Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte
Campus Natal-Central
Diretoria Acadêmica de Gestão e Tecnologia da Informação

Tecnologia em Análise e Desenvolvimento de Software

JPA Query Language

Prof. Fellipe Aleixo
fellipe.aleixo@ifrn.edu.br

Consultas e JPQL

- Consultas são um recurso fundamental de todos os bancos de dados relacionais
- Consultas em JPA são criadas utilizando-se duas linguagens:
  - SQL (*Structured Query Language*)
    - Linguagem de consulta para banco de dados relacionais
  - JPAQL (JPA *Query Language*)
    - Linguagem declarativa semelhante à SQL – personalizada para o trabalho com objetos Java
    - Para efetuar consultas são referenciadas propriedades e relacionamentos e não tabelas do banco de dados

API de Consulta

- Um objeto consulta em JPA segue a interface javax.persistence.Query

```
public interface Query {
        public List getResultList();
        public Object getSingleResult();
        public int executeUpdate();
        public Query setMaxResults(int maxResult);
        public Query setFirstResult(int startPosition);
        public Query setHint(String hintName, Object value);
        public Query setFlushMode(FlushModeType flushMode);
        ...
```

API de Consulta

- Consultas são criadas a partir do através dos seguintes métodos do EntityManager:

```
    ...
    public Query setParameter(String name, Object value);
    public Query setParameter(String name, Date value,
                              TemporalType temporalType);
    public Query setParameter(String name, Calendar value,
                              TemporalType temporalType);
    public Query setParameter(int position, Object value);
    public Query setParameter(int position, Date value,
                              TemporalType temporalType);
    public Query setParameter(int position, Calendar value,
                              TemporalType temporalType);
}
```

API de Consulta

```
public interface EntityManager {
        ...
        public Query createQuery (String jpaqlString);
        public Query createNamedQuery (String name);
        public Query createNativeQuery (String sqlString);
        public Query createNativeQuery (String sqlString, Class resultClass);
        public Query createNativeQuery (String sqlString, String resultSetMapping);
        ...
}
```

Exemplo de Criação de Consulta

```
try {
        Query query = entityManager.createQuery("from Usuario u"+
                                " where u.nome='João' and c.telefone='9998-1234'");
        Usuario usuario = (Usuario) query.getSingleResult();
} catch (EntityNotFoundException notFound) {
        ...
} catch (NonUniqueResultException nonUnique) {
        ...
}
```

- A consulta é executada quando o método getSingleResult() é executado
- Esse método espera que seja retornado um único objeto como resultado
- As exceções estendem RunTimeException

Parâmetros

- Parecida com uma PreparedStatement do JDBC, a JPAQL permite especificar parâmetros no momento da declaração de uma consulta
  - Possibilita que uma consulta seja reutilizada
  - Duas sintaxes são possíveis:
    - Parâmetros identificados
    - Parâmetros posicionais
  - A seguir, são mostradas duas variações do exemplo anterior fazendo uso dos dois tipos de sintaxe de parâmetros

Parâmetros

```
public List findByNomeETelefone(String nome, String telfone) {
        Query query = entityManager.createQuery("from Usuario u"+
                                                " where u.nome=:nome and c.telefone=:tel");
        query.setParameter("nome",nome);
        query.setParameter("tel",telefone);
        return query.getResultList();
}
```

```
public List findByNomeETelefone(String nome, String telfone) {
        Query query = entityManager.createQuery("from Usuario u"+
                                                " where u.nome=?1 and c.telefone=?2");
        query.setParameter(1,nome);
        query.setParameter(2,telefone);
        return query.getResultList();
}
```

- É recomendado o uso da identificação nominal

Paginando Resultados

- Para diminuir o número de resultados de uma consulta pode ser utilizada a paginação

```
public List getLivros(int max, int indice) {
        Query query = entityManager.createQuery("from Livro l");
        return query.setMaxResults(max).setFirstResult(indice).getResultList();
}
```

  - O método clear() do EntityManager pode ser utilizado antes de recuperar os próximos elementos para desacoplar os elementos inicialmente recuperados

Hints e FlushMode

- É possível a especificação de requisitos suplementares nas consultas, como o tempo máximo para uma consulta

```
Query query = manager.createQuery(“from Livro l”);
query.setHint(“org.hibernate.timeout”, 1000);
```

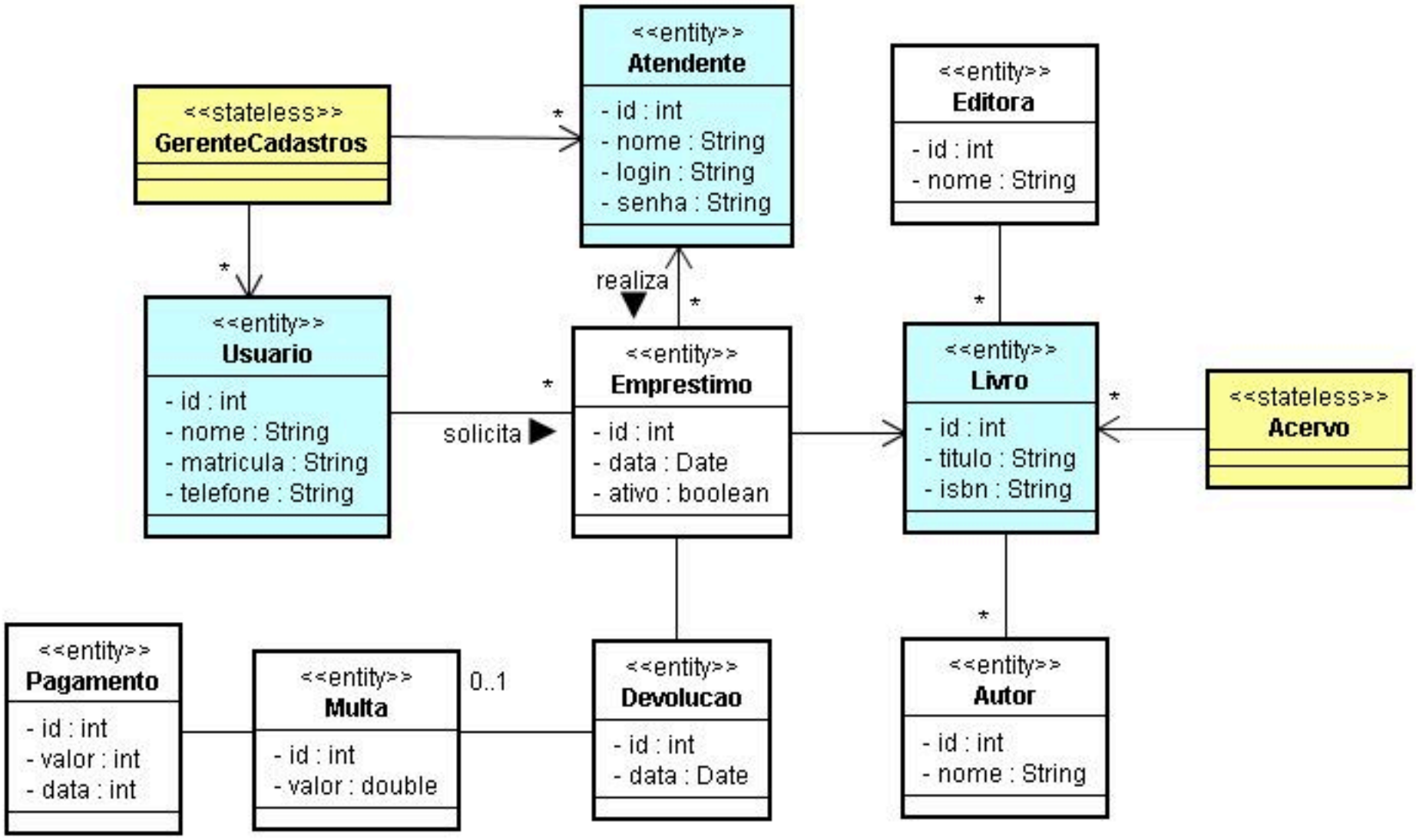
  - É permitido alterar o mecanismo de atualização do EntityManager, como por exemplo: fazer atualizações após a realização da consulta

```
Query query = manager.createQuery(“from Livro l”);
query.setFlushMode(FlushModeType.COMMIT);
```

JPA *Query Language*

- Para utilizar o recurso de consultas é necessário conhecer a linguagem própria para a realização de consultas específicas para objetos
- A JPAQL faz menção aos objetos de entidade e aos relacionamentos definidos no modelo de domínio da aplicação
- Os nomes das entidades são os mesmos do esquema abstrato, que pode ser definido:
  - Com o atributo name de @Entity ou no XML
  - Como o nome da classe do bean

JPA *Query Language*



Consultas Simples

- Exemplo de uma consulta simples:
  - Não contendo a cláusula WHERE
  - Fazendo referência a apenas uma entidade

```
SELECT OBJECT (l) FROM Livro AS l
```

  - A cláusula FROM determina quais os tipos de *bean* de entidade envolvidos na consulta
  - "AS l" atribui "l" como identificador para Livro
  - As cláusulas OBJECT e AS são opcionais

```
SELECT l FROM Livro l
```

Selecionando Propriedades e Relacionamentos

- A JPQL permite que uma consulta retorne quaisquer propriedades básicas ou de relacionamentos
- Por exemplo, poderíamos ter os nomes e matrículas de todos os usuários da biblioteca

```
SELECT u.nome, u.matricula FROM Usuario u
```

- Importante:
  - Os nomes das propriedades de persistência são identificados pelo tipo de acesso da classe de entidade (FIELD ou PROPERTY)

Selecionando Propriedades e Relacionamentos

- Para consultas que retornam mais de um item, é usado o método getResultList()
- Se o retorno implicar em propriedades de uma entidade, o resultado é um Object[]

```
Query query = entityManager.createQuery(
                                        “SELECT u.nome, u.matricula FROM Usuario u”);
List resultado = query.resultList();
Iterator it = resultado.iterator();
while (it.hasNext()) {
        Object[] objetos = (Object[]) it.next();
        String nome = (String) objetos[0];
        String matricula = (String) objetos[1];
}
```

Selecionando Propriedades e Relacionamentos

- Podem ser retornados relacionamentos

```
SELECT l.editora FROM Livro l
```

  - Selecionar todas as editoras a partir dos livros
- Também podem ser retornadas propriedades das entidades relacionadas
- A navegação pode percorrer vários relacionamentos, desde que o outro da relação não seja uma coleção

```
SELECT e.devolucao.multa.pagamento.valor FROM Emprestimo e
```

Selecionando Propriedades e Relacionamentos

- Expressões de construtor: É possível especificar um construtor dentro de uma cláusula SELECT

```
package exemplo;
public class InfoLogin {
        private String login;
        private String senha;
        public InfoLogin(String login, String senha) {
                this.login = login;
                this.senha = senha;
        } ... // também GETs e SETs...
}
```

```
SELECT new exemplo.InfoLogin(a.login,a.senha) FROM Atendente a
```

Cláusulas IN e INNER JOIN

- É muito comum que uma entidade se relacione com uma coleção de uma outra entidade
  - Exemplo: Editora com uma coleção de Livros
- Acessar elementos específicos de uma dada coleção, é uma propriedade muito importante
  - Esse acesso não é possível com o operador "."
  - Nesse caso é utilizado o operador IN para selecionar elementos de um relacionamento com uma coleção

```
SELECT l FROM Editora e, IN ( e.livros ) l
```

Cláusulas IN e INNER JOIN

- Uma vez que é possibilitada uma referência a cada elemento da coleção, propriedades destes podem ser acessadas

```
SELECT l.titulo
FROM Editora e, IN ( e.livros ) l
```

  - Mesmo que referenciem outras entidades
- A consulta acima pode ser escrita utilizando o operador INNER JOIN

```
SELECT l.titulo
FROM Editora e INNER JOIN e.livros l
```

Cláusulas IN e INNER JOIN

- A sintaxe INNER JOIN se assemelha mais com a linguagem SQL, podendo ser mais intuitiva
  - É aconselhada a sua utilização no caso do uso repetido do operador IN, como a seguir:

```
SELECT a.nome
FROM Editora e, IN ( e.livros ) l, IN ( l.autores ) a
```

  - Forma sugerida:

```
SELECT a.nome
FROM Editora e INNER JOIN e.livros l INNER JOIN l.autores a
```

  - A utilização da palavra INNER é opcional

Cláusula LEFT JOIN

- Permite a recuperação de um conjunto de entidades em que podem não existir os relacionamentos especificados na junção
  - Possibilita valores nulos para relacionamentos
  - Exemplo: recuperar todos os usuários da biblioteca e seus números de telefone associados – mesmo que existam usuários sem telefones

```
SELECT u.nome, u.matricula, t.numero
FROM Usuario u LEFT JOIN u.telefones t
```

Cláusula LEFT JOIN FETCH

- Permite pré-carregar os relacionamentos de uma entidade relacionada mesmo que a propriedades de relacionamento tenha um FetchType do tipo LAZY

```
SELECT u
FROM Usuario u LEFT JOIN FETCH u.telefones
```

Cláusula DISTINCT

- Assegura que a consulta não retorne duplicatas
  - Exemplo: a consulta que retorna todos os autores com livros cadastrados na biblioteca

```
SELECT DISTINCT a
FROM Livro l, IN ( l.autores ) a
```

Cláusula WHERE e Valores Literais

- É possível reduzir o escopo dos elementos retornados por uma consulta especificando um critério – através da cláusula WHERE
- Critérios podem ser compostos por literais
  - Literal String = caracteres entre aspas simples
  - Literal numérico = dígitos (separador ponto)
- Exemplo (livros de uma dada editora):

```
SELECT l
FROM Livro l
WHERE l.editora.nome = 'Bookman'
```

Cláusula WHERE e Valores Literais

- Exemplo com literal numérico:

```
SELECT m
FROM Multa m
WHERE m.valor = 5.50
```

- Precedência de operadores no WHERE:
  - (1º) navegação: "."
  - (2º) aritméticos: + e - (unários), *, /, + e -
  - (3º) comparações: =, >, >=, <, <=, <>, LIKE, BETWEEN, IN, IS NULL, IS EMPTY, MEMBER OF
  - (4º) lógicos: NOT, AND e OR

Cláusula WHERE e Operadores Aritméticos

- Podem ser realizadas operações aritméticas simples antes da realização das comparações
  - Exemplo: selecionar as multas que ultrapassaram uma margem de segurança

```
SELECT m
FROM Multa m
WHERE ( m.valor * 1.10 ) > 2.50
```

  - As regras aplicadas às operações aritméticas são as mesmas da linguagem Java
  - Os cálculos não podem envolver atributos String

Cláusula WHERE e Operadores Lógicos e Comparações

- Operadores lógicos:
  - São operados da mesma maneira que no SQL
  - Só avaliam operações booleanas
- Os operadores de comparação pode ser utilizados com valores numéricos e Strings
  - Para identificadores boolean ou que representam objetos de entidade só se aplicam: = e <>

```
SELECT p
FROM Pagamento p
WHERE p.valor >= 5.50 AND p.valor <= 10.50
```

Cláusula WHERE com BETWEEN

- BETWEEN representa um operador inclusivo que especifica um intervalo de valores
- Só pode ser aplicada a tipos primitivos numéricos e suas classes correspondentes

```
SELECT p
FROM Pagamento p
WHERE p.valor BETWEEN 5.50 AND 10.50
```

```
SELECT p
FROM Pagamento p
WHERE p.valor NOT BETWEEN 5.50 AND 10.50
```

Cláusula WHERE com IN

- O operador IN condicional é diferente do operador IN usado na cláusula FROM
  - Mais uma razão para preferir a cláusula JOIN
- Na cláusula WHERE, o operador IN testa a associação em uma lista de valores literais

```
SELECT e FROM Endereco e
WHERE e.estado IN ('CE', 'PB', 'PE', 'RN')
```

```
SELECT e FROM Endereco e
WHERE e.estado NOT IN ('CE', 'PB', 'PE', 'RN')
```

Cláusula WHERE com IS NULL

- Permite testar se um relacionamento é nulo

```
SELECT u
FROM Usuario u
WHERE u.endereco IS NULL
```

  - A negação da cláusula anterior, seria:

```
SELECT u
FROM Usuario u
WHERE u.endereco IS NOT NULL
```

  - IMPORTANTE: a verificação "se não é nulo" também pode ocorrer em parâmetros de entrada fornecidos para a consulta

Cláusula WHERE com IS EMPTY

- Permite que a consulta teste se um relacionamento baseado em coleção é vazio
  - Esse tipo de relacionamento nunca será nulo
- Exemplo (editoras sem livros cadastrados):

```
SELECT e
FROM Editora e
WHERE e.livros IS EMPTY
```

  - A negação da consulta anterior seria:

```
SELECT e
FROM Editora e
WHERE e.livros IS NOT EMPTY
```

Cláusula WHERE com MEMBER OF

- Permite verificar se uma entidade é membro de um relacionamento específico baseado em coleção
- Exemplo (editoras que publicaram livros de um dado autor):

```
SELECT e
FROM Editora e, IN ( e.livros ) l, Autor autor
WHERE autor = :umDadoAutor  AND  autor MEMBER OF l.autores
```

  - A negação equivalente seria: NOT MEMBER OF

Cláusula WHERE com LIKE

- Permite que a consulta selecione campos do tipo String que correspondem a um padrão específico
  - Exemplo (usuários com nomes com hífen):

```
SELECT u FROM Usuario u WHERE u.nome LIKE '%-%'
```

  - Caracteres especiais são utilizados na definição do padrão desejado, são eles:
    - % – representa qualquer seqüência de caracteres
    - _ – representa qualquer caractere único
    - \ – escape, permite referenciar os símbolos "%" e "_"

Expressões Funcionais

- A JPAQL define várias funções para processar Strings e valores numéricos
- Expressões funcionais na cláusula WHERE:
  - LOWER(String)
    - Converte uma string em letras minúsculas
  - UPPER(String)
    - Converte uma string em letras maiúsculas
  - TRIM([[LEADING | TRAILING | BOTH] [caractere] FROM] String)
    - Remove um determinado caractere no início, fim ou ambos de uma string. (o caractere padrão é o espaço)

Expressões Funcionais

- Expressões funcionais na cláusula WHERE:
  - CONCAT(String1, String2)
    - Retorna uma *String* resultado da concatenação das duas *Strings* passadas como parâmetro
  - LENGTH(String)
    - Retorna um inteiro indicando o tamanho da *String*
  - LOCATE(String1, String2[, inicio])
    - Retorna um inteiro indicando a posição em que a primeira *String* está localizado dentro da segunda *String*
    - Se estiver presente, o "inicio" define a posição inicial que iniciarão as comparações

Expressões Funcionais

- Expressões funcionais na cláusula WHERE:
  - SUBSTRING(String1, inicio, tamanho)
    - Retorna uma *String* que consiste de "tamanho" caracteres da *String* fornecida, a partir da posição "inicio"
  - ABS(numero)
    - Retorna o valor absoluto de um número
  - SQRT(double)
    - Retorna a raiz quadrada de um número real (double)
  - MOD(int, int)
    - Retorna o resto da divisão do primeiro (int) pelo segundo

Funções que Retornam Datas e Horas

- A JPAQL possui três funções que podem retornar a data e a hora atuais:
  - CURRENT_DATE
  - CURRENT_TIME
  - CURRENT_TIMESTAMP

```
SELECT pag FROM Pagamento pag WHERE pag.data = CURRENT_DATE
```

Funções Agregadas na Cláusula SELECT

- COUNT(identificador ou relacionamento)
  - Retorna o número de itens no conjunto final de resultados da consulta
  - Exemplo1 (quantos usuários moram no RN):

```
SELECT COUNT( u ) FROM Usuario u
WHERE u.endereco.estado = 'RN'
```

  - Exemplo2 (quantos CEPs iniciam com "59"):

```
SELECT COUNT( u.endereco.cep ) FROM Usuario u
WHERE u.endereco.cep LIKE '59%'
```

Funções Agregadas na Cláusula SELECT

- MAX(expressão de interligação)
- MIN(expressão de interligação)
  - Permitem localizar o maior e o menor valor de uma coleção – o resultado será do tipo avaliado
  - Exemplo (maior valor de multa já pago):

```
SELECT MAX( multa.valor )  FROM Multa multa
```

Funções Agregadas na Cláusula SELECT

- AVG(numérico)
- SUM(numérico)
  - Permitem o cálculo da média e soma de valores numéricos (propriedades específicas)
  - Aplicadas a expressões que terminem em tipos primitivos (byte, long, float, etc.) ou *Wrapperes*
  - Exemplos:

```
SELECT AVG( multa.valor )  FROM Multa multa
```

```
SELECT SUM ( multa.valor )  FROM Multa multa
```

Funções Agregadas na Cláusula SELECT

- O operador DISTINCT pode ser usado com qualquer das funções agregadas

```
SELECT DISTINCT COUNT( usr.endereco.cep )
FROM Usuario usr WHERE usr.endereco.cep LIKE '59%'
```

- Qualquer campo com valor nulo é eliminado do conjunto de resultados operados pelas funções agregadas
- As funções AVG(), SUM(), MAX() e MIN() retornam null ao avaliar uma coleção vazia

Cláusula ORDER BY

- Permite especificar a ordem das entidades em uma coleção retornada por uma consulta
  - Mesma semântica que na SQL
  - Exemplo (usuários em ordem alfabética):

```
SELECT usr FROM Usuario usr ORDER BY usr.nome
```

- A ordenação pode ser solicitada com ou sem a cláusula WHERE

```
SELECT u FROM Usuario u
WHERE u.endereco.cidade ='Natal' AND u.endereco.bairro ='Tirol'
ORDER BY usr.nome
```

Cláusula ORDER BY

- A ordem padrão é a ascendente
- A ordem desejada pode ser explicitada:
  - ASC – ascendente
  - DESC – descendente

```
SELECT usr FROM Usuario usr  ORDER BY usr.nome DESC
```

- Pode-se definir vários critérios de ordenação

```
SELECT usr FROM Usuario usr  ORDER BY usr.nome ASC, usr.matricula DESC
```

- Dependendo do banco de dados os valores nulos são considerados antes ou depois

Cláusula ORDER BY

- Os campos usados na cláusula ORDER BY devem ser campos básicos
  - Ou campos básicos em entidades relacionadas
  - Não é permitida a utilização de campos básicos da entidade que NÃO está sendo selecionada
  - Exemplo de consulta legal:

```
SELECT usr.endereco FROM Usuario usr  ORDER BY usr.endereco.cep
```

  - Exemplo de consulta ilegal:

```
SELECT usr FROM Usuario usr  ORDER BY usr.endereco.cep
```

GROUP BY e HAVING

- Comumente utilizadas para a classificação do resultado para aplicação de função agregada
  - Permite o agrupamento das entidades por categoria
  - Exemplo (quantidade de livros por editora):

```
SELECT edit.nome, COUNT (liv)  FROM Editora edit LEFT JOIN edit.livros liv
GROUP BY edit.nome
```

  - A consulta retorna o nome da editora e a quantidade de livros associados a ela
  - Com o LEFT JOIN, as editoras que não possuem livros são retornadas com "zero" livros

GROUP BY e HAVING

- A cláusula GROUP BY deve especificar uma das colunas que está retornando na consulta
  - A sintaxe de GROUP BY é ainda mais interessante se a combinarmos com uma expressão construtor
  - Exemplo (informações sobre os usuários):

```
SELECT new SumarioUsuario
                (usr.nome, COUNT(emp), SUM(emp.devolucao.multa.valor))
FROM Usuario usr LEFT JOIN usr.emprestimos emp
GROUP BY usr.nome
```

GROUP BY e HAVING

- A cláusula HAVING é utilizada em conjunto com a GROUP BY como um "filtro", restringindo o resultado final
  - São usados na cláusula HAVING expressões funcionais agregadas sobre os identificadores utilizados na cláusula SELECT
  - Exemplo:

```
SELECT edit.nome, COUNT (liv)
FROM Editora edit LEFT JOIN edit.livros liv
GROUP BY edit.nome
HAVING COUNT(liv) > 20
```

Subconsultas

- JPA permite o uso de consultas SELECT dentro de outras consultas (otimizar)
  - São suportadas sub-consultas nas cláusulas WHERE e HAVING
  - Exemplo (quantidade de multas com valor acima da média das multas aplicadas):

```
SELECT COUNT(mul) FROM Multa mul
WHERE mul.valor > (SELECT AVG(m.valor) FROM Muta m)
```

Subconsultas

- Em uma sub-consulta podem também ser referenciados identificadores da cláusula FROM da consulta externa
  - Exemplo (usuários que já pagaram mais de 100 reais de multa na biblioteca):

```
SELECT usr FROM Usuario usr
WHERE 100 >
        ( SELECT SUM(emp.devolucao.multa.valor) FROM usr.emprestimos emp )
```

Subconsultas

- Podemos quantificar os resultados de uma sub-consulta com expressões lógicas:
  - ALL – retorna true se todos os elementos retornados na sub-consulta correspondem a uma dada expressão condicional
  - ANY – retorna true se algum dos elementos retornados na sub-consulta correspondem a uma dada expressão condicional

```
SELECT usr FROM Usuario usr WHERE 0 < ANY
        ( SELECT SUM(emp.devolucao.multa.valor) FROM usr.emprestimos emp )
```

Subconsultas

- O operador EXISTS retorna true se o resultado da sub-consulta consistir em um ou mais valores
  - Se nenhum valor é retornado o resultado é false
  - Exemplo:

```
SELECT usr FROM Usuario usr
WHERE EXISTS ( SELECT emp FROM usr.emprestimos emp
                    WHERE emp.devolucao.multa.valor != 0 )
```

UPDATE e DELETE em Lote

- JPA permite que sejam realizadas operações de atualização e remoção em lote
  - Muito cuidado, pois é possível criar inconsistências entre o banco e as entidades gerenciadas – depende do fornecedor

```
UPDATE Usuario usr SET usr.bonus = (usr.bonus + 10)
WHERE EXISTS ( SELECT emp FROM usr.emprestimos emp
                        WHERE emp.ativo = true )
```

```
DELETE FROM Livro liv
WHERE EXISTS ( SELECT aut FROM liv.autores aut
                        WHERE aut.nome = 'Deitel' )
```

Consultas Nativas

- Somente em casos em que a JPAQL não satisfizer a necessidade de consulta usa-se SQL
  - Como tirar proveito de certas capacidades proprietárias só disponíveis em um banco de dados
- As consultas nativas (SQL) pode retornar:
  - (1) Entidades
  - (2) Valores de colunas
  - (3) Entidades e valores de colunas
- O EntityManager fornece os métodos específicos para criar tais consultas

Consultas Nativas

- (1) Consultas nativas escalares
  - Query createNativeQuery(String sql)
  - Esta consulta retorna valores escalares no mesmo modelo que uma JPAQL retorna valores escalares (Object[])

Consultas Nativas

- (2) Consultas nativas simples de entidade
  - Query createNativeQuery

    (String sql, class classeDeEntidade)
  - O resultado é implicitamente mapeado para uma entidade (ou conjunto de entidades) segundo as informações de mapeamento da dada entidade
  - É esperado que as colunas retornadas correspondam perfeitamente ao mapeamento especificado na classe de entidade

Consultas Nativas

- (3) Consultas nativas complexas
  - Query createNativeQuery
    (String sql, String nomeDeMapeamento)
  - Podem ser retornadas múltiplas entidades e valores escalares de coluna simultaneamente
  - O atributo "nomeDeMapeamento" referencia uma @javax.persistence.SqlResultSetMapping
    - Com essa anotação é definida a maneira como os resultados se encaixam no modelo O/R
    - Essa anotação é usada em conjunto com outras que permitem as definições de mapeamento

Consultas Nativas Complexas

```
public @interface SqlResultSetMapping {
        String name( );
        EntityResult[] entities( ) default { };
        ColumnResult[] columns( ) default { };
}
```

```
public @interface EntityResult {
        Class entityClass( );
        FieldResult[] fields( ) default { };
        String discriminatorColumn( ) default “”;
}
```

```
public @interface FieldResult {
        String name( );
        String column( );
}
```

```
public @interface ColumnResult {
        String name( );
}
```

Consultas Nativas Complexas

```
@Entity
@SqlResultSetMapping(name="usuarioENumeroCelular",
        entities={@EntityResult(entityClass=Usuario.class),
                  @EntityResult(entityClass=Telefone.class,
                  fields={@FieldResult(name="id", column="T_ID"),
                          @FieldResult(name="numero", column="numero")
                  })
        })
public class Usuario { ... }
```

```
{ ...
        Query query = manager.createNativeQuery(
                  "SELECT u.id,u.nome,u.matricula,t.id AS T_ID, t.numero"+
                  " FROM TABELA_USUARIO u, TABELA_TELEFONE t"+
                  " WHERE c.id = t.usuario_id AND t.tipo = 'celular'",
                  "usuarioENumeroCelular");
... }
```

Consultas Nativas Complexas

```
@Entity
@SqlResultSetMapping(name="edioraENumDeLivros",
        entities={@EntityResult(entityClass=Editora.class,
                fields={@FieldResult(name="id", column="ID"),
                        @FieldResult(name="nome", column="NOME")}
        }, columns={@ColumnResult(name="numLivros")}
)
public class Editora { ... }
```

```
{ ...
        Query query = manager.createNativeQuery(
                "SELECT e.ID, e.NOME, COUNT(LIVRO.ID) AS numLivros"+
                " FROM EDITORA e LEFT JOIN LIVRO ON e.ID = LIVRO.ED_ID"+
                " GROUP BY e.ID", "edioraENumDeLivros");
... }
```

Consultas Nomeadas

- JPA permite a criação de consultas JPAQL predefinidas e atribuir nomes a estas consultas
- É recomendável pré-declarar as consultas pela mesma razão que criamos variáveis de constantes String – reutilizar várias vezes
- Para definir tais consultas usamos a anotação @javax.persistence.NamedQuery

Consultas Nomeadas

```
public @interface NamedQuery {
        String name( );
        String query( );
        QueryHints[] hints( ) default { };
}
```

```
public @interface QueryHint {
        String name( );
        String value( );
}
```

```
public @interface NamedQueries {
        NamedQuery[] value( );
}
```

Consultas Nomeadas

```
@Entity
@NamedQueries({
        @NamedQuery(name="totalDeLivros",
                query="SELECT COUNT(liv) FROM Livro liv"),
        @NamedQuery(name="findLivrosPorEditora",
                query="SELECT liv FROM Livro liv WHERE liv.editora = :editora"),
})
public class Livro { ... }
```

```
{ ...
        Query query = em.createNamedQuery("findLivrosPorEditora");
        query.setParameter("editora", editora);
... }
```

Consultas Nativas Nomeadas

- @javax.persistence.NamedNativeQuery é usada para pré-definir consultas SQL nativas

```
public @interface NamedNativeQuery {
        String name( );
        String query( );
        Class resultClass( ) default void.class;
        String resultSetMapping( ) default “”;
}
```

```
public @interface NamedNativeQueries {
        NamedNativeQuery[] value( );
}
```

Consultas Nativas Nomeadas

```
@NamedNativeQueries({
        @NamedNativeQuery(name="sumarioEditora",
                query="SELECT e.ID, e.NOME, COUNT(LIVRO.ID) AS numLivros
                   FROM EDITORA e LEFT JOIN LIVRO ON e.ID = LIVRO.ED_ID
                   GROUP BY e.ID",
                resultSetMapping="edioraENumDeLivros")
})
@SqlResultSetMapping(name="edioraENumDeLivros",
        entities={@EntityResult(entityClass=Editora.class,
                fields={@FieldResult(name="id", column="ID"),
                        @FieldResult(name="nome", column="NOME")}
        }, columns={@ColumnResult(name="numLivros")
})
@Entity
public class Editora { ... }
```