

Universidade Presbiteriana Mackenzie
Faculdade de Computação e Informática

Encapsulamento das operações de persistência em banco de dados (parte 2)

NOTAS DE AULA - Teoria 09

Linguagem de Programação II
2º semestre de 2015
Versão 1.0

Entidade

O termo **entidade** é utilizado para um objeto de domínio que é persistido na base.

Uma classe de entidade representa uma tabela na base de dados relacional.

Uma instância da entidade corresponde a uma linha naquela tabela.

Prepared statement



Prepared statement

Suponha que queremos executar um comando SQL de inserção. Por exemplo:

```
INSERT INTO titulares VALUES (2, 'Marcos Antônio', '22333444', '09988877765');
```

Quando vamos executar este comando em um programa, geralmente as informações que serão inseridas estão em variáveis. Desta forma, teríamos um trecho de código semelhante a:

```
String sql = "INSERT INTO titulares VALUES (" + id + ", '" + nome + "', '"  
            + rg + "', '" + cpf + "')";  
Statement st = connection.createStatement();  
st.executeUpdate(sql);
```

Note que há o inconveniente de lembrarmos de detalhes, como digitar o apóstrofe quando o valor é endereçado a uma coluna que aceita strings.

Prepared statement (cont.)

Ao invés de construir um comando SQL separado para cada execução, podemos criar um *Prepared Statement*. Isto permite criar a sentença uma única vez e executá-la diversas vezes, trocando a cada execução o valor das variáveis da sentença.

Exemplo de preparação:

```
String sql = "INSERT INTO titulares VALUES (?, ?, ?, ?)";  
PreparedStatement stInsert = connection.prepareStatement(sql);
```

Cada vez que for necessário executar a inserção, basta utilizar:

```
stInsert.setInt(1, id);  
stInsert.setString(2, nome);  
stInsert.setString(3, rg);  
stInsert.setString(4, cpf);  
stInsert.executeUpdate();
```

Além de tornar o código mais legível, isto também pode trazer melhorias de performance.

Geração automática de campo identificador

Geração automática de campo identificador

Em muitas tabelas é necessário criar um campo numérico de valor único que pode ser gerado sequencialmente. Para que o próprio gerenciador de base de dados se encarregue disto, sem que o usuário ou a aplicação precise definir o valor para este campo, podemos declarar este campo conforme o exemplo abaixo:

```
CREATE TABLE titulares (  
    id BIGINT NOT NULL  
        PRIMARY KEY  
        GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1),  
    nome VARCHAR(255) NOT NULL,  
    rg VARCHAR(32) NOT NULL,  
    cpf VARCHAR(32) NOT NULL  
);
```

A inserção de registros nesta tabela pode ser realizada sem a necessidade de fornecer o valor para **id**, pois já será gerado automaticamente pelo banco de dados.

```
INSERT INTO titulares (nome, rg, cpf) VALUES ('Marcos Antônio', '22333444', '09988877765');
```

Se fizermos a consulta na tabela, veremos que temos um único registro com o campo **id** igual a 1.

Mapeamento bidirecional **1 para muitos / muitos para 1**

Observação: Os exemplos utilizam versões simplificadas das tabelas e classes para dar ênfase no assunto que está sendo explicado.

Mapeamento bidirecional 1 para muitos / muitos para 1

Sessao e **Ingresso** são entidades que possuem uma associação deste tipo. Neste caso, um ingresso referencia uma única sessão, e uma sessão referencia uma coleção de ingressos.

Script de criação da tabela de **sessoes**:

```
CREATE TABLE sessoes (  
    id BIGINT NOT NULL PRIMARY KEY  
        GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1),  
    horario time NOT NULL  
    /* outras colunas */  
);
```

Script de criação da tabela de **ingressos**:

```
CREATE TABLE ingressos (  
    id BIGINT NOT NULL PRIMARY KEY  
        GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1),  
    tipo VARCHAR(8) NOT NULL,  
    sessao_id BIGINT NOT NULL,  
    Foreign Key (sessao_id) REFERENCES sessoes (id)  
    /* outras colunas */  
);
```

Note que a tabela de **ingressos** contém um campo com o **id** da sessão.

Mapeamento bidirecional 1 para muitos / muitos para 1 (cont.)

A classe **Sessao** deve ter uma lista de ingressos:

```
public class Sessao {  
    private long id;  
    private LocalTime horario;  
    private List<Ingresso> ingressos;  
  
    public void setIngressos(List<Ingresso> ingressos) {  
        this.ingressos = ingressos;  
    }  
  
    public List<Ingresso> getIngressos() {  
        return ingressos;  
    }  
  
    public void addIngresso(Ingresso ingresso) {  
        ingressos.add(ingresso);  
    }  
  
    // outros atributos e métodos  
}
```

Note que a classe não contém uma lista de **ids** de ingressos, mas sim as referências para as instâncias de **Ingresso**.

Mapeamento bidirecional 1 para muitos / muitos para 1 (cont.)

A classe **Ingresso** deve ter uma referência para a sua sessão:

```
public class Ingresso {  
    public enum Tipo { MEIA, INTEIRA };  
    private long id;  
    private Tipo tipo;  
    private Sessao sessao;  
  
    // outros atributos e métodos  
}
```

Note que a classe não armazena o **id** da sessão, mas sim a referência de uma instância de **Sessão**.

Acesso *lazy* ("preguiçoso")



Acesso *lazy* ("preguiçoso")

Quando um objeto contém atributos que referenciam outros objetos, podemos utilizar a estratégia de acesso *lazy* que consiste em carregar estes objetos quando realmente forem ser utilizados.

Se aplicarmos, como exemplo, a estratégia *lazy* para acessar a entidade **Sessao**, isto quer dizer que a consulta das sessões não traria a lista de ingressos preenchida.

Acesso *lazy* (cont.)

Se implementarmos as classes DAO utilizando a estratégia *lazy*, o trecho abaixo geraria uma exceção **NullPointerException** uma vez que **getIngressos** retornaria **null**.

```
List<Sessao> sessoes = daoSessoes.buscarTodos();
Sessao primeiraSessao = sessoes.get(0);
System.out.println("Ingressos na primeira sessão: " +
    primeiraSessao.getIngressos().size());
```

Para obter o conjunto de ingressos pertencentes àquela sessão seria necessário realizar uma consulta adicional:

```
List<Sessao> sessoes = daoSessoes.buscarTodos();
Sessao primeiraSessao = sessoes.get(0);
List<Ingresso> ingressos = daoIngressos.buscarPela(primeiraSessao);
primeiraSessao.setIngressos(ingressos);
System.out.println("Ingressos na primeira sessão: " +
    primeiraSessao.getIngressos().size());
```

Esta é a forma manual de buscar as informações dos atributos. É possível também fazer a busca de forma automática, na primeira vez em que os valores dos atributos *lazy* são solicitados, mas isto tornaria o código das entidades um pouco mais rebuscado.



Uso de uma base Java DB criada em outro computador

Uso de uma base Java DB criada em outro computador

No primeiro computador

Quando você cria a base utilizando o NetBeans, você pode definir a **LOCALIZAÇÃO DA BASE DE DADOS**.

Vamos supor que você escolheu criar a base de dados **sistema_ingressos** na pasta **D:\JavaDb**.

Após a criação da base, passou a existir a pasta **D:\JavaDb\sistema_ingressos**.

Para trabalhar com a base de dados em outro computador, copie (ou envie) toda a pasta **D:\JavaDb**.

Uso de uma base Java DB criada em outro computador (cont.)

No outro computador

No outro computador, copie a pasta para o seu disco. Vamos supor que, no seu computador, você copiou o conteúdo para **C:\Users\JavaDb** (**sistema_ingressos** deverá estar dentro desta pasta).

No NetBeans, vá na aba **Serviços**, clique em **Java DB** com o botão direito do mouse e selecione **Propriedades**. Altere o valor do campo **Localização da base de dados** para **C:\Users\JavaDB** (a pasta onde você efetuou a cópia) e pressione OK. A base **sistema_ingressos** deveria aparecer imediatamente abaixo de **Java DB**.

Bom estudo!

