

FIA/P GRADUAÇÃO

ENTERPRISE APPLICATION DEVELOPMENT

Prof. Me. Thiago T. I. Yamamoto

#06 - JAVA PERSISTENCE QUERY LANGUAGE

TRAJETÓRIA



JPA Introdução



JPA API



Design Patterns e JUnit



Relacionamentos



JPQL

#06 - AGENDA

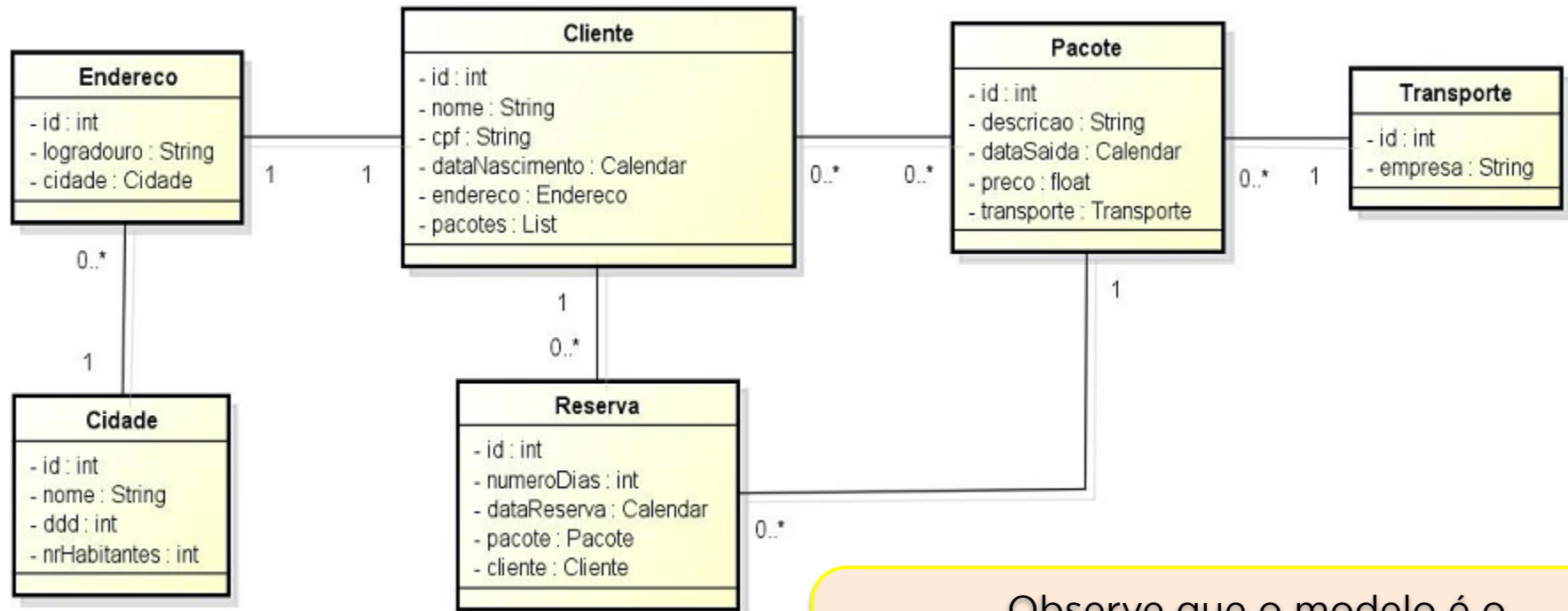


- Introdução ao JPQL
- Interfaces Query e TypedQuery
- Passagem de parâmetros
- Paginação de resultados e projeções
- Funções gerais e agregadoras
- Named Queries
- Queries nativas

- 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;
- É possível realizar outras queries além de buscas e chamadas a **procedures**;



- O modelo abaixo será utilizado para a realização de consultas:



Observe que o modelo é o **diagrama de classes** e **não a modelagem do banco de dados!**

Importe o projeto, altere o usuário e senha e execute a classe **PopulaBanco**.



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(c) 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!`

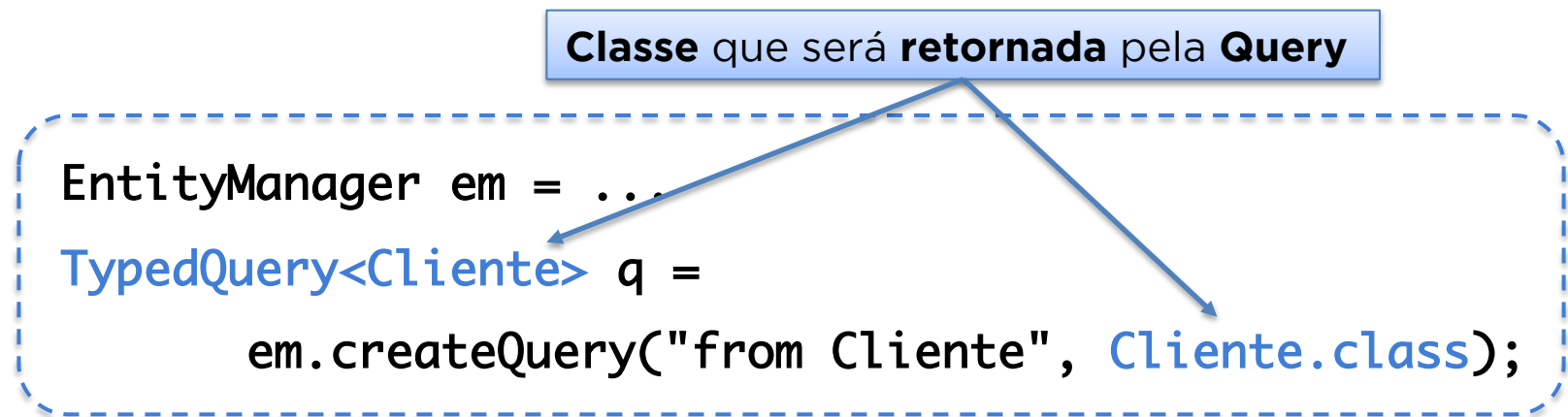
- Os métodos para as consultas encontram-se na interface **Query**;
- Para obter-se uma instância de **Query** devemos acionar um dos métodos abaixo a partir de uma instância **EntityManager**:
 - **createQuery(String P1)**: cria uma consulta com base no JPQL fornecido no **P1**;
 - **createNamedQuery(String 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**.

Nome da **entidade**!
Não da **tabela**!

```
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**;



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

```
EntityManager em = ...  
Query query = em.createQuery("from Cliente");  
List<ClienteEntity> clientes = query.getResultList();
```

- O método **setParameter()** deve ser utilizado para passar parâmetros para a query, evitando assim ataque **SQL Injection**;
- Os **parâmetros** da query são **nomeados** através do caractere ":";
- Os **parâmetros** passados podem ser tanto tipos **primitivos** quanto **entidades**;
- No exemplo abaixo o **cliente** com o id = 2 será considerado como **parâmetro pela consulta**;

Parâmetro definido na query

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

```
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 = ...
```

```
Cidade c1 = em.find(Cidade.class, 1);
```

```
Cidade c2 = em.find(Cidade.class, 2);
```

```
List<Cidade> cs = new ArrayList<Cidade>();
```

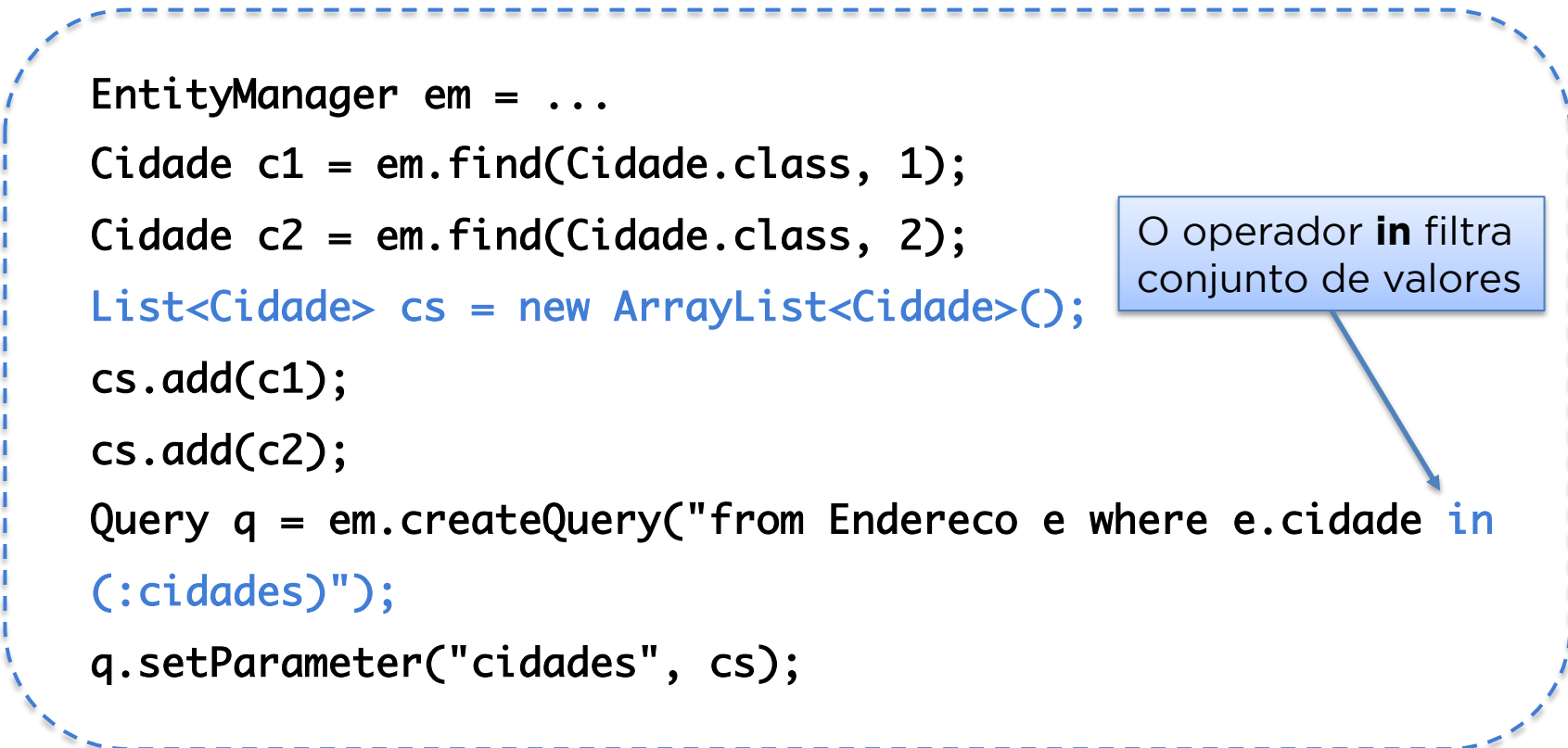
```
cs.add(c1);
```

```
cs.add(c2);
```

```
Query q = em.createQuery("from Endereco e where e.cidade in  
(:cidades)");
```

```
q.setParameter("cidades", cs);
```

O operador **in** filtra
conjunto de valores



CODAR 1!

Crie as seguintes **consultas**:



1. Obter todos os clientes;
2. Obter todos os clientes por parte do nome;
3. Obter todos os pacotes por um transporte específico (objeto transporte);
4. Obter todos clientes localizados por estado;
5. Obter todos os clientes que efetuaram reservas em uma quantidade de dias específica.



- Uma **consulta que retorne** 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:

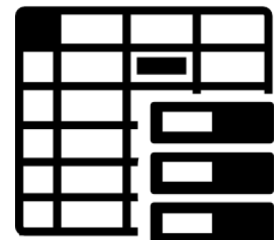
```
Query q = em.createQuery("from Cliente");
```

```
q.setMaxResults(5);
```

```
q.setFirstResult(0);
```

A Query **retorna** no **máximo 5** entidades

Retorna 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;

```
EntityManager em = //...  
Query q = em.createQuery("select c.nome, c.uf from Cidade c");  
List<Object[]> resultado = q.getResultList();  
for (Object[] obj : resultado)  
    System.out.println(obj[0] + ", " + obj[1]);
```

Nome

UF

- Caso o retorno da consulta seja **um único valor** não é necessário utilizar o *array* de objetos;
- Por exemplo, em funções agregadoras (SUM, COUNT, etc...);

```
EntityManager em = //..  
TypeQuery<Long> q = em.createQuery(  
    "select count(*) from Cliente", Long.class);  
Long total = q.getSingleResult();  
System.out.println(total);
```



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 *List*)

Ordenação

from Cliente c order by c.nome

Like

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

Sub-Consultas

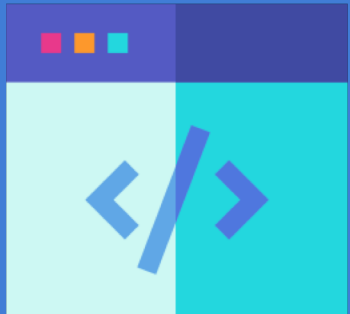
```
from Reserva r where r.numDias = (select max(r2.numDias)  
from Reserva r2)
```

Sub-Consultas com operador NOT IN

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

CODAR 2!

Escrever o conjunto de **métodos** abaixo nas respectivas classes **DAO**:



1. **buscarPorDatas(Calendar inicio, Calendar fim):**
retorna todos os pacotes cuja data de saída esteja no intervalo especificado nos parâmetros;
2. **buscar(String nome, String cidade):** retorna os clientes que possuam parte do nome o texto informado como parâmetro e que tenham algum endereço por parte do nome de cidade informado;
3. **buscarPorEstados(List<String> estados):** retorna todos os clientes conforme os estados passados como parâmetro;



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 e
- **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

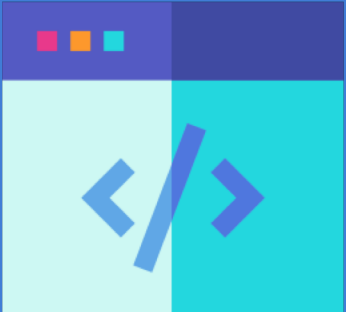
CODAR 3!

Ajuste o método **criado** anteriormente para:

1. Pesquisar os clientes por nome e não diferenciar letras maiúsculas de minúsculas. Retornar o resultado ordenado pelo nome do cliente;

Desenvolva novas funções para:

1. Contar a quantidade de clientes de um estado específico;
2. A soma dos preços dos pacotes por um transporte específico;





NAMED QUERIES

- Um sistema 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**, 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**, precisamos utilizar o método **createNamedQuery(String nome)** da interface Query, passando como parâmetro o nome da consulta a ser executada;

Nome da query que será utilizada no **DAO** para criar a consulta

```
@NamedQuery(name= "Cliente.porNome",  
query="from Cliente c where c.nome like :nome")
```

```
@Entity
```

```
public class Cliente { ... }
```

```
Query q = em.createNamedQuery("Cliente.porNome");
```

```
q.setParameter("nome", "%JOAO%");
```

```
List<Cliente> 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  
@Table(name="TAB_CLIENTE")  
public class Cliente {  
    //...  
}
```



CONSULTAS NATIVAS

- Pode-se criar **consultas nativas** do 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 **List<Object[]>** onde cada posição do array corresponde a uma coluna na projeção da consulta;

SQL do Banco de dados, não é **JQPL**

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



VOCÊ APRENDEU..



- Realizar pesquisas com **JPQL**, que são consultas parecidas com SQL, porém utiliza o **nome das classes** ao invés das tabelas;
- Os métodos **getSingleResult** e **getResultList**;
- Passar **parâmetros** para as queries, **realizar a paginação** e **projeção dos resultados**;
- **Funções gerais**, para agrupar, ordenar e etc..
- **Funções agregadoras**, para realizar soma, média e etc..
- Trabalhar com **named queries**, que são estruturas pré-compiladas;
- Criar **query nativa** do banco de dados;

Copyright © 2013 – 2019
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”*