JPQL - Java Persistence Query Language

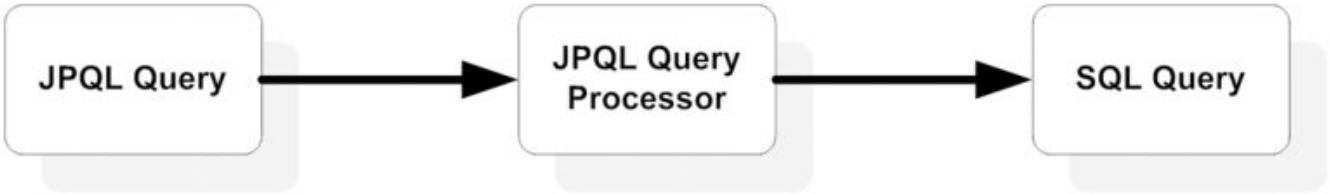
DSC AULA 5

JPQL x SQL

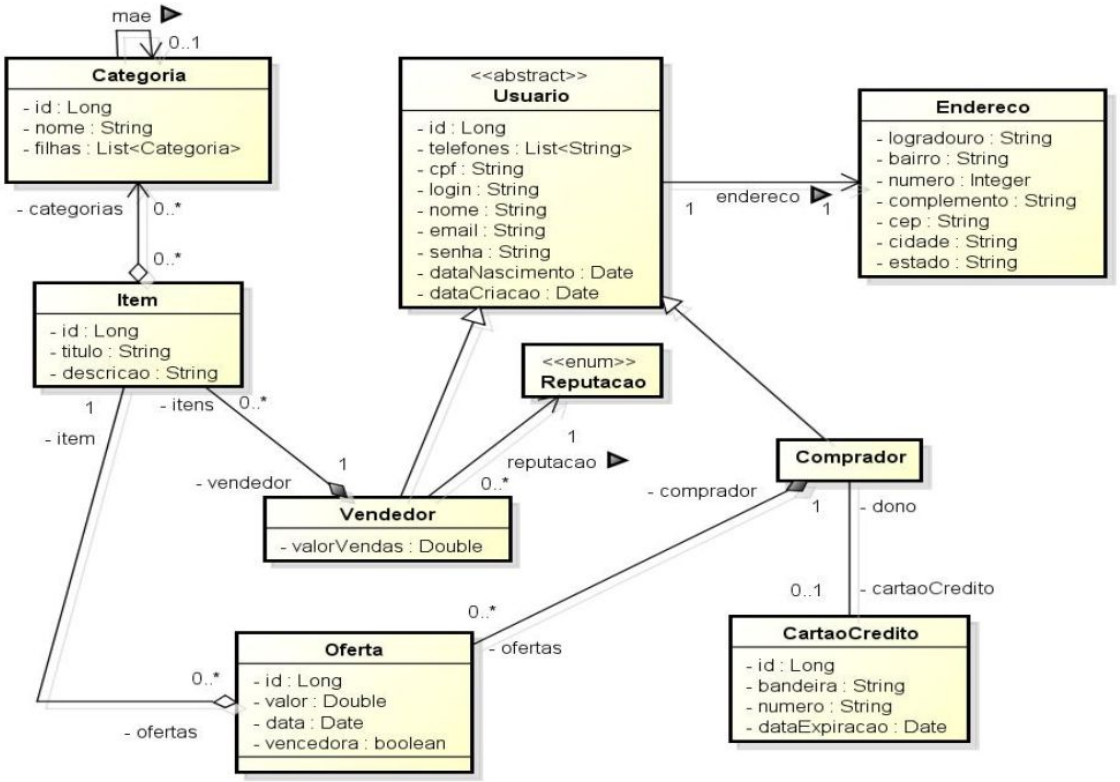
- JPQL opera sobre classes e objetos (entidades)
- SQL opera sobre tabelas, colunas e linhas.
- Embora JPQL e SQL sejam semelhantes, elas operam em dois mundos muito diferentes.

JPQL

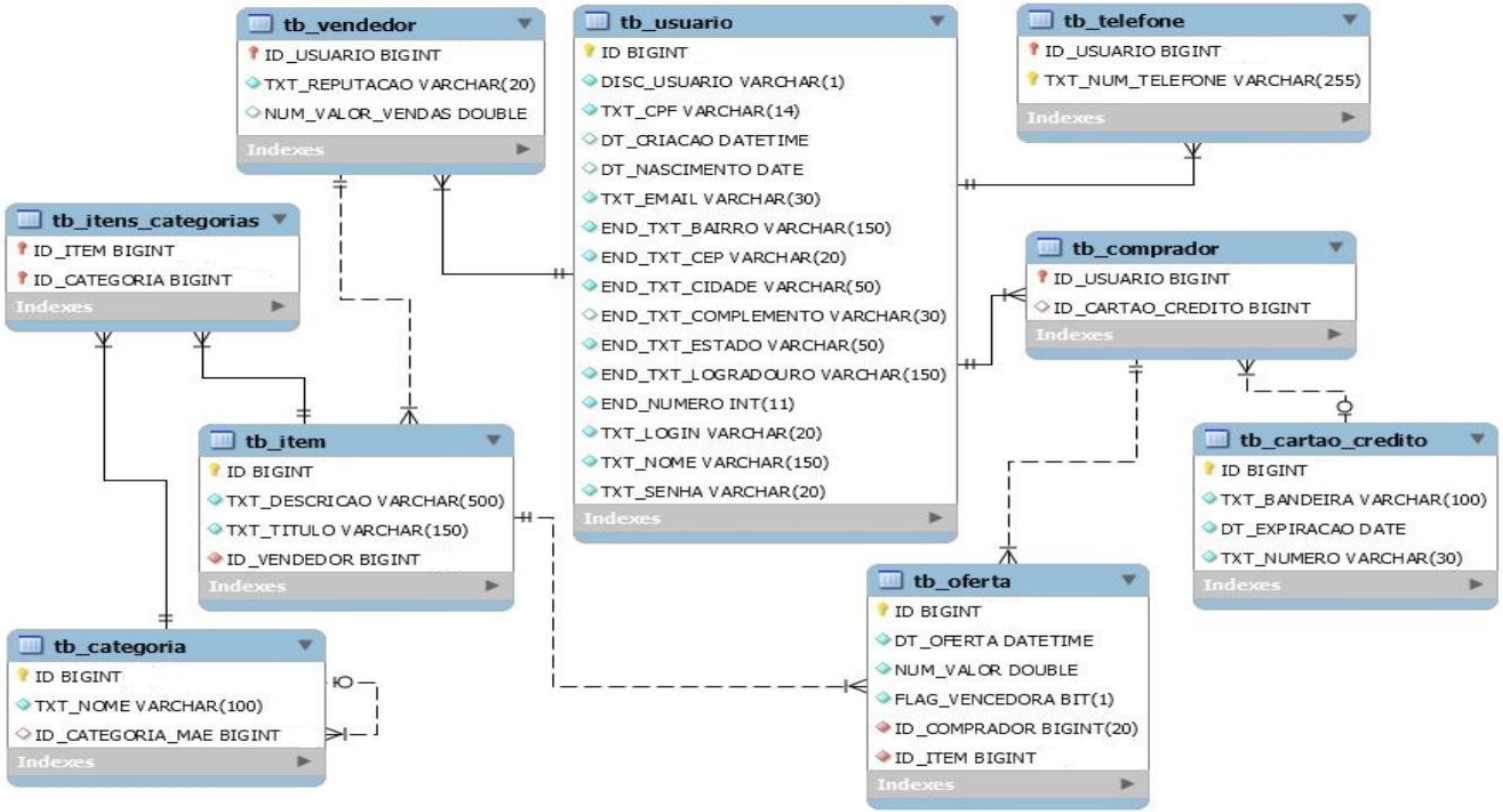
- JPQL suporta três tipos de statements: select, update e delete.
  - Nesse curso, vamos abordar apenas o select.
- Cada query JPQL é traduzida em uma query SQL para então ser executada.



As classes que iremos trabalhar como exemplo



Modelo de dados utilizado



Observe o mapeamento e a query abaixo

```java
public class Categoria implements Serializable {

    @Id
    @Column(name = "ID_CATEGORIA")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "TXT_NOME", length = 100, nullable = false, unique = true)
    private String nome;
    @ManyToOne(fetch = FetchType.LAZY, optional = true)
    @JoinColumn(name = "ID_CATEGORIA_MAE", referencedColumnName = "ID_CATEGORIA")
    private Categoria mae;
    @OneToMany(mappedBy = "mae", orphanRemoval = true)
    private List<Categoria> filhas;
```

```java
TypedQuery<Categoria> query = em.createQuery(
        "SELECT c FROM Categoria c WHERE c.nome LIKE :nome",
        Categoria.class);
query.setParameter("nome", "Instrumentos%");
List<Categoria> categorias = query.getResultList();
```

- Considere a entidade Categoria e a consulta JPQL ao lado, que seleciona categorias pelo nome.
- Observe que objetos estão sendo selecionados, e não campos de tabelas.
- Observe, ainda, que a comparação com o nome é através do valor do atributo da instância de categoria.

SQL gerado

- SELECT ID_CATEGORIA, TXT_NOME, ID_CATEGORIA_MAE FROM TB_CATEGORIA WHERE TXT_NOME LIKE ?
  - Observe que o SQL gerado, como esperado, devido ao carregamento preguiçoso, não recupera as informações da categoria mãe, mas apenas o identificador.

Named Queries

- Alternativamente pode-se escrever as *queries* na própria entidade, dando a cada uma delas um nome

```
@Entity
@Table(name = "TB_CATEGORIA")
@NamedQueries(
        {
            @NamedQuery(
                    name = "Categoria.PorNome",
                    query = "SELECT c FROM Categoria c WHERE c.nome LIKE :nome ORDER BY c.id"
            )
        }
)
public class Categoria implements Serializable {
```

Named Queries

- No caso de name queries, para utilizá-las, deve ser invocado o método *createNamedQuery*

```
TypedQuery<Categoria> query = em.createNamedQuery("Categoria.PorNome",
        Categoria.class);
query.setParameter("nome", "Instrumentos%");
List<Categoria> categorias = query.getResultList();
```

Estrutura JPQL

- Vamos analisar com calma a estrutura de nossa query nomeada.
  - Podemos ver palavras reservadas padrão de SQL também em JPQL, que possuem basicamente o mesmo significado que possuem em SQL
  - São elas: SELECT, FROM, WHERE, LIKE, ORDER BY
  - A diferença principal é que agora estamos lidando com objetos
  - Podemos encontrar em queries JPQL também palavras reservadas como GROUP BY e HAVING, que têm o mesmo significado que possuem em SQL.

```
"SELECT c FROM Categoria c WHERE c.nome LIKE :nome ORDER BY c.id"
```

JPQL – Path Expression

- Um expressão de caminho (path expression) é uma variável identificadora seguida pelo operador de navegação (.) e um atributo ou associação (um objeto ou uma coleção)
- Normalmente é utilizada numa cláusula WHERE ou ORDER BY, mas também pode ser utilizada no SELECT
- No exemplo abaixo, c.filhas é uma coleção. Como expressões são conhecidas como expressões de conjuntos de valores.
    - Observação: não é permitido navegar através dos valores de expressões de conjunto de valores (ex. você não pode fazer c.filhas.nome)

```
"SELECT c FROM Categoria c WHERE c.filhas IS NOT EMPTY"
```

JPQL – Path Expression

- Se a associação for muitos para um ou um para um, os campos da associação serão de um tipo de objeto específico (expressões de caminho de valor único).
  - Você pode navegar em expressões de caminho de valor único
  - No exemplo abaixo, selecionamos compradores que possuem cartão de crédito com a bandeira passada como parâmetro.

```
"SELECT c FROM Comprador c WHERE c.cartaoCredito.bandeira like ?1"
```

JPQL – Cláusula SELECT

- Uma cláusula SELECT pode ter mais de uma variável identificadora, uma ou mais expressões de caminho de valor único, ou funções de agregação separadas por vírgulas (como COUNT, por exemplo)

```
SELECT [DISTINCT] expression1, expression2, .... expressionN
```

- No exemplo abaixo, são selecionadas as diferentes bandeiras de cartões de crédito (VISA, MAESTRO, etc.)
  - Observe que a query, nesse caso, retorna uma lista de String.

```
TypedQuery<String> query
        = em.createQuery("SELECT DISTINCT(c.bandeira) FROM CartaoCredito c ORDER BY c.bandeira", String.class);
List<String> bandeiras = query.getResultList();
```

JPQL – Cláusula SELECT

- Recuperando um único valor com o uso da função de agregação COUNT, que possui o mesmo significado de função SQL análoga.
  - Observe que nesse caso estamos retornando a quantidade de categorias que possuem o atributo *categoria.mae* não nulo.
  - O resultado de um COUNT em JPQL é sempre um Long

```
TypedQuery<Long> query = em.createQuery(
        "SELECT COUNT(c) FROM Categoria c WHERE c.mae IS NOT NULL", Long.class);
Long resultado = query.getSingleResult();
```

JPQL – Cláusula SELECT

- Podemos ainda selecionar mais de um valor numa mesma query, como a maior e menor data de nascimento entre compradores.
  - Observe que o retorno é um array de objetos.

```
Query query = em.createQuery(
        "SELECT MAX(c.dataNascimento), MIN(c.dataNascimento) FROM Comprador c");
Object[] resultado = (Object[]) query.getSingleResult();
```

JPQL – Palavras Reservadas

Types	Reserved words
Statements and clauses	SELECT, UPDATE, DELETE, FROM, WHERE, GROUP, HAVING, ORDER, BY, ASC, DESC
Joins	JOIN, OUTER, INNER, LEFT, FETCH
Conditions and operators	DISTINCT, OBJECT, NULL, TRUE, FALSE, NOT, AND, OR, BETWEEN, LIKE, IN, AS, UNKNOWN, EMPTY, MEMBER, OF, IS, NEW, EXISTS, ALL, ANY, SOME
Functions	AVG, MAX, MIN, SUM, COUNT, MOD, UPPER, LOWER, TRIM, POSITION, CHARACTER_LENGTH, CHAR_LENGTH, BIT_LENGTH, CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP

Filtrando com WHERE

- Quase todos os tipo de literais Java como boolean, float, enum, String, int, etc. têm suporte na cláusula WHERE JPQL.
  - Porém, não é possível utilizar tipos numéricos como octal ou hexadecimais, nem tipos array como byte[] ou char[]. Lembre-se que JPQL é traduzida para SQL, que impõe restrição de que tipos BLOB e CLOB não podem ser utilizados em WHERE

Filtrando com WHERE - operadores suportados

Operator type	Operator
Navigational	`.`
Unary sign	`+, -`
Arithmetic	`*, /, +, -`
Relational	`=, >, >=, <, <=, <>, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]`
Logical	`NOT, AND, OR`

Filtrando com WHERE

- A query abaixo recupera os compradores que possuem cartões de crédito “VISA” ou “MASTER”
  - Observe como devemos trabalhar com parâmetros numerados (“?” seguido por número do parâmetro, iniciando em 1)

```
TypedQuery<Comprador> query;
query = em.createQuery(
        "SELECT c FROM Comprador c "
        + "WHERE c.cartaoCredito.bandeira LIKE ?1 "
        + "OR c.cartaoCredito.bandeira LIKE ?2",
        Comprador.class);
query.setParameter(1, "VISA"); //Setando parâmetro posicional.
query.setParameter(2, "MASTERCARD"); //Setando parâmetro posicional.
List<Comprador> compradores = query.getResultList();
```

Filtrando com WHERE

- A query abaixo recupera os compradores que possuem cartões de crédito “VISA” ou “MASTER”
  - Observe o uso do IN, idêntico ao que se usa em SQL.

```
TypedQuery<Comprador> query;
query = em.createQuery(
        "SELECT c FROM Comprador c "
        + "WHERE c.cartaoCredito.bandeira IN ('MAESTRO', 'MASTERCARD')",
        Comprador.class);
List<Comprador> compradores = query.getResultList();
```

Filtrando com WHERE

- A query recupera os usuários que nasceram entre duas datas.
  - O uso do BETWEEN em JPQL também é idêntico ao que conhecemos no SQL.

```
TypedQuery<Usuario> query;
query = em.createQuery(
        "SELECT u FROM Usuario u WHERE u.dataNascimento BETWEEN ?1 AND ?2",
        Usuario.class);
query.setParameter(1, getData(21, Calendar.FEBRUARY, 1980));
query.setParameter(2, getData(1, Calendar.DECEMBER, 1990));
List<Usuario> usuarios = query.getResultList();
```

Lidando com valores nulos

- Quando uma expressão condicional encontra um valor nulo, a expressão pode ter como resultado desconhecido (UNKNOWN).
- Uma cláusula WHERE complexa, que combina mais de uma expressão condicional pode produzir um resultado desconhecido.
- É recomendável utilizar os operadores IS NULL ou IS NOT NULL para verificar se uma expressão de caminho de valor único contém valores nulos ou não nulos.

```
"SELECT c FROM Categoria c WHERE c.mae IS NOT NULL"
```

Lidando com valores nulos

Expression 1 value	Boolean operator	Expression 2 value	Result
TRUE	AND	null	UNKNOWN
FALSE	AND	null	FALSE
NULL	AND	null	UNKNOWN
TRUE	OR	null	TRUE
NULL	OR	null	UNKNOWN
FALSE	OR	null	UNKNOWN
	NOT	null	UNKNOWN

Lidando com coleções vazias

- Não é possível utilizar IS NULL para comparar uma expressão de caminho que seja de coleção.
  - Em vez disso, utilize IS EMPTY para saber se uma determinada coleção é vazia.

```
"SELECT i FROM Item i WHERE i.ofertas IS EMPTY"
```

Verificando a existência de uma entidade em uma coleção

- Podemos utilizar o operador MEMBER OF para testar a existência de uma entidade em uma coleção.
  - a query abaixo retorna a categoria mãe da categoria cujo id é 2, visto que ela, passada como parâmetro para a query precisa estar na coleção de filhas da categoria retornada.

```
Categoria categoria = em.find(Categoria.class, new Long(2));
TypedQuery<Categoria> query;
query = em.createQuery(
        "SELECT c FROM Categoria c WHERE :categoria MEMBER OF c.filhas",
        Categoria.class);
query.setParameter("categoria", categoria);
categoria = query.getSingleResult();
```

Funções de manipulação de String

String functions	Description
`CONCAT(string1, string2)`	Returns the value of concatenating two strings or literals together.
`SUBSTRING(string, position, length)`	Returns the substring starting at `position` that's `length` long.
`TRIM([LEADING | TRAILING | BOTH] [trim_character] FROM] string_to_trimmed)`	Trims the specified character to a new length. The trimming can be `LEADING`, `TRAILING`, or from `BOTH` ends. If no `trim_character` is specified, then a blank space is assumed.
`LOWER(string)`	Returns the string after converting to lowercase.
`UPPER(string)`	Returns the string after converting to uppercase.
`LENGTH(string)`	Returns the length of a string.
`LOCATE(searchString, stringToBeSearched[initialPosition])`	Returns the position of a given string within another string. The search starts at position 1 if `initialPosition` isn't specified.

Funções de manipulação de String

- Suponha que desejamos comparar o resultado da concatenação de duas strings com um valor literal.

```
WHERE CONCAT(u.firstName, u.lastName) = 'ViperAdmin'
```

- Suponha que desejamos utilizar a função substring para determinar se as três primeiras letras de u.lastName iniciam com ‘VIP’.

```
WHERE SUBSTRING(u.lastName, 1, 3) = 'VIP'
```

Funções aritméticas

Arithmetic functions	Description
`ABS(simple_arithmetic_expression)`	Returns the absolute value of `simple_arithmetic_expression`
`SQRT(simple_arithmetic_expression)`	Returns the square root value of `simple_arithmetic_expression` as a `double`
`MOD(num, div)`	Returns the result of executing the modulus operation for `num, div`
`SIZE(collection_value_path_expression)`	Returns the number of items in a collection

Funções aritméticas

- As funções aritméticas são autoexplicativas, como este exemplo de SIZE, que retorna as categorias que possuírem pelo menos três categorias filhas.

```
"SELECT c FROM Categoria c WHERE SIZE(c.filhas) >= 3"
```

Funções de agregação

Aggregate functions	Description	Return type
AVG	Returns the average value of all values of the field it's applied to	Double
COUNT	Returns the number of results returned by the query	Long
MAX	Returns the maximum value of the field it's applied to	Depends on the type of the persistence field
MIN	Returns the minimum value of the field it's applied to	Depends on the type of the persistence field
SUM	Returns the sum of all values on the field it's applied to	May return either Long or Double

GROUP BY e HAVING

- Você pode precisar agrupar dados por algum campo.
- Na 1ª consulta abaixo, são selecionadas as categorias, juntamente com suas respectivas quantidades de itens relacionados.

```
"SELECT c, COUNT(i) FROM Categoria c, Item i WHERE c MEMBER OF i.categorias GROUP BY c"
```

- Na 2ª consulta, por sua vez, são selecionadas as mesmas linhas com uma restrição a mais, devido ao HAVING, que filtra as categorias para que sejam retornadas apenas aquelas que possui 4 ou mais itens associados.

```
"SELECT c, COUNT(i) FROM Categoria c, Item i WHERE c MEMBER OF i.categorias GROUP BY c HAVING COUNT(i) >= 4"
```

Ordenando o resultado da consulta

- Podemos controlar a ordem dos valores e dos objetos recuperados em uma consulta utilizando ORDER BY
  - No exemplo abaixo, selecionamos todos os cartões, ordenados pela bandeira (em ordem decrescente) e pelo nome do dono (em ordem ascendente)

```
"SELECT c FROM CartaoCredito c ORDER BY c.bandeira DESC"
```

INNER JOIN

- Uma situação comum é a necessidade de unir duas ou mais entidades com base em seus relacionamentos.
  - INNER JOIN ou JOIN são equivalentes
  - A consulta ao lado retorna todos os compradores que possuem cartão de crédito (Comprador c JOIN c.cartaoCredito cc) ordenados por data de criação (descendente).

```
"SELECT c FROM Comprador c JOIN c.cartaoCredito cc ORDER BY c.dataCriacao DESC"
```

LEFT OUTER JOIN

- Permite que você recupere entidades adicionais que não combinam com as condições JOIN quando associações entre as entidades são opcionais.
  - LEFT OUTER JOIN ou LEFT JOIN são equivalentes.
  - A consulta ao lado retorna os CPF's e as bandeiras dos cartões dos compradores, com ordenação por CPF. Caso o comprador não possua cartão de crédito, a consulta retornará o valor null e o CPF do comprador.

```
"SELECT c.cpf, cc.bandeira FROM Comprador c LEFT JOIN c.cartaoCredito cc ORDER BY c.cpf"
```

JOIN FETCH

- Útil no caso de termos carregamento preguiçoso, mas desejamos carregar as entidades associadas definidas como FetchType.LAZY.
- Exemplo: considere o relacionamento abaixo, no qual o cartão de crédito possui mapeamento com *FetchType.LAZY* em Comprador.

```
@Entity
@Table(name = "TB_COMPRADOR")
@DiscriminatorValue(value = "C")
@PrimaryKeyJoinColumn(name = "ID_USUARIO", referencedColumnName = "ID_USUARIO")
public class Comprador extends Usuario implements Serializable {

    @OneToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL, optional = true)
    @JoinColumn(name = "ID_CARTAO_CREDITO", referencedColumnName = "ID_CARTAO_CREDITO")
    private CartaoCredito cartaoCredito;
```

JOIN FETCH

- Se quisermos retornar as informações de comprador e de cartão de crédito na mesma query, podemos fazer o seguinte

```
"SELECT c FROM Comprador c JOIN FETCH c.cartaoCredito"
```

Para Baixar

- O código completo do exemplo 7, acesse
  - https://svn.riouxsvn.com/dsc-ifpe/exemplo_07/
  - Um boa prática é ficar atento ao SQL gerado na execução do projeto