



Desenvolvimento para Servidores-II

Fundamentos de JDBC

Neste tópico abordaremos os fundamentos de JDBC (Java *Database Connectivity*)

Prof. Ciro Cirne Trindade

O que é JDBC?

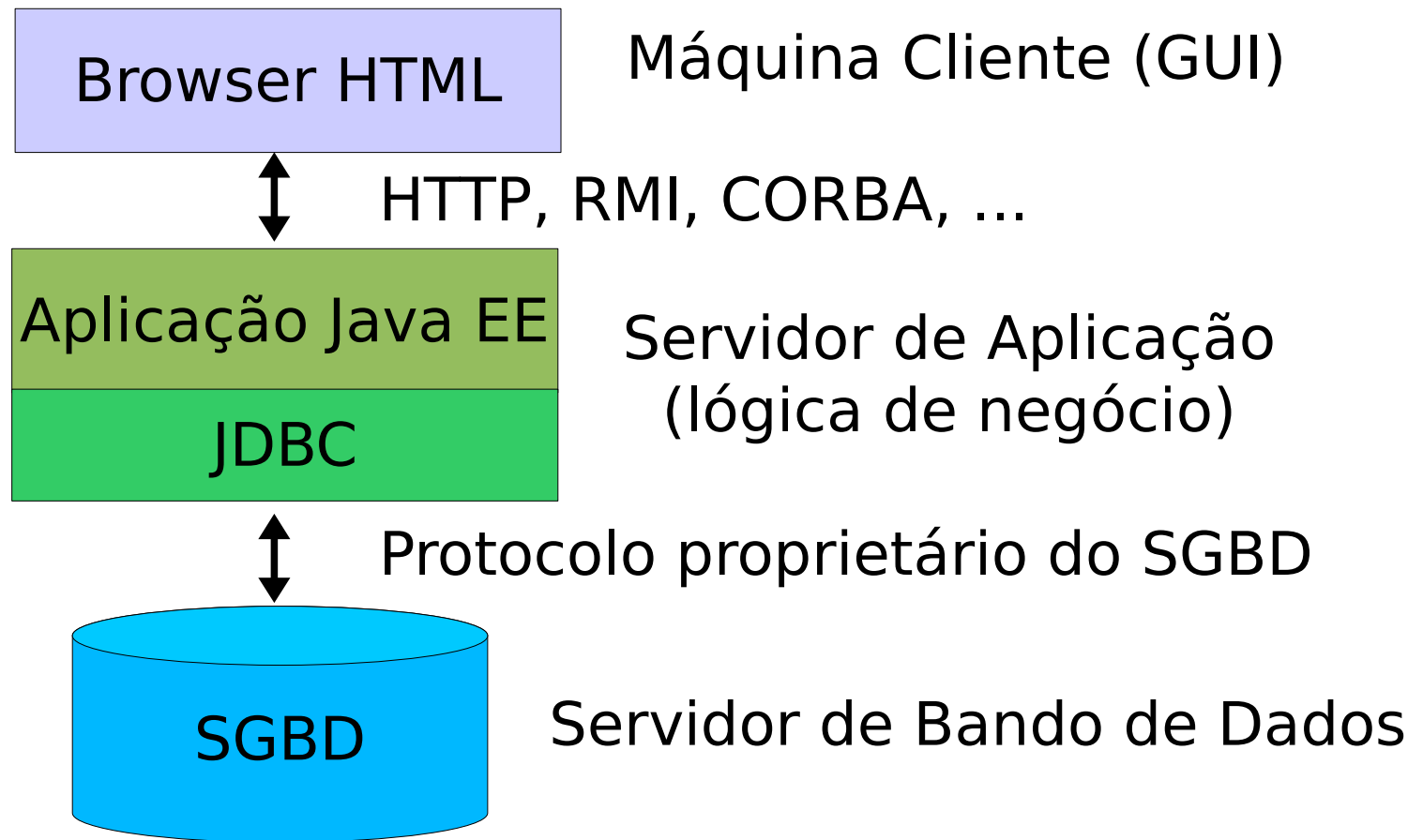
- É uma API padrão para manipular dados localizados em banco de dados, preferencialmente relacionais
 - Incorpora conexões com banco de dados
 - Indpende da plataforma utilizada para persistir dados
 - Suporta Java RMI
 - Suporta ODBC

ODBC + SQL
permitem conexões
e acessos
padronizados a
bancos de dados
relacionais

JDBC API

- Permite 3 coisas:
 - Estabelecer uma conexão com um banco de dados
 - Enviar comandos SQL
 - Processar os resultados

Arquitetura em 3 camadas para acesso a dados



JDBC API Básica

- **Connection**: interface que representa uma conexão com o banco de dados
- **DriverManager**: classe que mantém uma lista de classes **Driver**, o método **getConnection()** devolve o driver apropriado
- **Statement**: interface usada para enviar comandos SQL sem parâmetros
- **PreparedStatement**: interface que representa comandos SQL pré compilados, podem recebe um ou mais parâmetros
- **ResultSet**: interface que contém o resultado da execução de um comando SQL

getConnection()

- `public static Connection
getConnection(String url,
String user, String password)
throws SQLException`
 - `url`: URL do banco de dados na forma
jdbc:subprotocolo://endereço-IP-
banco[:porta]/nome-do-banco
 - Exemplo: jdbc:mysql://localhost/dps
 - `user`: usuário do banco em nome de
quem a conexão será estabelecida
 - `password`: senha do usuário

Driver JDBC para MySQL

- Driver JDBC para a versão 5.1 do MySQL
 - mysql-connector-java-5.1.35-bin.jar
- Adicionar este jar às Bibliotecas do seu projeto no NetBeans

Class.forName?

- Até a versão 3 do JDBC, antes de chamar o `DriverManager.getConnection()` era necessário registrar o driver JDBC que iria ser utilizado através do método `Class.forName("com.mysql.jdbc.Driver")`, no caso do MySQL, que carregava essa classe, e essa se comunicava com o `DriverManager`
- A partir do JDBC 4, que está presente a partir do Java 6, esse passo não é mais necessário



Criando um banco de dados no MySQL (1/3)

- No terminal

- `$ mysql -u root -p`

- `$ Enter password:`

fatec

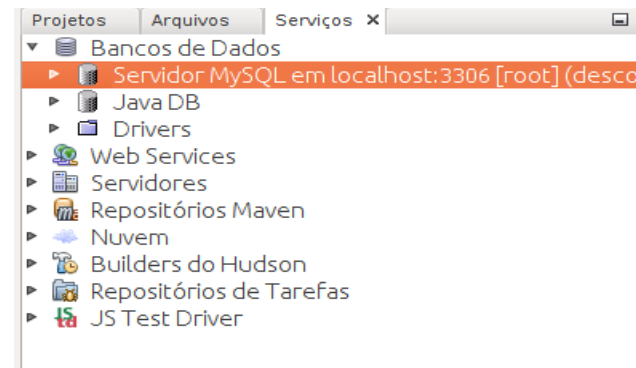
- `mysql> create database dps;`

Criando um banco de dados no MySQL (2/3)

- No NetBeans (1/3)
 - O NetBeans já vem com suporte nativo para o MySQL
 - Clique na aba Serviços
 - Se o Servidor do MySQL não estiver registrado, dê um clique direito em Banco de Dados e selecione a opção “Registrar Servidor MySQL”

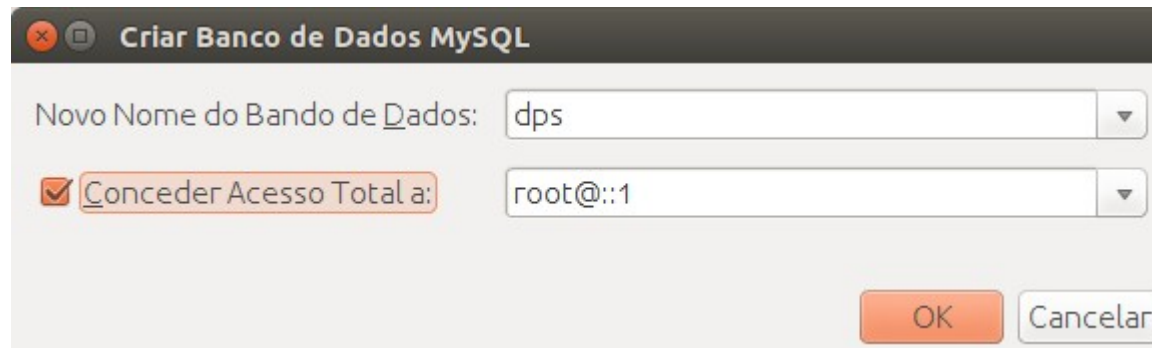
Criando um banco de dados no MySQL (2/3)

- No NetBeans (2/3)
 - Com o servidor MySQL registrado, dê um clique com o botão direito do mouse sobre o Servidor MySQL e selecione a opção Conectar



Criando um banco de dados no MySQL (3/3)

- No NetBeans (3/3)
 - Dê um novo clique com o botão direito do mouse no Servidor MySQL e selecione a opção Criar Banco de Dados



Design Patterns

- Um *design pattern* é uma solução eficiente para um problema recorrente
- Um *pattern* provê um conjunto de interações específicas que podem ser aplicadas a objetos genéricos para resolver um problema conhecido

Desing pattern Factory

- Prega o encapsulamento da construção (fabricação) de objetos
- Vamos implementar um classe `ConnectionFactory` que implementa o *desing pattern* Factory



ConnectionFactory

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

public class ConnectionFactory {
    private static final String DATABASE_URL =
        "jdbc:mysql://localhost/dps";
    private static final String DATABASE_USER = "root";
    private static final String DATABASE_PASSWORD = "fatec";

    public Connection getConnection() throws SQLException {
        try {
            return DriverManager.getConnection(
                DATABASE_URL, DATABASE_USER, DATABASE_PASSWORD);
        } catch (SQLException e) {
            throw e;
        }
    }
}
```

Testando a conexão

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

public class TestaConexao {

    public static void main(String[] args) {
        ConnectionFactory cf = new ConnectionFactory();
        try {
            Connection con = cf.getConnection();
            System.out.println("Conexão aberta!");
            con.close();
        } catch (SQLException e) {
            System.out.println("Falha na conexão: " +
                               e.getMessage());
        }
    }
}
```



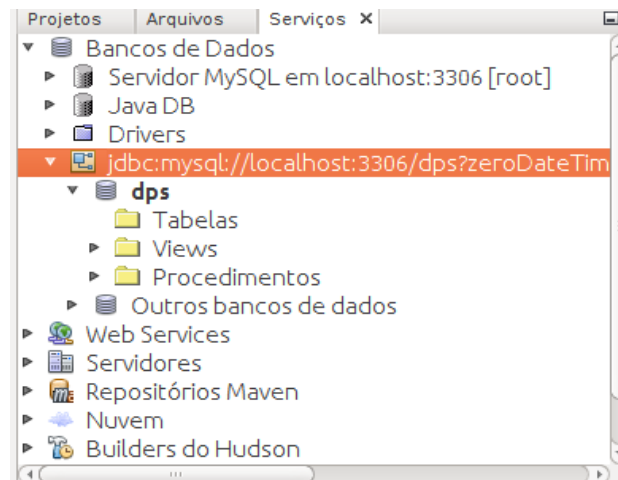

Criando uma tabela de contatos no MySQL (1/3)

■ No terminal

- `$ mysql -u root -p`
- `$ Enter password:`
- `mysql> use dps;`
- `mysql> create table contatos (
id BIGINT NOT NULL AUTO_INCREMENT,
nome VARCHAR(80),
email VARCHAR(120),
endereco VARCHAR(255),
primary key (id)
);`

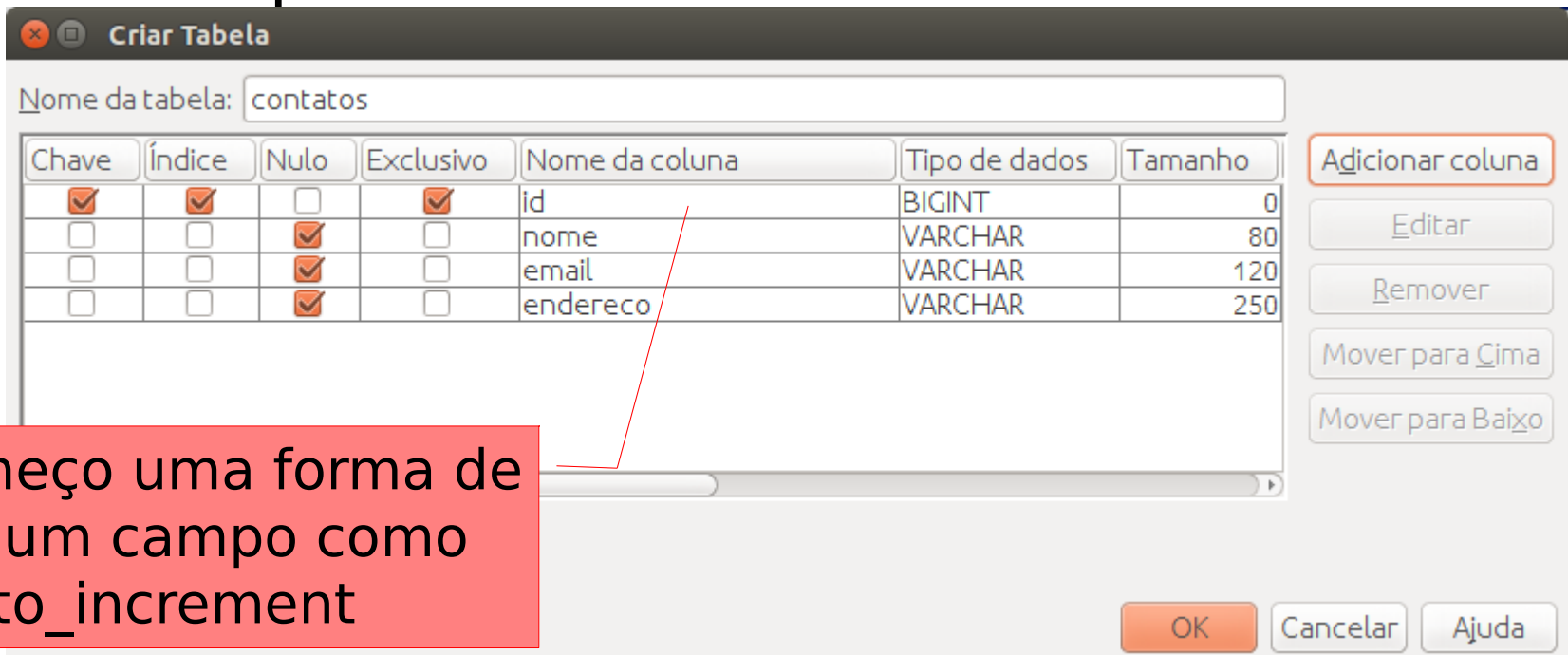
Criando uma tabela de contatos no MySQL (2/3)

- No NetBeans (1/2)
 - Dê um clique direito do mouse na pasta Tabelas do bando de dados dps
 - Selecione a opção Criar tabela



Criando uma tabela de contatos no MySQL (3/3)

- No NetBeans (2/2)
 - Forneça o nome da tabela e clique no botão Adicionar coluna para criar os campos da tabela



Nome da tabela: contatos

Chave	Índice	Nulo	Exclusivo	Nome da coluna	Tipo de dados	Tamanho
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	id	BIGINT	0
<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	nome	VARCHAR	80
<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	email	VARCHAR	120
<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	endereco	VARCHAR	250

Adicionar coluna

Editar

Remover

Mover para Cima

Mover para Baixo

OK Cancelar Ajuda

Não conheço uma forma de definir um campo como auto_increment

Inserindo dados no banco

- Para inserir dados em uma tabela de um banco de dados basta usar a cláusula `INSERT`
- Precisamos especificar quais os campos que desejamos atualizar e os valores
 - ```
String sql = "insert into contatos
 (nome, email, endereco) values ('" +
 nome + "', '" + email + "', '" +
 endereco + "')";
```

# Inserindo dados no banco

- O código anterior possui três problemas:
  - Legibilidade: difícil saber se faltou uma vírgula ou fechar um parênteses
  - SQL *injection*: o usuário final é capaz de alterar seu código SQL para executar aquilo que ele deseja
  - Formato dos dados: é preciso passar as strings entre apóstrofo

# Inserindo dados no banco

- Uma forma mais simples:
  - `String sql = "insert into contatos (nome, email, endereco) values (?, ?, ?)";`
  - Os pontos de interrogação (?) são os parâmetros que iremos utilizar nesse código SQL que será executado

# PreparedStatement (1/3)

- As cláusulas são executadas em um banco de dados através da interface `PreparedStatement`
- Para criar um `PreparedStatement` relativo à conexão, basta chamar o método `prepareStatement` da interface `Connection`, passando como argumento o comando SQL
  - `PreparedStatement stmt = connection.prepareStatement(sql);`

# PreparedStatement (2/3)

- Logo em seguida, chamamos o método `setString()` do `PreparedStatement` para preencher os valores que são do tipo `String`, passando a posição (começando em 1) da interrogação no SQL e o valor que deve ser colocado:
  - `stmt.setString(1, "Ciro Trindade");`
  - `stmt.setString(2, "ciroct@gmail.com");`
  - `stmt.setString(3, "R. Santos Dumont, 162");`



# PreparedStatement (3/3)

- Por fim, uma chamada a `execute()` executa o comando SQL:
  - `stmt.execute();`



# Exemplo: classe

## JDBCInserere (1/2)

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

public class JDBCInserere {
 // recebe os valores via argumentos do main()
 public static void main(String[] args) throws
 SQLException{
 if (args.length == 3) {
 // conectando
 Connection con = new
 ConnectionFactory().getConnection();
 // cria um preparedStatement
 String sql = "insert into contatos
 (nome,email,endereco) values (?, ?, ?)";
 PreparedStatement stmt =
 con.prepareStatement(sql);
```

# Exemplo: classe

## JDBCInserer (2/2)

```
// preenche os valores
stmt.setString(1, args[0]);
stmt.setString(2, args[1]);
stmt.setString(3, args[2]);

// executa
stmt.execute();
stmt.close();
System.out.println("Gravado!");
con.close();
}
else {
 System.out.println("Forneça os valores via
argumentos do main");
}
}
}
```

# ResultSet (1/2)

- A interface `ResultSet` é usada para receber o resultado de um `SELECT`
- O `ResultSet` é uma lista aonde cada item representa um registro devolvido pelo `SELECT`
- Para executar um `SELECT` utiliza-se o método `executeQuery()` da interface `Statement` OU `PreparedStatement`

# ResultSet (2/2)

- Para iterar sobre essa lista usa-se o método `next ( )`
  - O método `next ( )` devolve `true` se ainda houver itens no `ResultSet` ou `false`, caso contrário
- Para recuperar os campos de cada item do `ResultSet` usa-se métodos *getters*



# Exemplo: classe

## JBCListar (1/2)

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class JDBCListar {
 public static void main(String[] args) throws
 SQLException{
 // conectando
 Connection con = new
 ConnectionFactory().getConnection();
 // cria um preparedStatement
 String sql =
 "select * from contatos order by nome";
 PreparedStatement stmt =
 con.prepareStatement(sql);
```

# Exemplo: classe

## JCBCListar (2/2)

```
// executa a consulta
ResultSet rs = stmt.executeQuery();

while(rs.next()) { // itera sobre o ResultSet
 // recupera os campos de cada registro
 Long id = rs.getLong("id");
 String nome = rs.getString("nome");
 String email = rs.getString("email");
 String endereco = rs.getString("endereco");

 System.out.println(id + " - " + nome + " - " +
e-mail + " - " + endereco);
}
stmt.close();
con.close();
}
```

# JavaBeans

- Em uma aplicação Java criamos um JavaBean para representar uma tabela em um banco de dados
- Um JavaBean é uma classe em que:
  - Todos os atributos são privados
  - Possui um construtor *default*
  - Possui os métodos de acesso (*getters* e *setters*) públicos



# JavaBeans Contato (1/2)

```
public class Contato {
 private Long id;
 private String nome;
 private String email;
 private String endereco;

 public Contato() { }

 public Long getId() {
 return id;
 }
 public void setId(Long id) {
 this.id = id;
 }
 public String getNome() {
 return nome;
 }
}
```

# JavaBeans Contato (2/2)

```
public void setNome(String nome) {
 this.nome = nome;
}
public String getEmail() {
 return email;
}
public void setEmail(String email) {
 this.email = email;
}
public String getEndereco() {
 return endereco;
}
public void setEndereco(String endereco) {
 this.endereco = endereco;
}
} // fim da classe Contato
```

# *Design pattern DAO (Data Access Object)*

- O *pattern Data Access Object* separa o código que acessa o banco de dados do código que trabalha com os dados
- Outros componentes da aplicação delegam a responsabilidade de acesso aos dados a um objeto DAO, que se comunica com o resto do sistema passando, normalmente, JavaBeans ou coleções de JavaBeans

# Exemplo: classe de acesso à tabela contato (ContatoDAO) (1/3)

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.List;
import java.util.ArrayList;

public class ContatoDAO {
 private static final String SQL_INSERTIR_CONTATO =
 "insert into contatos (nome, email, endereco) values
 (?, ?, ?) ";
 private static final String SQL_LISTAR_CONTATOS =
 "select * contatos order by nome";

 // a conexão com o banco de dados
 private Connection connection;
```

# Exemplo: classe de acesso à tabela contato (ContatoDAO) (2/3)

```
public void adicionar(Contato contato) throws SQLException
{
 try {
 connection = new
 ConnectionFactory().getConnection();
 try {
 PreparedStatement stmt = connection.
 prepareStatement(SQL_INSERTIR_CONTATO);
 // seta os valores
 stmt.setString(1, contato.getNome());
 stmt.setString(2, contato.getEmail());
 stmt.setString(3, contato.getEndereco());
 // executa
 stmt.execute();
 stmt.close();
 } finally {
 connection.close();
 }
 } catch (SQLException e) {
 throw e;
 }
}
```

# Exemplo: classe de acesso à tabela contato (ContatoDAO) (3/3)

```
public List<Contato> listar() throws SQLException {
 List<Contato> contatos = new ArrayList<Contato>();
 try {
 connection = new ConnectionFactory().getConnection();
 try {
 PreparedStatement stmt = connection.
 prepareStatement(SQL_LISTAR_CONTATOS);
 ResultSet rs = stmt.executeQuery();
 while(rs.next()) {
 Contato c = new Contato();
 c.setId(rs.getLong("id"));
 c.setNome(rs.getString("nome"));
 c.setEmail(rs.getString("email"));
 c.setEndereco(rs.getString("endereco"));
 contatos.add(c);
 }
 stmt.close();
 rs.close();
 } finally {
 connection.close();
 }
 } catch (SQLException e) { throw e; }
 return contatos;
}
```

# Classe TestaDAO (1/2)

```
import java.sql.SQLException;
import java.util.Scanner;

public class TestaDAO {
 public static void main(String[] args) {
 int op;
 Scanner in = new Scanner(System.in);
 do {
 System.out.println("CONTATOS");
 System.out.println("<1> Adicionar contato");
 System.out.println("<2> Listar contatos");
 System.out.println("<0> Sair do programa");
 System.out.print("Opção: ");
 switch(op = in.nextInt()) {
 case 1:
 Contato contato = new Contato();
 in.nextLine();
 System.out.print("Nome: ");
 contato.setNome(in.nextLine());
 System.out.print("E-mail: ");
 contato.setEmail(in.nextLine());
 System.out.print("Endereço: ");
 contato.setEndereco(in.nextLine());
```

# Classe TestaDAO (2/2)

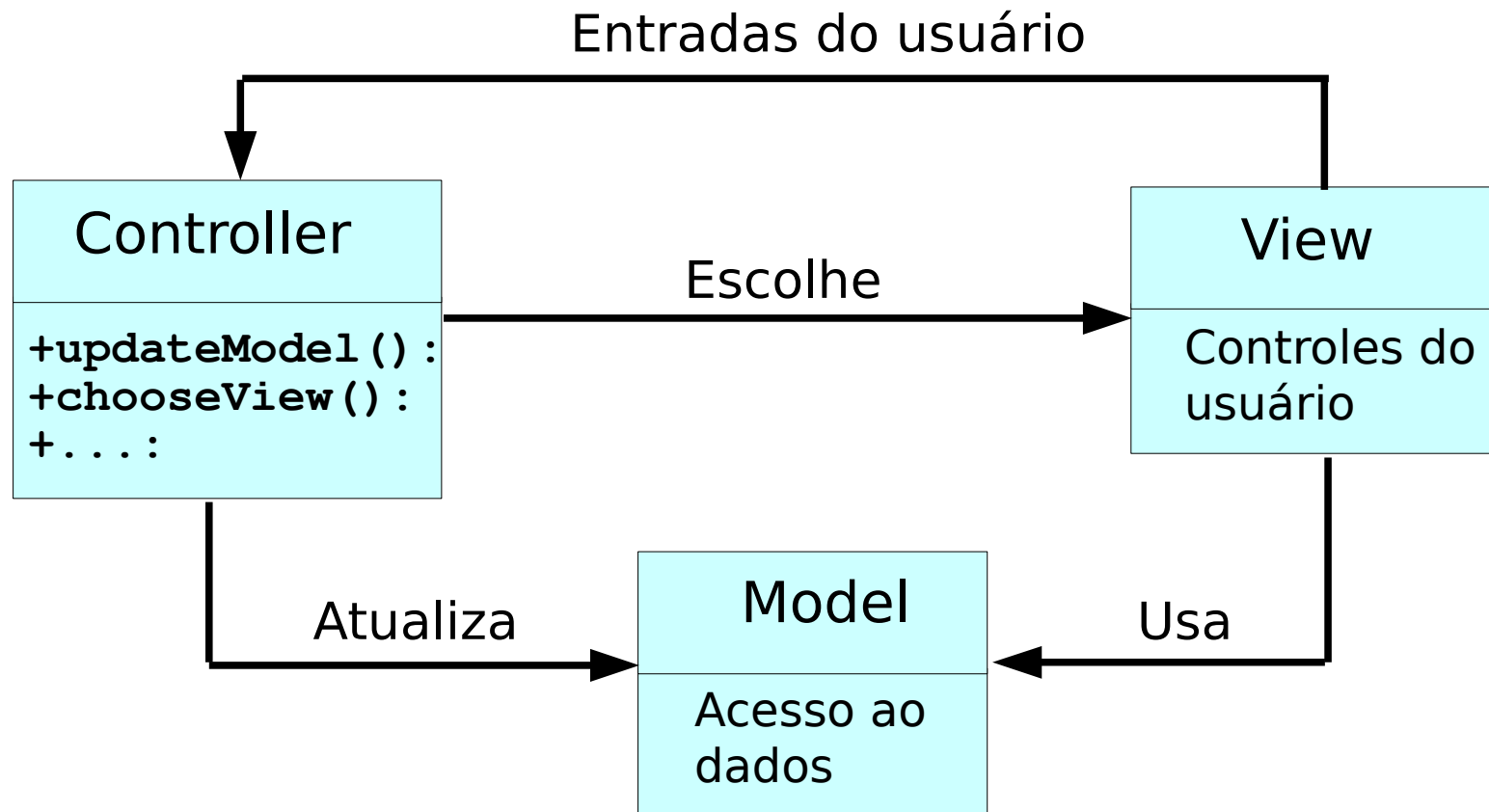
```
ContatoDAO cdao = new ContatoDAO();
try {
 cdao.adicionar(contato);
} catch (SQLException e) {
 System.out.println("Erro: " + e);
}
break;
case 2:
 System.out.println("Contatos Cadastrados");
 ContatoDAO cdao = new ContatoDAO();
 try {
 for (Contato c : cdao.listar()) {
 System.out.println(c.getId() + " - " +
 c.getNome() + " - " + c.getEmail() +
 " - " + c.getEndereco());
 }
 } catch (SQLException e) {
 System.out.println("Erro: " + e);
 }
 break;
default: System.out.println("Opção inválida!");
}
} while (op != 0);
}
```



# *Desing pattern MVC (Model-View-Controller)*

- Quebra a aplicação e 3 partes
  - *Model* (modelo): armazena o estado da aplicação, responsável pelo acesso aos dados
  - *View* (visão): interpreta os dados na camada *model* e os apresenta ao usuário
  - *Controller* (controle): processa as entradas do usuário, atualiza o modelo (*model*) e exibe uma nova visão (*view*)

# Visão geral do *pattern* MVC



# MVC no Java EE

- *Model*: na forma de JavaBeans e DAOs, provê acesso aos dados na camada de negócios
- *View*: páginas JSF
- *Controller*: FacesServlet e *managed beans*



# Configurando o GlassFish para utilizar a API JDBC

- O driver JDBC do banco de dados deve estar na pasta `glassfish\lib` do GlassFish

# Exercício

- Implemente uma aplicação web que permita realizar as seguintes operações em uma agenda de contatos:
  - Inserir um novo contato
  - Listar todos os contatos cadastrados
  - Consultar os dados de um contato dado seu nome
  - Excluir um contato dado seu `id`

# Referências

- ORACLE. *JDBC Basics*.  
<https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html>.
- CRAWFORD, W.; KAPLAN, J. *J2EE Design Patterns*, O'Reilly, 2003.
- CAELUM. *J-21: Java para desenvolvimento web*.  
<http://www.caelum.com.br/curso/fj-21-java-web/>