

Acesso a dados com Java

Tiago T. Wirtti

7 de agosto de 2014

Resumo

Neste texto serão abordadas as técnicas de persistência de dados em banco de dados relacional usando Java. O SGBD (Sistema Gerenciador de Banco de Dados) utilizado será o MySQL. O primeiro passo é apresentar informalmente a linguagem SQL (*Structured Query Language*). Em seguida explicaremos como conectar o banco de dados com a plataforma Java. O terceiro passo é construir uma camada de persistência que desacople o gerenciamento de dados da lógica de negócio. Por último construiremos uma pequena aplicação com interface gráfica que realiza um CRUD sobre uma tabela simples.

1 Apresentando (brevemente) a linguagem SQL

Nesta seção faremos uma introdução rápida à linguagem SQL. SQL, do inglês, *Structured Query Language*, ou, em português, Linguagem de Consulta Estruturada, é a linguagem de pesquisa declarativa padrão para banco de dados relacional. Muitas das características originais do SQL foram inspiradas na álgebra relacional. A linguagem SQL surgiu originalmente em 1970 nos laboratórios da IBM [1].

Agora vamos conhecer um pouco de SQL. Para simplificar o processo, vamos criar duas tabelas simples e depois realizar as quatro operações básicas (incluir, alterar, listar e excluir) sobre elas.

1.1 Criando uma tabela no MySQL Workbench

Partiremos de um modelo previamente construído no MySQL Workbench [2]. O modelo está no arquivo `veiculos.mwb`, disponível com este material. A Figura 1 mostra o modelo composto por duas tabelas: `veiculo` e `marca`. Observe que o campo `MARCA_VEC_FK` na tabela `veiculo` é chave estrangeira associada ao campo `ID_MAR`, na tabela `marca`. O SQL correspondente, que é gerado de forma automática pelo MySQL Workbench, é mostrado na Figura 2. É importante observar que na Figura 2 (marcador 1), o banco de dados `veiculos` destruído (se existir um `-IF EXISTS`) e é criado logo em seguida. O marcador 2, na mesma figura, mostra a destruição da tabela `marca`, se a mesma existir, e, em seguida, a sua criação (marcador 3). O marcador 4 mostra a definição de uma restrição (`CONSTRAINT`)

que associa a chave estrangeira MARCA_VEC_FK à chave primária ID_MAR. Por último, no marcador 5, um índice é criado para melhorar o desempenho da busca na tabela veiculo. Este *script* encontra-se no arquivo `veiculos.sql`, disponível com este material. Atenção: o *script* sempre vai destruir o banco e recriá-lo, portanto os dados gravados serão perdidos.

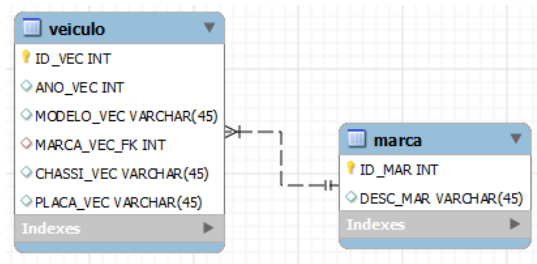


Figura 1: Modelo de dados relacional.

```

1 ➡ DROP SCHEMA IF EXISTS `veiculos`;
   CREATE SCHEMA IF NOT EXISTS `veiculos`
   DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci ;
   USE `veiculos` ;

2 ➡ -- Table `veiculos`.`marca`
   DROP TABLE IF EXISTS `veiculos`.`marca` ;

3 ➡ CREATE TABLE IF NOT EXISTS `veiculos`.`marca` (
   `ID_MAR` INT NOT NULL,
   `DESC_MAR` VARCHAR(45) NULL,
   PRIMARY KEY (`ID_MAR`))
   ENGINE = InnoDB;

   -- Table `veiculos`.`veiculo`
   DROP TABLE IF EXISTS `veiculos`.`veiculo` ;

4 ➡ CREATE TABLE IF NOT EXISTS `veiculos`.`veiculo` (
   `ID_VEC` INT NOT NULL,
   `ANO_VEC` INT NULL,
   `MODELO_VEC` VARCHAR(45) NULL,
   `MARCA_VEC_FK` INT NULL,
   `CHASSI_VEC` VARCHAR(45) NULL,
   `PLACA_VEC` VARCHAR(45) NULL,
   PRIMARY KEY (`ID_VEC`),
   CONSTRAINT `MARCA_VEC_FK`
   FOREIGN KEY (`MARCA_VEC_FK`)
   REFERENCES `veiculos`.`marca` (`ID_MAR`)
   ON DELETE NO ACTION
   ON UPDATE NO ACTION)
   ENGINE = InnoDB;

5 ➡ CREATE INDEX `MARCA_VEC_FK_idx` ON `veiculos`.`veiculo` (`MARCA_VEC_FK` ASC);

```

Figura 2: SQL do modelo relacional.

1.2 Fazendo o "CRUD" com SQL

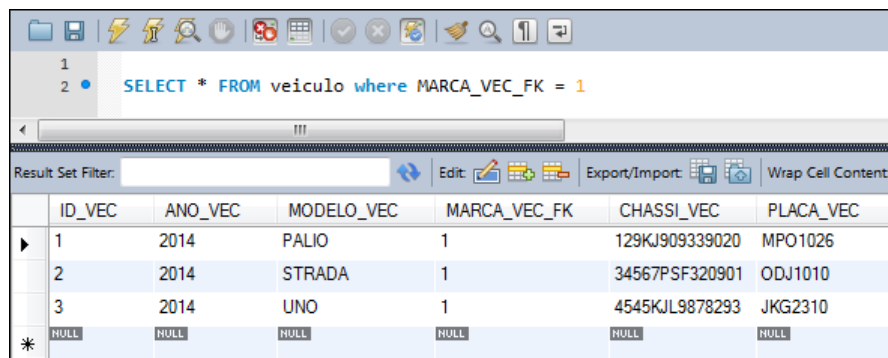
Agora que temos o banco pronto, precisamos inserir dados nele. A Figura 3 mostra o *script* de inserção (*insert.sql*), disponível com este material. Só com curiosidade, o acrônimo CRUD significa, em inglês, *Create*, *Read*, *Update* e *Delete*. O *create* corresponde ao comando *insert* na linguagem SQL.

```
insert into marca (ID_MAR, DESC_MAR) values (1, 'FIAT');
insert into marca (ID_MAR, DESC_MAR) values (2, 'VOLKSWAGEN');
insert into marca (ID_MAR, DESC_MAR) values (3, 'CHEVROLET');
insert into marca (ID_MAR, DESC_MAR) values (4, 'FORD');
insert into marca (ID_MAR, DESC_MAR) values (5, 'TOYOTA');
insert into marca (ID_MAR, DESC_MAR) values (6, 'HONDA');
insert into marca (ID_MAR, DESC_MAR) values (7, 'SUBARU');
insert into marca (ID_MAR, DESC_MAR) values (8, 'KIA');
insert into marca (ID_MAR, DESC_MAR) values (9, 'NISSAN');

insert into veiculo (ID_VEC, ANO_VEC, MODELO_VEC, MARCA_VEC_FK, CHASSI_VEC, PLACA_VEC)
values (1, 2014, 'PALIO', 1, '129KJ909339020', 'MPO1026');
insert into veiculo (ID_VEC, ANO_VEC, MODELO_VEC, MARCA_VEC_FK, CHASSI_VEC, PLACA_VEC)
values (2, 2014, 'STRADA', 1, '34567PSF320901', 'ODJ1010');
insert into veiculo (ID_VEC, ANO_VEC, MODELO_VEC, MARCA_VEC_FK, CHASSI_VEC, PLACA_VEC)
values (3, 2014, 'UNO', 1, '4545KJL9878293', 'JGK2310');
```

Figura 3: Inserindo dados no banco.

Para verificar o resultado da inserção, basta usar o comando *select*, como mostrado na Figura 4. O resultado pode ser observado abaixo, na mesma figura. O *script* de seleção está no arquivo *select.sql*. Na terminologia do CRUD o *select* é representado pelo *read*.



The screenshot shows a database management interface. At the top, a SQL query is entered: `SELECT * FROM veiculo where MARCA_VEC_FK = 1`. Below the query, a table displays the results. The table has six columns: ID_VEC, ANO_VEC, MODELO_VEC, MARCA_VEC_FK, CHASSI_VEC, and PLACA_VEC. There are three data rows, all with MARCA_VEC_FK = 1. A fourth row shows NULL values for all columns. The interface includes a toolbar with various icons and a 'Result Set Filter' section.

	ID_VEC	ANO_VEC	MODELO_VEC	MARCA_VEC_FK	CHASSI_VEC	PLACA_VEC
▶	1	2014	PALIO	1	129KJ909339020	MPO1026
	2	2014	STRADA	1	34567PSF320901	ODJ1010
	3	2014	UNO	1	4545KJL9878293	JGK2310
*	NULL	NULL	NULL	NULL	NULL	NULL

Figura 4: Comando de seleção.

Para alterar um registro devemos usar o comando *update*. Um exemplo de utilização deste comando é mostrado na Figura 5. O *script* do comando *update* está disponível no arquivo *update.sql*.

Para concluir o CRUD ainda falta a exclusão, que na linguagem SQL corresponde ao comando *delete*. Atenção: tenha cuidado com o comando *delete* feito

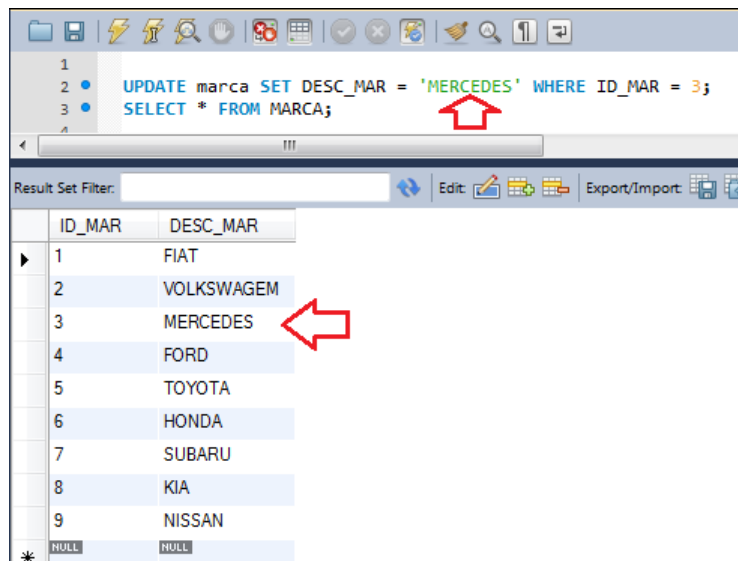


Figura 5: Comando de atualização.

sem a cláusula where, ou seja, delete from TABELA. Esse comando (sem where) limpa a tabela completamente! Na Figura 6 mostramos um comando de exclusão.

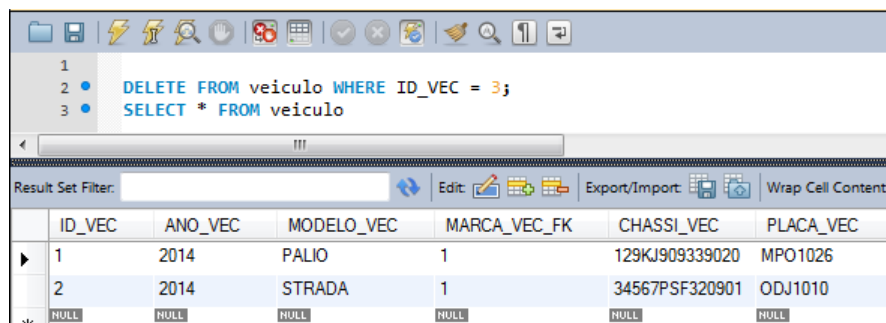


Figura 6: Comando de exclusão.

2 Configurando e conectando Java com MySQL

Agora que temos o banco de dados funcional, precisamos fazer com que este banco seja acessível via Java. O Java usa a tecnologia JDBC (*Java Database Connectivity*) [3]. Para preparar um projeto Java que acessa MySQL, faça:

- Baixar e instalar o conector JDBC [4].

- Obter o arquivo `mysql-connector-java-XYZ-bin.jar` a partir da instalação acima (o local pode variar de acordo com a instalação).
- Criar um projeto Java no Eclipse.
- Criar uma pasta `lib` (ou outro nome) dentro do projeto.
- Copiar o arquivo `mysql-connector-java-XYZ-bin.jar` para dentro da pasta `lib` conforme ilustrado na Figura 7.

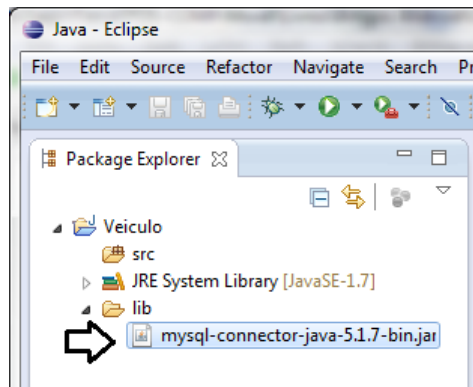


Figura 7: Comando de exclusão.

- Adicione o `.jar` ao projeto. Para isso (1) selecione o projeto, (2) pressione `<ALT + ENTER>`, (3) em Java Build Path, aba Libraries, pressione o botão `Add Jars` (Figura 8) e (4) selecione o arquivo `mysql-connector-java-XYZ-bin.jar` na pasta `lib`.

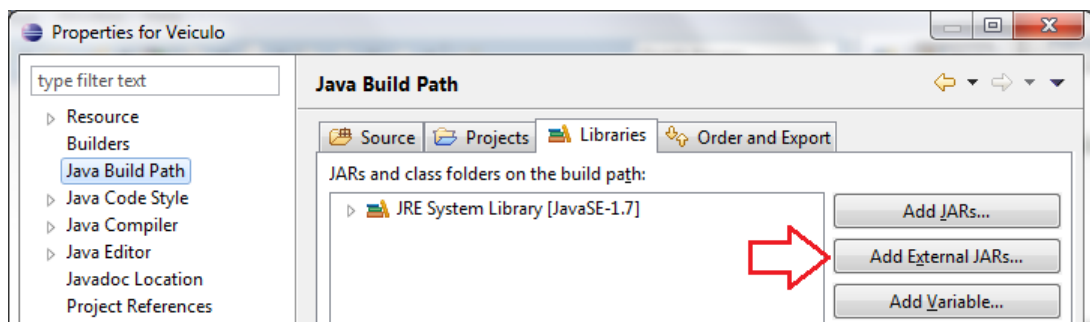


Figura 8: Configuração do JDBC.

O resultado pode ser observado na Figura 9.

Assim que o projeto está configurado para usar Java com o JDBC MySQL, é necessário então conectar o projeto ao banco de dados. A conexão pode ser feita

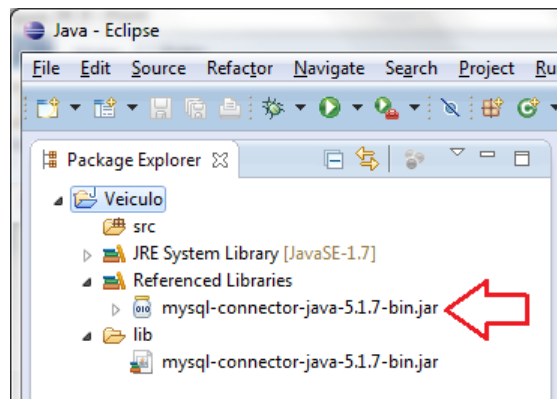


Figura 9: Projeto Eclipse configurado com JDBC MySQL.

através de uma classe de conexão, como mostrado na Figura 10. Observe que no marcador 1 da Figura 10 são iniciados os parâmetros da conexão e no marcador 2 uma conexão é instanciada pela "fábrica" `DriverManager.getConnection(...)`.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Conexao {
    private final static String host = "localhost";
    private final static String alias = "veiculos";
    private final static String url = "jdbc:mysql://" + host + "/" + alias;
    1 private final static String driver = "com.mysql.jdbc.Driver";
    private final static String usu = "root";
    private final static String senha = "123456";
    private static Connection conexao = null;

    public static Connection abreConexao() throws SQLException {
        try {
            2 Class.forName(driver);
            conexao = DriverManager.getConnection(url, usu, senha);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        return conexao;
    }
}
```

Figura 10: Conexão com MySQL via Java.

Para termos um código mais profissional, sem aqueles parâmetros definidos manualmente (*hardcode*), podemos usar o artifício de externalizar texto, disponível no Eclipse. Para isso basta acionar o comando de menu `Source > Externalize`

Strings... e, na janela que surge, marcar os textos que deseja externalizar. Os textos externalizados serão armazenados em um arquivo `messages.properties` (Figura 11) e seu conteúdo será chamado por uma classe de apoio `Messages`. A Figura 12 mostra como fica o código de definição de parâmetros do banco.

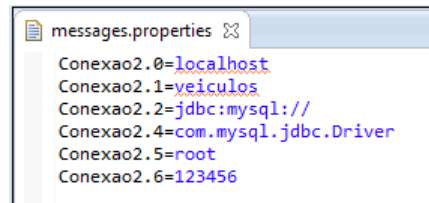


Figura 11: Arquivo `messages.properties`.

```
private final static String host = Messages.getString("Conexao2.0");
private final static String alias = Messages.getString("Conexao2.1");
private final static String url = Messages.getString("Conexao2.2")
    + host + "/" + alias;
private final static String driver = Messages.getString("Conexao2.4");
private final static String usu = Messages.getString("Conexao2.5");
private final static String senha = Messages.getString("Conexao2.6");
private static Connection conexao = null;
```

Figura 12: Externalizando as *strings* da conexão.

Agora podemos testar a classe de conexão adicionando à classe `Conexao` um método `main` com o conteúdo mostrado na Figura 13. Os métodos `getCatalog()` e `getClass()` retornam, respectivamente, o nome do banco e o nome da classe do conector. Faça o teste!

```
public static void main(String[] args) {
    System.out.println("Teste da Conexão:");
    try {
        System.out.println("Nome: " + (Conexao.abreConexao()).getCatalog());
        System.out.println("Classe: " + (Conexao.abreConexao()).getClass());
    } catch (SQLException e) {
        System.out.println("Mensagem: " + e.getMessage());
        System.out.println("Erro no: " + e.getErrorCode());
    }
}
```

Figura 13: Implementação do teste de conexão.

3 Camada de persistência em Java

Em um sistema bem projetado, a camada de persistência deve fazer a comunicação entre o sistema e a base de dados de forma que o código do sistema fique pouco (ou nada) dependente do banco de dados. Como consequência podemos trocar o banco de dados de MySQL para Oracle, por exemplo, sem que hajam alterações nas camadas do MVC. Esse é o conceito de desacoplamento, que é ilustrado na Figura 14. Repare que utilizamos dois padrões: o MVC e o DAO (do inglês, *Data Access Object*).

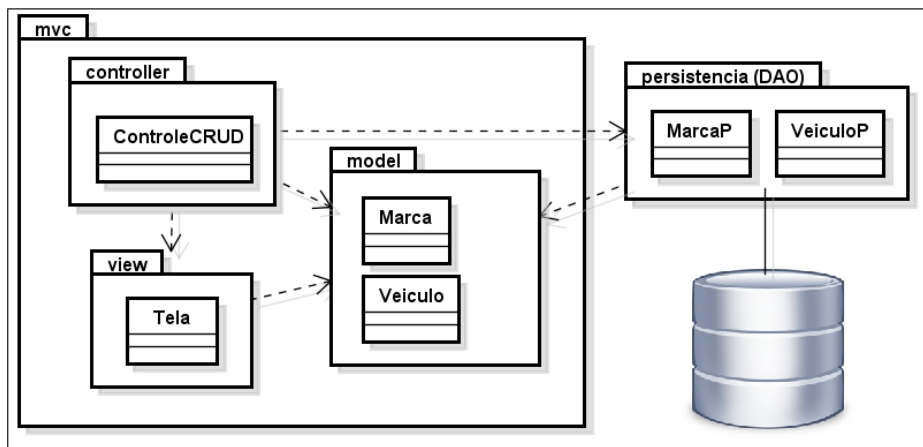


Figura 14: Esquema de um sistema com persistência.

Na nossa camada de persistência é usual termos uma classe de persistência para cada classe do modelo. Assim, para a classe *Marca* (do modelo), temos uma classe de persistência *MarcaP*, ou *MarcaDao*, se você preferir. Vamos projetar a classe de persistência *MarcaP*. A seguir faremos um *checklist* do que devemos ter na classe de persistência:

- Um atributo para a conexão.
- Operação "recuperar todos"(`getLista()`).
- Operação "recuperar por id"(`getById(id)`).
- Operação "salvar"(`insert(obj)`).
- Operação "alterar"(`update(obj)`).
- Operação "excluir por id"(`delete(id)`).

É importante ressaltar que essas operações são básicas e comuns a todas as classes de persistência que precisaremos implementar. Além dessas operações podemos definir outras mais específicas, como "recuperar elemento por sua descrição", ou "excluir elemento por sua descrição", e etc. Para as operações que

são comuns a várias persistências, podemos criar uma interface genérica, que vai garantir que toda classe de persistência que a implementar terá os métodos básicos necessários. Chamaremos essa interface de Persistencia, e uma possível implementação é mostrada na Figura 15.

```
public interface Persistencia<T> {  
    List<T> getList() throws SQLException;  
    T getById(Long id) throws SQLException;  
    void insert(T t) throws SQLException;  
    void update(T t) throws SQLException;  
    void deleteById(Long id) throws SQLException;  
    void delete() throws SQLException;  
}
```

Figura 15: Projeto da persistência.

A classe de persistência MarcaP deve implementar a interface Persistencia<T> e, dessa forma, realizar o contrato da interface (Figura 16).

```
public class MarcaP implements Persistencia<Marca>{  
    private Connection conexao;  
    public MarcaP() throws SQLException{  
        conexao = Conexao.abreConexao();  
    }  
    public MarcaP(Connection conexao) {  
        this.conexao = conexao;  
    }  
    @Override  
    public List<Marca> getList() throws SQLException {  
        return null;  
    }  
    @Override  
    public Marca getById(Long id) throws SQLException {  
        return null;  
    }  
    @Override  
    public void insert(Marca t) throws SQLException { }  
    @Override  
    public void update(Marca t) throws SQLException { }  
    @Override  
    public void deleteById(Long id) throws SQLException { }  
    @Override  
    public void delete() throws SQLException { }  
}
```

Figura 16: Projeto da persistência.

Implementaremos a seguir os métodos da classe MarcaP. O primeiro é o método getList(), Figura 17. No marcador 1 (Figura 17) observamos a criação da *query* que seleciona todos os elementos da tabela. A sua execução, que retorna um ResultSet, ocorre logo depois (marcador 2). Ainda na Figura 17, no marcador 3,

ocorre a caminhada pelo `ResultSet rs` e carga da lista de elementos (`result`), que é retornada pelo método.

```
@Override
public List<Marca> getList() throws SQLException {
    PreparedStatement pstmt = conexao.prepareStatement("select *
                                                         from marca");
    2 ➡ ResultSet rs = pstmt.executeQuery();
    List <Marca> result = new ArrayList <Marca>();
    while (rs.next()){
        3 ➡ Marca marca = new Marca();
        marca.setId(rs.getLong("ID_MAR"));
        marca.setDesc(rs.getString("DESC_MAR"));
        result.add(marca);
    }
    return result;
}
```

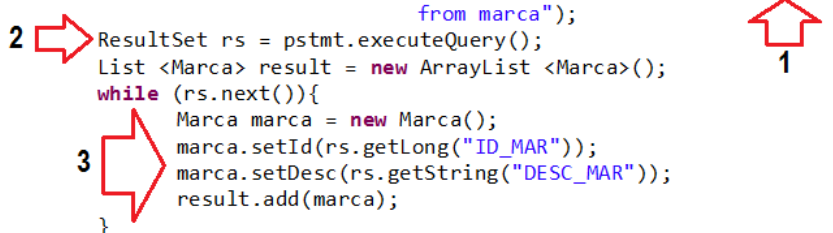


Figura 17: Recupera todos os registros.

O método seguinte, `getById(id)`, recebe um identificador de objeto e monta a *query* com uma cláusula *where* parametrizada com um sinal de interrogação (?), indicado pelo marcador 2 da Figura 18. O parâmetro que substituirá a o sinal de interrogação (?) durante a execução da *query* é passado no marcador 2 da Figura 18. No marcador 3 da mesma figura, se o `ResultSet rs` possui algum valor, então um objeto é carregado e retornado pelo método.

```
@Override
public Marca getById(Long id) throws SQLException{
    PreparedStatement pstmt = conexao.prepareStatement("select *
                                                         from marca where ID_MAR = ?");
    2 ➡ pstmt.setLong(1, id);
    ResultSet rs = pstmt.executeQuery();
    Marca marca = new Marca();
    3 ➡ if (rs.next()){
        marca.setId(rs.getLong("ID_MAR"));
        marca.setDesc(rs.getString("DESC_MAR"));
    }
    return marca;
}
```

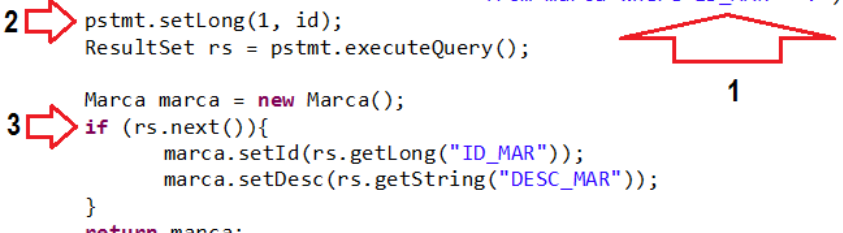


Figura 18: Recupera registros por id.

O método `insert(obj)` (Figura 19) adiciona um novo objeto `obj` ao banco relacional. Observe que o comando `insert` recebe dois parâmetros representados por (?, ?), como destacado pelo marcador 1. Os parâmetros são passados conforme indicado no marcador 2 da Figura 19: o primeiro é o identificador e o segundo é a descrição.

```

public void insert(Marca marca) throws SQLException{
    PreparedStatement pstmt = conexao.prepareStatement("insert into
                                                         marca values (?, ?)");

    2 ➡ pstmt.setInt(1, 0);
        pstmt.setString(2, marca.getDesc());
        pstmt.execute();
        pstmt.close();
}

```

1

Figura 19: Adiciona objeto.

O método `update(obj)` tem a mesma estrutura geral do método `insert(obj)`, diferindo apenas na cláusula SQL, como destacado pelo marcador 1 da Figura 20.

```

public void update(Marca marca) throws SQLException{
    PreparedStatement pstmt = conexao.prepareStatement("update marca
                                                         set DESC_MAR = ? where ID_MAR = ?");

    pstmt.setString(1, marca.getDesc());
    pstmt.setLong(2, marca.getId());
    pstmt.execute();
    pstmt.close();
}

```

1

Figura 20: Atualiza objeto.

O método `deleteById(id)` fica como na Figura 21.

```

public void deleteById(Long id) throws SQLException {
    PreparedStatement pstmt = conexao.prepareStatement("delete from
                                                         marca where ID_MAR = ?" );

    pstmt.setLong(1, id);
    pstmt.execute();
    pstmt.close();
}

```

Figura 21: Exclui objeto por identificador.

Como já mencionado nesta seção, muitos outros métodos podem ser implementados para a classe `MarcaP`. Entretanto, se o fizermos correremos o risco de tornar o texto muito repetitivo.

A classe `VeiculoP` segue, em linhas gerais, a mesma lógica da classe `MarcaP`, pois ambas devem implementar a interface `Persistencia`. Entretanto, há uma diferença que deve ser notada. Observe, na Figura 1 (modelo relacional) que a `marca` é apenas chave estrangeira (`MARCA_VEC_FK`) na tabela `veiculo`, mas no modelo OO (Figura 22) `marca` é um atributo da classe `Veiculo`.

Por isso, ao escrever os métodos da persistência de `VeiculoP`, devemos nos

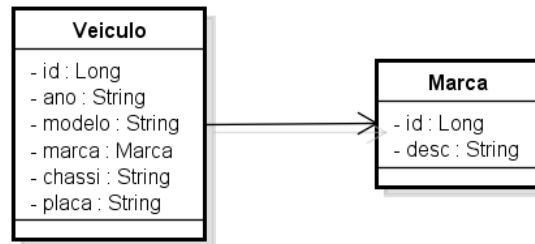


Figura 22: Modelo OO de Veiculo e Marca.

preocupar com o mapeamento objeto x relacional, conforme destacado no marcador 1 da Figura 23.

```

public List<Veiculo> getList() throws SQLException {
    PreparedStatement pstmt = conexao.prepareStatement("select *
                                                         from veiculo");

    ResultSet rs = pstmt.executeQuery();
    List <Veiculo> result = new ArrayList <Veiculo>();
    while (rs.next()){
        Veiculo veiculo = new Veiculo();
        veiculo.setId(rs.getLong("ID_VEC"));
        veiculo.setAno(rs.getString("ANO_VEC"));
        veiculo.setModelo(rs.getString("MODELO_VEC"));
        //---- OBTENDO O OBJETO MARCA
        Marca m = (new MarcaP()).getId(rs.getLong("MARCA_VEC_FK")); ← 1
        veiculo.setMarca(m);
        veiculo.setChassi(rs.getString("CHASSI_VEC"));
        veiculo.setPlaca(rs.getString("PLACA_VEC"));
        result.add(veiculo);
    }
    return result;
}
  
```

Figura 23: Codificando o mapeamento objeto relacional entre Veiculo e Marca.

A mesma ideia se aplica a todos os métodos de consulta deste modelo.

Para concluir este texto, gostaria de apresentar dois testes da camada de persistência. O primeiro é o teste do método `getList()` da classe `MarcaP` e o segundo é o teste do mesmo método, mas da classe `VeiculoP`. O código Java é mostrado na Figura 24. Execute este teste e veja o resultado.

```

public class Teste {
    public static void main(String[] args) {
        try {
            List <Marca> listaMar = (new MarcaP()).getList();
            System.out.println(listaMar);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        System.out.println("-----");
        try {
            List <Veiculo> listaVec = (new VeiculoP()).getList();
            System.out.println(listaVec);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

Figura 24: Teste de MarcaP.

Referências

- [1] SQL, Wikipedia. <<http://pt.wikipedia.org/wiki/SQL>>. Acessado em: 5 agosto 2014.
- [2] Chapter 4 Getting Started Tutorial, Documentation, MySQL. <<http://dev.mysql.com/doc/workbench/en/wb-getting-started-tutorial.html>>. Acessado em: 5 agosto 2014.
- [3] Lesson: JDBC Basics, The Java Tutorial. <<http://docs.oracle.com/javase/tutorial/jdbc/basics/>>. Acessado em: 6 agosto 2014.
- [4] Download Connector/J, Downloads, MySQL. <<http://dev.mysql.com/downloads/connector/j/>>. Acessado em: 6 agosto 2014.