

FIAP GRADUAÇÃO

# ENTERPRISE APPLICATION DEVELOPMENT

*Prof. THIAGO T. I. YAMAMOTO*

#06 – JPQL: JAVA PERSISTENCE QUERY LANGUAGE



- ♦ Introdução
- ♦ Linguagem JP-QL
- ♦ Interface Query
- ♦ Definindo parâmetros
- ♦ Exemplos de JP-QL

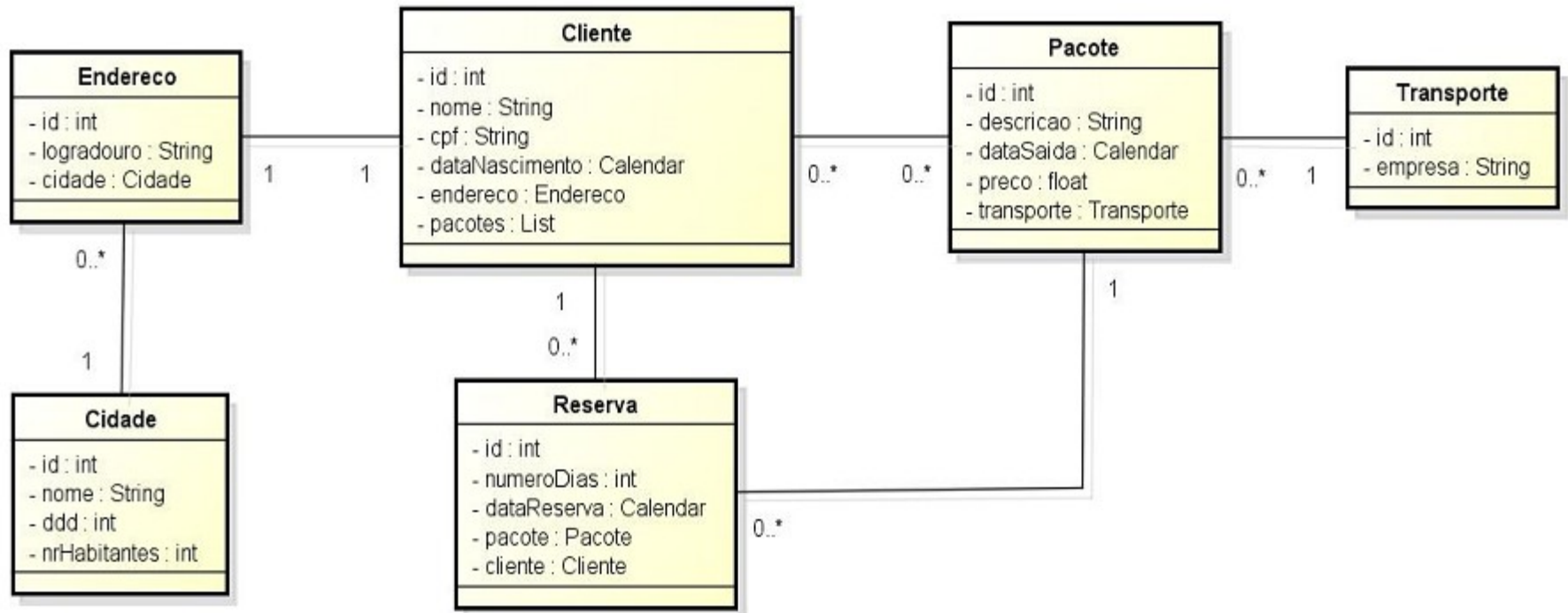
A especificação JPA estabelece uma linguagem de consultas às entidades denominada **Java Persistence Query Language** (JPQL);

Tem bastante semelhança com o SQL;

É permitido também a realização de consultas utilizando SQL nativo porém implica em prejuízo na portabilidade;

Consultas podem receber nomes e depois serem referenciadas por meio deles (*named queries*);

O modelo abaixo será utilizado para a realização de consultas:



Importe no eclipse o projeto 06-JPA-JPQL, altere o usuário e senha do banco no persistence.xml e execute a classe br.com.fiap.banco.PopulaBanco.

# LINGUAGEM JPQL

É uma linguagem muito semelhante ao SQL, porém as consultas são realizadas sobre as **entidades** mapeadas e não sobre as tabelas no banco de dados;

Não é necessário tornar explícita as associações entre as entidades como se faz em SQL com os *joins* entre tabelas;

Costuma-se atribuir um apelido (alias) aos nomes das entidades e pode-se omitir a palavra *select* das instruções JPQL.

## Exemplos

**select Object(c) from Cliente as c → from Cliente c**

**select Object(h) from Cliente c where c.id = 1 → from Cliente c where c.id = 1**

**select new Cliente (id, nome) from Cliente c → com o construtor de Cliente**

**select c.dadoPagamento.cpf from Cliente c → não é necessário join com DadoPagamento!**

Os métodos para realização de consultas encontram-se na interface **Query**;

Para obter-se uma instância de **Query** devemos acionar um dos métodos abaixo a partir de uma instância **EntityManager**:

- **createQuery(String P1)**: cria uma consulta com base no JPQL fornecido no P1;
- **createNamedQuery(String P1)**: referencia uma consulta por meio de seu nome definido em P1;
- **createNativeQuery(String P1)**: cria uma consulta com base no SQL nativo fornecido no P1.

## Exemplo

**EntityManager em = ...**

**Query q = em.createQuery("from Cliente");**



A especificação JPA 2.0 introduziu uma nova interface para consultas, a **TypedQuery**;

**TypedQuery** permite trabalhar com o recurso Java *generics*;

É uma sub-interface de **Query** e, portanto, possui os mesmos métodos declarados;

O método **createQuery** deve receber como segundo parâmetro a classe da entidade que será retornada;

## Exemplo

```
EntityManager em = ...  
TypedQuery<ClienteEntity> q = em.createQuery("from Cliente",  
ClienteEntity.class);
```

# I MÉTODOS DE QUERY E TYPEDQUERY FIAP

A interface Query oferece uma série de métodos para execução das consultas, definição de parâmetros, controle de paginação, etc.

Alguns métodos bastante utilizados:

**getSingleResult():** executa a consulta e retorna um único resultado ou **EntityNotFoundException** caso nenhuma entidade seja localizada ou **NonUniqueResultException** caso exista mais de uma entidade como resultado;

**getResultList():** executa a consulta e pode retornar mais de um resultado representado por uma **List**.

## Exemplo

```
EntityManager em = ...
```

```
Query q = em.createQuery("from Cliente");
```

```
List<ClienteEntity> clientes = q.getResultList();
```

Os parâmetros passados podem ser tanto tipos de dados primitivos quanto entidades. No exemplo abaixo o cliente com o id = 2 será considerado como parâmetro pela consulta:

```
EntityManager em = ... // Obter o EntityManager
Cliente c = em.find(Cliente.class, 2);
Query q = em.createQuery("from Endereco e where e.cliente = :idCli");
q.setParameter("idCli", c);
```

Pode-se também passar como parâmetro um conjunto de entidades. No caso abaixo, a consulta retornará todos os endereços das cidades id = 1 e 2:

```
EntityManager em = ... // Obter o EntityManager
Cidade c1 = em.find(Cidade.class, 1);
Cidade c2 = em.find(Cidade.class, 2);
List<Cidade> cs = new ArrayList<Cidade>();
cs.add(c1);
cs.add(c2);
Query q = em.createQuery("from Endereco e where e.cidade in (:cidades)");
q.setParameter("cidades", cs);
```

Executar as seguintes consultas:

1. Obter todos os clientes;
2. Obter todos os pacotes que possuem um transporte específico;
3. Obter todos clientes localizados no estado de SP;
4. Obter todos os clientes que efetuaram reservas em uma quantidade de dias específica.

Uma consulta que retorne muitos um conjunto de entidades muito extenso pode consumir muita memória e tempo de processamento;

O ideal é retornar um subconjunto de entidades inicialmente e somente obter o próximo subconjunto quando necessário;

Pode-se utilizar o recurso de paginação do resultado por meio dos métodos:

**setMaxResults(int P1):** define uma quantidade máxima de entidades, definida em **P1**, a serem retornadas;

**setFirstResult(int P1):** define a posição do primeiro registro, definido em **P1**, a partir do qual os resultados serão obtidos;

## Exemplo

```
Query q = em.createQuery("from Cliente");  
q.setMaxResults(5); //cinco entidades  
q.setFirstResult(0); // a partir do primeiro registro
```

Ao invés de retornar uma entidade completa podemos selecionar apenas alguns atributos para o retorno de uma consulta;

Nestes casos o resultado da consulta é um *array* de objetos onde cada atributo possui um índice correspondente;

## Exemplo

*// Obter o EntityManager*

EntityManager em = ...

Query q = em.createQuery("select c.nome, c.uf from Cidade c");

Collection<Object[]> resultado = q.getResultList();

for (Object[] o:resultado)

    System.out.println(o[0] + ", " + o[1]);

# EXEMPLOS JPQL

## Textos Literais

from Cidade c where c.nome = "SAO PAULO"

## Valores Distintos

select distinct h.cidade.nome from Endereco h

## Intervalo de Valores

from Cidade c where c.habitantes between 10000 and 20000

from Cidade c where c.habitantes not between 10000 and 20000

## Conjunto de Valores

from Cidade c where c.uf in ('SP', 'BA')

## Valores Nulos

from Pacote p where p.dataSaida is null

from Pacote p where p.dataSaida is not null

from Cliente c where c.enderecos is empty → retorna os clientes sem endereços (aplicado quando existe associação baseada em *Collection*)



## Ordenação

from Cliente c order by c.nome

## Like

from Cliente c where c.nome like 'MANOEL%'

## Sub-Consultas

from Reserva r where r.numDias = (select max(r2.numDias) from Reserva r2)

## Sub-Consultas com operador NOT IN

from Cliente c where c not in (select r.cliente from Reserva r)

Escrever o conjunto de métodos abaixo nas respectivas classes DAO:

**buscarPorDatas(Calendar inicio, Calendar fim):** retorna todos os pacotes cuja data de saída esteja no intervalo especificado como parâmetro;

**buscar(String nome, String cidade):** retorna os clientes que possuam no nome o texto informado como parâmetro e que tenham algum endereço no nome de cidade informado;

**buscarPorEstados(Collection<String> estados):** retorna todos os clientes conforme os estados passados como parâmetro na *Collection*;

# FUNÇÕES GERAIS

- **UPPER / LOWER:** transforma uma string em maiúscula / minúscula  
`select lower(c.nome) from Cliente c`  
`select upper(c.nome) from Cliente c`
- **TRIM:** remove caracteres em branco do início e fim de uma string  
`select trim(c.nome) from Cliente c`
- **CONCAT:** concatena duas strings  
`select concat(e.logradouro, e.cidade.nome) from Endereco`
- **LENGTH:** retorna o tamanho de uma string  
`from Cliente c where length(c.nome) < 20`
- **LOCATE:** retorna a posição de início de uma *string* dentro de outra ou 0 se não localizada  
`select locate('JOAO', c.nome) from Cliente c`
- **SUBSTRING:** retorna uma *sub-string* a partir de uma *string* original  
`select substring(c.nome, 3, 5) from Cliente c`

- **CURRENT\_DATE, CURRENT\_TIME, CURRENT\_TIMESTAMP:**  
retornam a data, hora e data e hora atual respectivamente  
**from Pacote p where p.dataSaida = current\_date**

- **COUNT**: conta o número de ocorrências de determinado valor  
`select count(c) from Cliente c`
- **MAX**: obtém o valor máximo armazenado em um atributo  
`select max(r.numDias) from Reserva r`
- **MIN**: obtém o valor mínimo armazenado em um atributo  
`select min(r.numDias) from Reserva r`
- **AVG**: obtém a média de valores armazenado em um atributo  
`select avg(r.numDias) from Reserva r`
- **SUM**: obtém a soma de valores armazenado em um atributo  
`select sum(r.numDias) from Reserva r`
- **GROUP BY**: agrupamento de valores  
`select count(e), e.cidade.nome from Endereco e group by e.cidade.nome`

# NAMED QUERIES

Um sistema normalmente necessita efetuar várias consultas às suas entidades;

A declaração de consultas encontra-se espalhada pelo código dificultando sua manutenção;

A API JPA oferece as **Named Queries**, que:

- São estruturas pré-compiladas possibilitando a identificação de erros antes da execução do código;
- Ajudam a manter o código mais limpo uma vez que a declaração e a execução das consultas ficam separadas;
- São *thread-safe*, portanto, podem ser compartilhadas por muitas classes;
- Podem ser parametrizadas normalmente;
- Devem ser declaradas somente nas classes de entidade.



Para criar uma Named Query basta declará-la em uma entidade utilizando a anotação **@NamedQuery** que possui os argumentos abaixo:

- **name**: nome da consulta que será utilizado para referenciá-la (incluir o nome do pacote da entidade);
- **query**: a declaração da consulta em si.


Para executar a Named Query, utilizar o método **createNamedQuery(String nome)** da interface Query passando como parâmetro o nome da consulta a ser executada:

```
@NamedQuery(name= "Cliente.porNome",  
query="from Cliente c where c.nome like :nome")  
@Entity(name='Cliente')  
public class ClienteEntity { ... }
```

```
Query q = em.createNamedQuery("Cliente.porNome");  
q.setParameter("nome", "%JOAO%");  
List<ClienteEntity> cs = q.getResultList();
```

Para definir mais de uma named query em uma entidade deve-se utilizar a anotação **NamedQueries** conforme abaixo:

```
@NamedQueries({  
    @NamedQuery(name="consulta1", query="...")  
    ,@NamedQuery(name="consulta2", query="...")  
})  
@Entity(name="Cliente")  
@Table(name="TAB_CLIENTE")  
public class ClienteEntity {  
    ...  
}
```



As consultas devem ser criadas dentro de **NamedQueries** separadas por **vírgula**!

# NATIVE QUERIES

Pode-se criar consultas nativas ao banco de dados, isto é, utilizando SQL de um banco específico ao invés de JPQL;

Atenção para a perda de portabilidade;

Utilizar o método **createNativeQuery (String sql, Class entidade)**, onde os campos retornados pelo SQL devem corresponder aos atributos da entidade passada no segundo argumento (opcional);

Caso a entidade não seja fornecida no segundo argumento, retorna uma **Collection<Object[]>** onde cada posição do array corresponde a uma coluna na projeção da consulta (Ex: 0 = COD\_CIDADE; 1 = NOM\_CIDADE, etc...)

## Exemplo

```
Query q = em.createNativeQuery("SELECT COD_CIDADE, NOM_CIDADE, DES_UF FROM TAB_CIDADE", CidadeEntity.class);
```

Copyright © 2013 - 2017 Prof. Me. Thiago T. I. Yamamoto

Todos direitos reservados. Reprodução ou divulgação total ou parcial deste documento é expressamente proibido sem o consentimento formal, por escrito, do Professor (autor).

*“Nenhum obstáculo é tão grande se sua vontade  
de vencer for maior”*