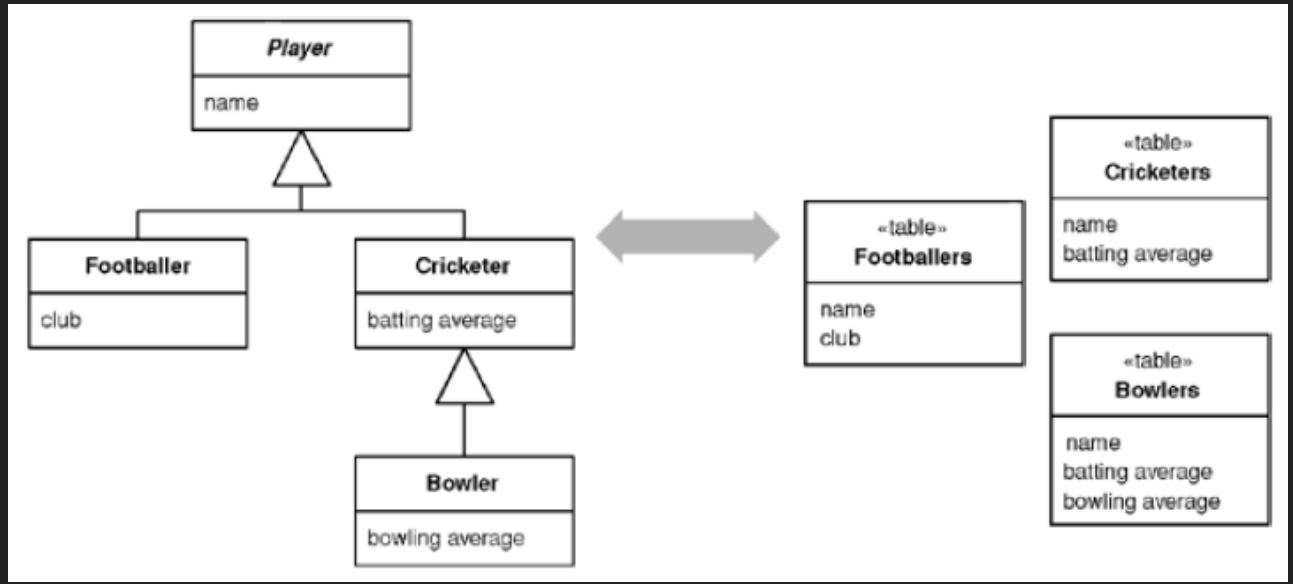
Uma solução é usar a chave da superclasse como chave estrangeira nas restantes tabelas

ORM – Class Table Inheritance

- Vantagens:
- Todas as colunas são relevantes, não há desperdício de memória
- As relações entre classes do modelo de dados refletem-se na BD
- Desvantagens
- Não é fácil juntar toda a informação de forma eficiente. Um *join* de muitas tabelas tende a ser lento. Fazer múltiplos *selects* a todas as tabelas em questão também sobrecarrega o acesso à BD
- A tabela da superclasse irá ser sobrecarregada de acessos
- Se se estiver a construir uma coleção de jogadores (i.e., podem ser de vários desportos) as tabelas a invocar variam de jogador para jogador
- Um *refactoring* que transfira um atributo entre as classes altera a BD

ORM – Concrete Table Inheritance

- Neste padrão é usado uma tabela por classe *concreta*
- As classes abstratas (como a superclasse) não são representadas por tabelas (dado não se declarar objetos de classes abstratas)
- Isto levanta o problema da coerência das chaves entre tabelas



ORM – Concrete Table Inheritance

- Vantagens:
- Cada tabela é auto-contida e não há campos irrelevantes
- Não há a necessidade de *joins*
- Cada tabela só é acedida quando a respetiva classe é acedida, não há *bottlenecks* à partida
- Desvantagens
- A gestão das chaves primárias é um assunto complexo
- Na BD não são expressas as relações com as classes abstratas
- Um *refactoring* que transfira um atributo entre as classes altera a BD
- Se existe uma mudança na superclasse, todas as tabelas têm de ser modificadas
- Um *find* na superclasse força a consulta a todas as tabelas

ORM - Metadata

- Uma abordagem importante a este problema é incluir detalhes em metadata sobre como efetuar o mapeamento
- A metadata, neste contexto, é uma descrição textual das relações entre as classes e as tabelas da BD
- Esta descrição é processada automaticamente, seja em tempo de compilação (gerando código que efetua o mapa descrito) ou em tempo de execução (através de mecanismos de reflexão que escolhem quais os métodos, *queries*, etc. adequados no momento)
- A metadata pode estar num documento próprio, usualmente escrito em formato XML, ou embutido no próprio código.
- Vamos usar uma ferramenta do J2EE, a tecnologia JPA, que aborda o problema da ORM através de metadata embutida no código.

JPA

- Ferramenta do J2EE que providencia ORM para Java
- JPA = **J**ava **P**ersistance **A**PI é composta por três partes:
  - A API (cf. pacote javax.persistence)
  - Uma linguagem para *queries* designada JPQL
  - Um ORM via metadados (uso de anotações)
- Os conceitos do modelo de domínio irão corresponder a tabelas da BD. No JPA isto corresponde a uma entidade
- Uma entidade é uma classe Java cujo estado de cada objecto pode ser armazenado como um registo da respetiva tabela
  - Esta informação bem como a relação entre entidades será descrita por anotações e/ou num ficheiro XML que acompanha a aplicação
- Existem várias implementações:
  - ObjectDB, OpenJPA (desenvolvido pela Apache), EclipseLink (desenvolvido pelo Eclipse e a que vamos usar), ...

Arquitectura JPA

- **EntityManager** é uma interface que providencia serviços para interagir com as entidades (e.g., *persist*, *merge*, *remove* que gravam, alteram e apagam entidades da BD).
  - Corresponde a uma ligação à BD. Uma aplicação que precisa de várias ligações, usa várias instâncias desta classe.
  - Serve de fábrica à criação de transações e *queries* (cf. abaixo).
- **EntityManagerFactory** cria instâncias de EntityManagers à medida das necessidades.
  - Ela é parametrizada para lidar com uma BD específica.
  - Normalmente há apenas uma instância por cada BD usada pela aplicação.

- **EntityTransaction** – interface que gere as transações. É preciso uma transação activa para inserir/remover dados na BD.
- **Query** – instâncias que permitem fazer *queries* à BD.

EntityManagerFactory

- Para criar um EntityManagerFactory é necessário fornecer os dados sobre a BD e definir que classes são entidades, i.e., definir uma Persistence Unit (no ficheiro **persistence.xml**)

como invocar no código

```
EntityManagerFactory emf =
       Persistence.createEntityManagerFactory("domain-model-jpa");
```

---

*(ficheiro **persistence.xml**)*

```
<persistence ...>

        <persistence-unit name="domain-model-jpa" transaction-type="RESOURCE_LOCAL">
  <class>domain.Cliente</class>
          <class>domain.Desconto</class>
          ...
          <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>
```

Classes que representem entidades

Definição da conexão a BD

EntityManagerFactory

Onde criar o EntityManagerFactory ?

- Na classe principal, como atributo de classe:

```
public class Main {
       public final static EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("domain-model-jpa");
```

- Num bloco static. Vantagem: pode-se acrescentar instruções para verificar se a conexão foi establecida:

```
public class Main {
   public static EntityManagerFactory emf = null;
   static {
      try {
         emf = Persistence.createEntityManagerFactory("domain-model-jpa");
      } catch (PersistenceException e) {
         throw new ApplicationException("Could not create DB connection. [[" +
                                        e.getMessage() + "]]");
      }
   }
```

`ApplicationException` deve ser uma subclass de `RuntimeException`.

EntityManager

O EntityManager representa uma conexão à BD.

```
EntityManager em = emf.createEntityManager();

try {
  // Use the EntityManager to access the
database
}
finally {
  em.close();
}
```

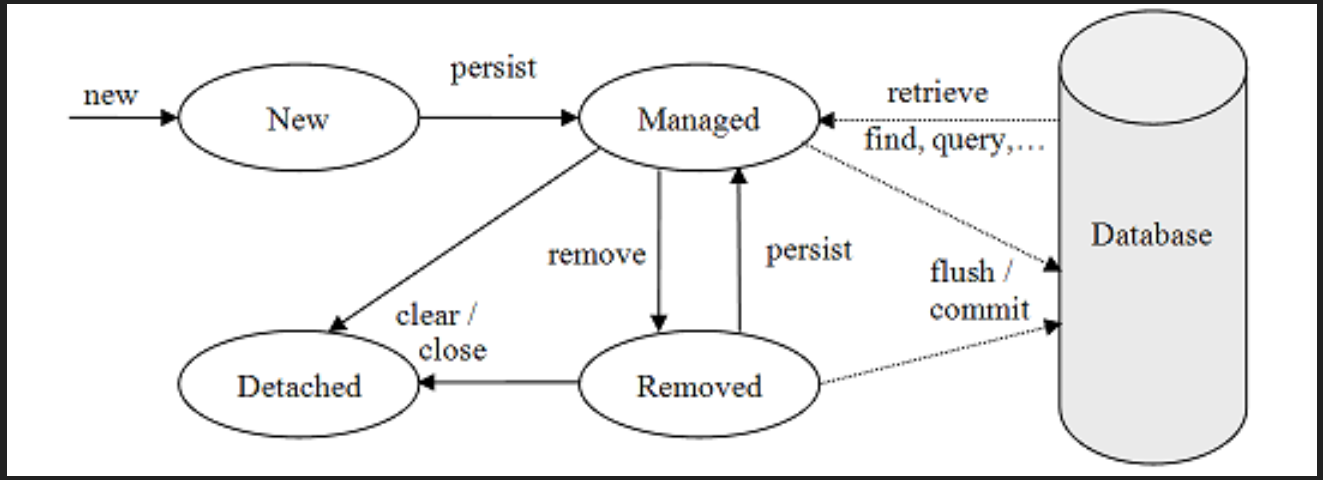
EntityTransaction

- Uma instância do EntityTransaction representa a conexão activa à BD através da qual realizamos operações que mudam as tabelas dessa BD
  - É iniciada pelo método **begin()** que activa uma nova transição sendo esta terminada com **commit()**, onde as alterações pedidas são efetuadas na DB
  - Se se usar **rollback()** desfaz-se as alterações e deixa-se tudo como antes na BD

```
try {
    em.getTransaction().begin();
    Cliente novoClient = new Cliente(...); // Client must be an entity
    em.persist(novoCliente);    // object will be saved in the next commit
    em.getTransaction().commit();
} catch (Exception e) {         // some unforseen problem just happened!
    if (em.getTransaction().isActive())  // is transaction still active?
        em.getTransaction().rollback();    // if so, then undo
} finally {
    em.close();
}
```

Os métodos persist, merge, remove e refresh **DEVEM** ser invocados dentro de um contexto de transação. Caso contrário: TransactionRequiredException

Ciclo de vida de uma Entidade

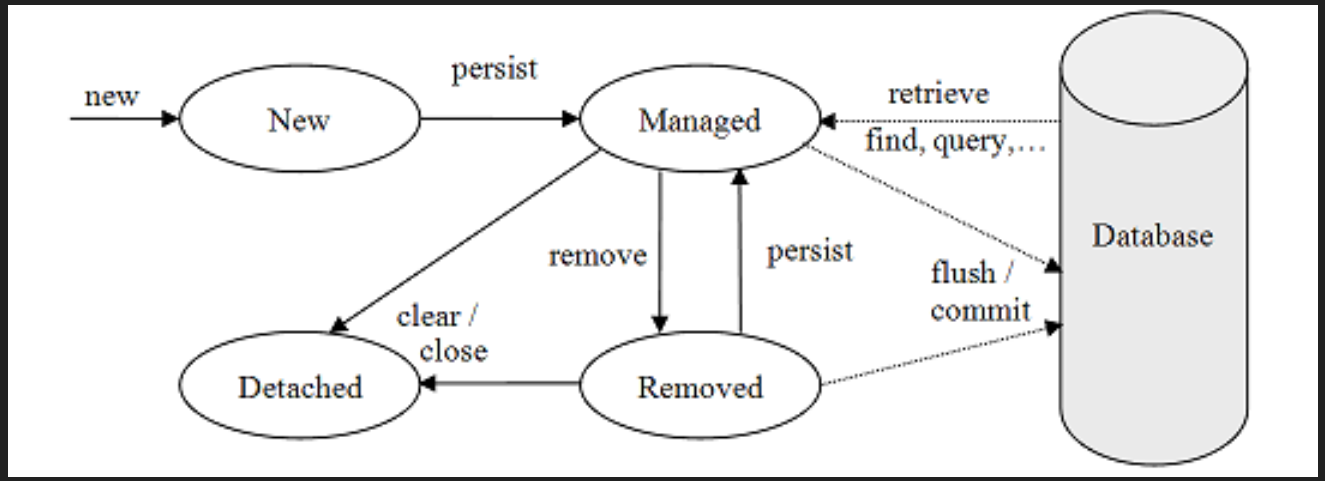


A entidade e quando criada não possui relação com o EntityManager

- Quando ocorre **`em.persist(e)`** durante uma transação ativa, a entidade passa ao estado managed
- Igualmente, se uma entidade é recolhida da BD (e.g., por uma *query*) ela também fica neste estado

Qualquer alteração do estado da entidade é observada pelo EntityManager, e será armazenada no próximo **`commit()`**

Ciclo de vida de uma Entidade



Uma entidade marcada para apagar muda de estado para `removed`, mas só será apagada da BD após o `commit` seguinte.

- Se o **EntityManager** for fechado, as entidades por ele geridas passam para o estado detached.
- - Existe uma forma explícita via o método **`em.detach(obj)`**
- – É possível reverter, i.e., voltar a estar managed com **`em.merge(obj)`**

Contexto de Persistência

- O contexto de persistência é a coleção de todos objetos geridos (*managed entities*) pelo respetivo EntityManager (EM)
- Se for pedido um objeto que já está no EM, ele é devolvido e poupa-se uma operação sobre a BD (funciona como *cache*)
- Toda esta gestão é automática e o EM certifica-se que o estado do contexto mantém-se coerente
- Pode-se verificar se um objeto pertence ao contexto

```
boolean isManaged = em.contains(cliente);
```

- E pode-se igualmente limpar o contexto: em.clear()

neste caso, todos os objetos lá armazenados passam ao estado detached.

Gestão da Memória

- O contexto de persistência tem memória limitada o que pode levantar problemas em transações muito grandes
- Para colmatar este ponto pode-se utilizar uma combinação do comando **`flush()`** e **`clear()`**
  - **`flush()`** sincroniza o estado da BD com o contexto de persistência
  - **`clear()`** limpa o contexto de persistência, todas as entidades *managed* passam a *detached* (sem *commit* ou *flush* prévio, não persistem na BD)

```
em.getTransaction().begin();
    for (int i = 1; i <= 1000000; i++) {
        Point point = new Point(i, i);
        em.persist(point);
        if ((i % 10000) == 0) {  // salva & limpa a cada 10 mil pontos
            em.flush();
            em.clear();
        }
    }
    em.getTransaction().commit();
```

Gestão da Memória

Uma alternativa a transações muito grandes, é a aplicação de múltiplas transações (esta é considerada uma solução melhor):

```
em.getTransaction().begin();
for (int i = 1; i <= 1000000; i++) {
    Point point = new Point(i, i);
    em.persist(point);
    if ((i % 10000) == 0) {
        em.getTransaction().commit();
        em.clear();
        em.getTransaction().begin();
    }
}
em.getTransaction().commit();
```

Entidades JPA

- Relembrando: uma entidade é uma classe Java cujo estado de cada objecto irá ser armazenado como um registo na respetiva tabela
- Uma classe entidade tem de cumprir o seguinte:
  - Tem de ser anotada com Entity

```
@Entity
public class Cliente { ... }
```

  - Tem de ter um construtor sem argumentos
  - A classe não pode ser final
  - Os métodos e atributos anotados também não podem ser finais
  - Atributos persistentes não podem ser públicos
- Atributos como coleções, mapas e *arrays* podem ser persistidos

Entidades JPA

Anotação Entity aceita um parâmetro name:

```
@Entity(name="CLIENTE")
public class Cliente { ... }
```

- Na ausência do parâmetro **name**, o nome da **entidade** é igual ao nome da **classe**.
- O nome da entidade é o nome usado nas queries **JPQL**
- A classe, a entidade e tabela da base de dados podem ter nomes diferentes.
- O parâmetro name na anotação **Entity** permite especificar um nome para a entidade diferente do nome da classe.
- A anotação **@Table** permite especificar o nome usado para a tabela da BD. Ex.: **@Table(name="CLIENT")**

- **Pode parecer desnecessáriamente complexo ! Essas variantes podem ser uteis para lidar com várias implementações do JPA e/ou da BD.**

Entidades e chaves

- Uma entidade tem um identificador único que irá corresponder à chave da respetiva tabela.
  - Se a chave é simples (tipos primitivos ou *wrapper classes*, String, java.util.Date ou sql.Date) tem de ser anotada com **@Id** :

```
@Entity
public class Cliente {
    @Id
    private int idCliente;
}
```

```
@Entity
public class Cliente {
    @Id @GeneratedValue(strategy=AUTO)
    private int idCliente;
}
```

- É preferível automatizar a geração dos valores da chave através da anotação **@GeneratedValue**
  - O valor **AUTO** informa que existe um contador para a BD inteira. Nenhum valor de chave é repetido mesmo em tabelas diferentes (valor *default*).

Entidades e chaves

O JPA permite igualmente obter a chave de uma dada entidade

```
PersistenceUnitUtil util = emf.getPersistenceUnitUtil();

Object clienteId = util.getIdentifier(cliente);
```

Se devolver null a entidade ainda não foi armazenada na BD

Classes Embebidas

Se a chave é complexa, a chave deve corresponder a uma classe (anotada com **@Embeddable**) e é usada na entidade com a anotação **@EmbeddedId**

```
@Embeddable
public class ClientName
       implements Serializable {

  @Column(name="FIRSTNAME", nullable=false)
  private String firstname;

  @Column(name="LASTNAME", nullable=false)
  private String lastname;
```

```
@Entity
public class Client
       implements Serializable {

  @EmbeddedId
  protected ClientName client;
```

Classes Embebidas

```java
@Embeddable
public class ClientName
        implements Serializable {

  @Column(name = "FIRSTNAME", nullable = false)
  private String firstname;

  @Column(name = "LASTNAME", nullable = false)
  private String lastname;
```

```java
@Entity
public class Client
        implements Serializable {

  @EmbeddedId
  protected ClientName client;
```

- A classe da chave deve ser pública
- O atributo que corresponde à chave deve ser público ou protegido
- As classes devem serializáveis e implementar os métodos hashcode() e equals(Object) e um construtor sem argumentos
- Uma classe embebida não resulta numa tabela (não precisa de chave) sendo guardada diretamente na classe que a embebe

Classes Embebidas

- Por exemplo

```
ClientName clName = new ClientName("João", "Sousa");
Cliente cliente = new Cliente(clName);

em.getTransaction().begin();
em.persist(cliente);
em.getTransaction().commit();
```

- O objeto **`ClientName`** é armazenado na tabela **`Cliente`** na codificação obtida após a serialização
- Um objeto embedido não pode ser partilhado por outros registos da tabela (mesmo se não forem chaves)
- Não é necessário fazer-se o *persist* explícito dos objetos embebidos
- Se a classe **`ClientName`** não fosse embebida, ter-se-ia que persisti-la senão seria lançada a excepção **`IllegalStateException`**

Anotação de atributos

- Um atributo sem anotação é considerado como `@Basic` (anotação *default*)
  - Existe a opção `fetch` que indica ao sistema se o valor deve ser carregado de imediato (EAGER, valor default) ou pode ser carregado apenas quando necessário (LAZY).

```
@Basic(fetch=LAZY)
protected String name;
```

- Para coleções o valor default é LAZY:

```
@Entity
class Employee {
  @ManyToMany(fetch=EAGER)
  private Collection<Project> projects;
}
```

Anotação de atributos

- Cada atributo, em princípio, corresponderá a uma coluna na tabela
- Com a anotação **@Column** podemos dar informação específica sobre determinados atributos

```
@Column(nullable = false, unique = true)
private int npc;
```

- Pode-se definir o nome da coluna (**name=…**), a sua dimensão (**length=…**), entre outras informações

Relações entre Entidades

O JPA define quatro tipos de multiplicidade entre as entidades A e B

- **@OneToOne** – cada instância de A está associada a uma instância de B
- **@OneToMany** – uma instância de A pode estar relacionada com múltiplas instâncias de B
- **@ManyToOne** – múltiplas instâncias de A podem estar relacionadas com a mesma instância de B
- **@ManyToMany** – múltiplas instâncias de A podem estar relacionadas com múltiplas instâncias de B

```
@Entity
public class Cliente {

 @OneToOne
 private Desconto desconto;
...
```

```
@Entity
public abstract class Desconto {

  @Id
  private int tipoDesconto;
...
```

Relações entre Entidades

As relações podem ser uni ou bidirecionais.

- Uma relação entre as entidades A e B é unidirecional se apenas uma delas tem atributos da outra, mas não vice-versa.
- Numa relação bidirecional um dos lados é o 'dono' (*owning side*), o outro lado é o inverso (*inverse side*)
- Numa relação **ManyToOne**, o lado Many é sempre o dono
- O lado inverso usa a anotação **@mappedBy** no atributo correspondente ao dono da relação. Significa que essa informação não está na tabela inversa, mas que é preenchida com informação recolhida na tabela dono

```
@Entity
public class Employee {
  String name;
  @ManyToOne
  Department department;
}
```

```
@Entity
public class Department {
  @OneToMany(mappedBy="department")
  Set<Employee> employees;
}
```

Relações entre Entidades

Dono ?

- Numa relação 1-1 o ‘dono’ é aquele lado que contém a chave estrangeira

- Numa relação unidirecional há apenas um ‘dono’ que é a entidade que tem um atributo da outra entidade

Relações entre Entidades

A anotação **@JoinColumn** define que uma dada coluna representa a associação das entidades, indicando que o respectivo atributo representa a chave estrangeira

```
public class Cliente {

  @Id
  private int idCliente;
  ...
  @OneToOne
  @JoinColumn(nullable = false, name = "fkDesconto")
  private Desconto desconto;
```

➔ Neste exemplo, a coluna desconto (que será chamada **fkDesconto**) corresponde à chave estrangeira oriunda da tabela `Desconto`.

➔ Se fosse uma relação **@OneToMany** a chave estrangeira ficaria na outra classe:

```
public class Venda { ...
  @OneToMany
  @JoinColumn(nullable = false, name = "fkVenda")
  private List<VendaProduto> items;
```

Persistir Entidades

Pode-se guardar entidades explicitamente:

```
Cliente cliente = new Cliente(...);
 em.getTransaction().begin();
 em.persist(cliente);
 em.getTransaction().commit();
```

- O objecto cliente passa ao estado managed no *entity manager* `em`, e é guardado na BD após o **`commit()`**
  - Se o `persist` receber um objeto que não é uma entidade é lançada a excepção **`IllegalArgumentException`**
  - Se já existir uma entidade com a mesma chave é lançada uma **`EntityExistsException`**
  - É necessário uma transação activa senão é lançada uma **`IllegalStateException`**
- É preciso que todos os objetos **relacionados** sejam guardados no **`commit`**, senão a operação dá erro (**`IllegalStateException`**)
  - Para isso faz-se uma persistência explícita de todos esses objetos (trabalhoso e susceptível a erros) ou usa-se o mecanismos de cascata (cf. adiante)

Persistência por Cascata

Em vez de uma persistência explícita pode usar-se o mecanismo de cascata

```
@Entity
    public class Venda {
      @OneToMany(cascade = ALL)
        private List<ProdutoVenda> produtosVenda;
```

- Cada vez que uma venda é persistida, automaticamente todas as entidades **ProdutoVenda** relacionadas são igualmente persistidas, sem ser necessário explicitar esses comandos
- Os outros valores possíveis para o atributo cascade são: `DETACH`, `MERGE`, `PERSIST`, `REFRESH`, `REMOVE` (que executam a respetiva operação em cascata)
- O valor **ALL** inclui todos os anteriores, ie, é equivalente a

```
cascade={PERSIST, MERGE, REMOVE, REFRESH, DETACH}
```

Atributos Temporários

Pode haver atributos que não sejam para armazenar

- Estes incluem atributos estáticos, atributos finais ou outros atributos declarados pelo Java como transientes
- Em Java é possível declarar atributos transientes, i.e., atributos que não são para guardar numa serialização:

```
class C implements Serializable {
    private transient int atr;
    ...
```

- A anotação JPA correspondente é:

```
@Entity
public class C {
  @Transient
  double seed;
}
```

Recolher Entidades da BD

O JPA pode ir buscar entidades à BD

- Essas entidades até poderão estar já no contexto de persistência que, como referido, funciona como *cache*
- Podem ser realizadas *queries* usando a linguagem **JPQL** (cf. adiante)

Recolha por classe e chave primária (eg, cliente com id 123):

```
EntityManager em;
...
Cliente cliente = em.find(Cliente.class, 123);
```

Ou é devolvido o objeto da cache ou é criado um novo objeto.

Se a chave não existir, é devolvido null

Pode usar-se o método **e.getReference(...)** uma versão *lazy* do find(). Neste caso, devolve um objeto oco (*hollow object*) tendo apenas inicializado o atributo chave. Tudo o resto só é carregado se houver uma consulta ao estado do objeto. Útil quando apenas se quer saber da existência da instância/registo

Recolher Entidades da BD

Pode fazer-se um *refresh* de uma entidade para substituir o estado do objeto em memória com aquele da BD:

- Se a cascata estiver activa, o *refresh* é também realizado aos objetos referenciados

```
em.refresh(cliente);
```

Seja o objeto preenchido ou oco, se existir o acesso a um dos campos (via um *get*), o conteúdo é sempre carregado.

- Do ponto de vista do programador é como se toda a BD estivesse em memória
- Porém, isto apenas acontece enquanto as entidades não estão *detached*. Se o `EntityManager` for fechado, este comportamento deixa de ocorrer. Objetos ocos já não podem ser preenchidos.
- O JPA permite verificar se um dado elemento está carregado ou não:

```
PersistenceUtil util   = Persistence.getPersistenceUtil();
boolean isObjectLoaded = util.isLoaded(cliente);
boolean isFieldLoaded  = util.isLoaded(cliente, "desconto");
```

Updates

Basta ter a entidade em memória, criar uma transação activa e realizar um *commit*:

```
Cliente cliente = em.find(Cliente.class, 4345);

em.getTransaction().begin();
cliente.setDesconto(novoDesconto);
em.getTransaction().commit();
```

Remoções

As remoções funcionam de forma similar:

```
Cliente cliente = em.find(Cliente.class, 1);

em.getTransaction().begin();
em.remove(cliente);
em.getTransaction().commit();
```

Objetos embebidos são igualmente apagados. Os efeitos por cascata também funcionam:

```
@Entity
class Employee {

    @OneToOne(cascade=REMOVE) // ou cascade=ALL
    private Address address;
}
```

Remoção de Orfãos

Existe uma opção relacionada que tem a ver com a remoção de entidades que perderam todas as referências para elas.

```
@Entity
class Employee {
    @OneToOne(orphanRemoval=true)
    private Address address;
}
```

Deste modo garante-se que não ficam na BD registos que não são úteis pois não podem mais ser consultados.

Funciona para as relações **`@OneToOne`** e **`@OneToMany`**.

Pode ser usado para coleções. Não havendo mais referências, quando um elemento é eliminado da coleção ele é eliminado da BD.

Queries

- O **J**ava **P**ersistence **Q**uery **L**anguage (**JPQL**) é uma versão OO do SQL
- No JPA a interface **TypedQuery** é usada para realizar perguntas à BD

```
TypedQuery<Cliente> q =
   em.createQuery("SELECT c FROM Cliente c", Cliente.class );
```

Vai buscar todos os clientes que, assim, podem ser acedidos por **q**

Existem os seguintes métodos:

- **getSingleResult()** para ir buscar apenas um resultado:

```
TypedQuery<Long> q2 = em.createQuery(
      "SELECT COUNT(c) FROM Cliente c", Long.class);
long clientCount = q2.getSingleResult();
```

- **getResultList()** para ir buscar vários resultados.

```
List<Cliente> results = q.getResultList();
for (Cliente c : results) {
   System.out.println( c.getName() );
}
```

Queries

```
TypedQuery<Cliente> q =
   em.createQuery("SELECT c FROM Cliente c", Cliente.class );
```

Nome atribuido a entidade!

Se o nome da entidade for CLIENTE:

```
TypedQuery<Cliente> q =
   em.createQuery("SELECT c FROM CLIENTE c", Cliente.class );
```

Queries

- Operação **DELETE**:

```
int count = em.createQuery("DELETE FROM Cliente").executeUpdate();
```

  - Apagaria todos os clientes

- Operação **UPDATE**:

```
int count = em.createQuery(
      "UPDATE Cliente SET telefone = 0").executeUpdate();
```

  - Faria reset aos telefones dos clientes

  - É necessário que exista uma transação activa (senão é lançada uma excepção `TransactionRequiredException`)

Parâmetros nas Queries

Eles permitem a reutilização de *queries*: diferentes valores dos parâmetros correspondem a diferentes queries efetuadas sobre a BD.

- Um parâmetro é uma palavra prefixada pelo símbolo ‘ **:** ‘
- O método **setParameter()** define um valor para o parâmetro e devolve um objeto do mesmo tipo **TypeQuery<T>**
- O tipo do parâmetro é implicitamente definido quando é passado o valor (no caso abaixo, só na invocação do nome é que se percebe ser uma *string*.

```
public Cliente getCliente(EntityManager em, String name) {
  TypedQuery<Cliente> query = em.createQuery(
      "SELECT c FROM Cliente c WHERE c.name = :par", Cliente.class);
  return query.setParameter("par", name).getSingleResult();
}
```

Parâmetros nas Queries

Para além de parâmetros com nome, existem também os parâmetros ordinais onde se define um número de ordem:

```
public Cliente getCliente(EntityManager em, String name) {

  TypedQuery<Country> query = em.createQuery(
      "SELECT c FROM Cliente c WHERE c.name = ?1",
      Cliente.class);

  return query.setParameter(1, name).getSingleResult();
}
```

Named Queries

- Os exemplos anteriores mostram formas de criar dinamicamente novas *queries*
- O JPA permite também definir *queries* estáticas que podem melhorar a organização do código

  Isto porque as queries são definidas nas anotações e não no código.

```
@Entity
@NamedQuery(name="Cliente.findByNPC",
        query="SELECT c FROM Cliente c WHERE c.npc = :npc")

public class Cliente { … }
```

  Aqui tem de se definir um nome da *query* bem como descrever que operação SQL efectuar (conjuntamente com a descrição dos eventuais parâmetros)

Named Queries

Se uma classe tiver mais que uma NamedQuery é necessário fazer o seguinte:

```
@Entity
@NamedQueries({
  @NamedQuery(name="Cliente.findByNPC",
      query="SELECT c FROM Cliente c WHERE c.npc = :npc"),

  @NamedQuery(name="Cliente.findByName",
      query="SELECT c FROM Cliente c WHERE c.name = :name")
})
public class Cliente { … }
```

Named Queries

Para as usar usa-se outro método do `EntityManager`:

```
TypedQuery<Cliente> query =
    em.createNamedQuery("Cliente.findByNPC", Cliente.class);

query.setParameter("npc", 123456789);

Cliente cliente = query.getSingleResult();
```

Cliente.java

```
@Entity
@NamedQuery(name="Cliente.findByNPC",
        query="SELECT c FROM Cliente c WHERE c.npc = :npc")
```

JPQL – Select

Selecionar colunas especificas da tabela:

```
SELECT c.name, c.npc FROM Cliente AS c
```

- Cada resultado vem num *array* **`Object[]`**.
- A dimensão do *array* corresponde ao número de valores devolvidos por cada resultado do select:

```
TypedQuery<Object[]> query = em.createQuery(
      "SELECT c.name, c.npc FROM Country AS c",
      Object[].class);

List<Object[]> results = query.getResultList();
for (Object[] result : results) {
  System.out.println("None: "  + result[0] +
                     ", npc: " + result[1]);
}
```

JPQL – Select

Podemos ter múltiplas variáveis:

```
SELECT c1, c2 FROM Country c1, Country c2
              WHERE c2 MEMBER OF c1.neighbors
```

Neste e.g. a entidade **`Country`** teria um atributo coleção designado **`neighbors`** de tipo **`Collection<Country>`**, com todos os países que um dado país faz fronteira

Este select devolveria todos os pares de países com fronteira entre si

São efetuados dois ciclos (um com **`c1`**, e outro interno com **`c2`**) sobre todos os países da tabela **`Country`**

JPQL – Select

Neste mesmo exemplo, uma solução mais eficiente seria realizar um *inner join*:

```
SELECT c1, c2 FROM Country c1 JOIN c1.neighbors c2
```

Aqui, o segundo ciclo (da variável `c2`) seria declarada no contexto mais limitado dos vizinhos do atual valor `c1`.

Assumindo que cada país tem apenas uma capital, as seguintes instruções são equivalentes:

```
SELECT c.name, p.name FROM Country c JOIN c.capital p
SELECT c.name, c.capital.name FROM Country c
```

No 1º e.g., a variável p fica associada ao valor (único) da capital do país actual.

JPQL – Select

No e.g. seguinte são devolvidos pares (**país**, **língua**) onde a **população** seja maior que o parâmetro **:p**, e pelo menos uma das linguagens oficiais pertença ao parâmetro **:languages** (cujo tipo é uma coleção Java):

```
SELECT c, l FROM Country c JOIN c.languages l
    WHERE c.population > :p AND l IN :languages
```

No seguinte devolvemos pares de moeda com o total de população dos países europeus que a usam, incluíndo apenas moedas usadas em mais do que um país:

```
SELECT c.currency, SUM(c.population)
   FROM Country c
   WHERE 'Europe' MEMBER OF c.continents
   GROUP BY c.currency
   HAVING COUNT(c) > 1
```

Relembrar que a lista devolvida seria de elementos de tipo **Object[]**

JPQL – Select

Devolve os nomes dos países com mais de 10^6 pessoas ordenados pela sua população por ordem crescente (e se têm a mesma população, ordenados pelo nome por ordem decrescente):

```
SELECT c.name
   FROM Country c
   WHERE c.population > 1000000
   ORDER BY c.population ASC, c.name DESC
```

Como se pode observar, existe uma grande expressividade nos tipos de *selects* que podemos realizar.

JPQL – Delete

Comandos que alteram a BD:

```
int deletedCount = em.createQuery(
                "DELETE FROM Country c").executeUpdate();
```

Apaga todos os registos da tabela

```
query = em.createQuery(
    "DELETE FROM Country c WHERE c.population < :p");

int deletedCount =
    query.setParameter("p", 100000).executeUpdate();
```

Apaga os países com menos de 10^5 pessoas

JPQL – Update

Comandos que alteram a BD:

```
query = em.createQuery(
    "UPDATE Country c SET c.population = 0, c.area = 0");

int updateCount = query.executeUpdate();
```

Coloca a zero a população e área de todos os países

```
query = em.createQuery(
  "UPDATE Country SET population = population * 1.1 " +
  "WHERE c.population < :p");

int updateCount = query.setParameter("p",
100000).executeUpdate();
```

As populações com menos de 10^5 pessoas crescem 10%