Programação Orientada a Objetos
Unidade 6 – Conectividade com Banco de Dados

Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP

Introdução

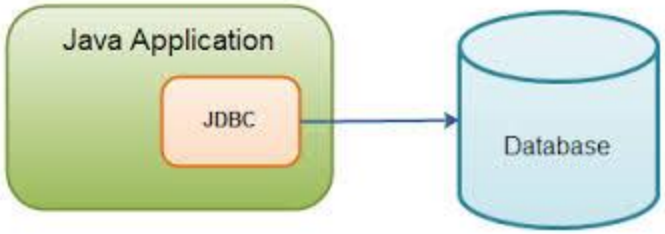
- ✓ A SQL Embutida às vezes é chamada de **Técnica de Programação de Banco de Dados Estática**, pois o texto da consulta é escrito no código fonte do programa e **não** pode ser alterado sem uma nova compilação ou reprocessamento do código fonte;

Programação SQL - CLI

- ✓ Nesta técnica de programação **SQL**, disponibiliza-se ao programa **SQL** uma biblioteca de funções (também conhecida por interface de programação de aplicação (**API**), para acesso ao sistema gerenciador de banco de dados;

- ✓ Nesta técnica, a partir do programa SQL efetuam-se chamadas de função de biblioteca, e por essa razão a técnica é chamada **SQL/CLI – CALL LEVEL INTERFACE**;

- ✓ Um exemplo deste tipo de programação SQL é **JDBC – Java DataBase Connectivity**, uma biblioteca de funções (driver) para acessar banco de dados com Java. Outro exemplo é **ODBC – Open DataBase Connectivity**.



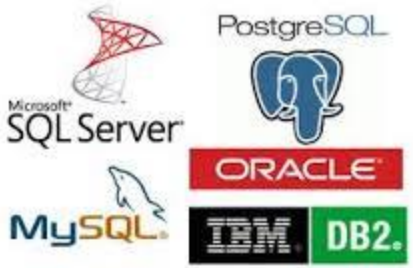
JDBC
Java Database Connectivity

Quais as vantagens de se usar CLI ao invés de SQL embutido ?

Programação SQL CLI - Vantagens

- ✓ A principal vantagem do uso de uma interface de chamada de função (**CLI**) é que ela facilita o acesso à múltiplos bancos de dados no mesmo programa de aplicação, mesmo que eles estejam armazenados em **diferentes sistemas de gerenciamento de banco de dados**;

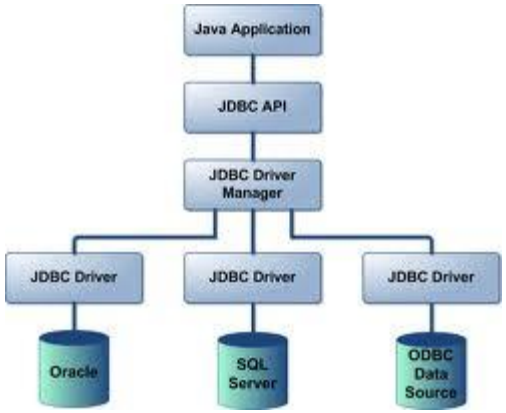
- ✓ Ao se empregar SQL CLI, não há necessidade de se empregar pré-processadores para o processamento do código SQL. No entanto, a **sintaxe** e outras verificações dos comandos SQL precisam ser feitas **em tempo de execução**.



JDBC

JDBC Java Database Connectivity

- ✓ API for Java que define a forma pela qual um programa acessa um banco de dados;
- ✓ Primeira versão do **JDBC – Java Database Connectivity** liberada pela Sun em 1996;
- ✓ Este liberação permitiu que programadores Java pudessem fazer conexão a um banco de dados, atualização e consultas através da linguagem **SQL**;
- ✓ Baseou-se na abordagem da Microsoft para a sua **API ODBC**;
- ✓ Características: **Portabilidade**, API **independente** do SGBD subjacente, Estrutura em **camadas**.



Conectividade JDBC

- ✓ Programas desenvolvidos com Java e JDBC são independentes de plataforma e de fornecedores de SGBD.
- ✓ O mesmo programa Java pode rodar em um PC, uma workstation, etc.
- ✓ Pode-se mover dados de um **SGBD** para outro (por exemplo, **SQL Server** para **DB/2**).

Padrão JDBC de acesso a Bases de Dados

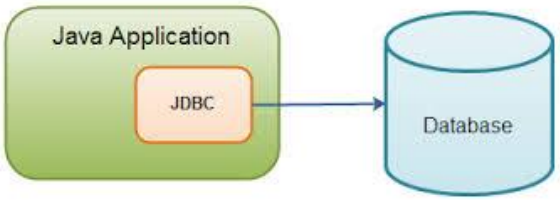
- ✓ API de acesso para executar comandos SQL;
- ✓ Implementado no **pacote padrão java.sql**;
- ✓ Envio para qualquer tipo de Banco de Dados relacional;
- ✓ Interface baseada no **X/OPEN SQL CLI**;
- ✓ Independente de **API**/Linguagem proprietária dos fabricantes de SGBD (IBM DB/2, Microsoft, Oracle, Informix, …)
- ✓ Uso de drivers específicos de fabricantes do SGBD.

A arquitetura JDBC

- ✓ **JDBC** é composto por um conjunto de **interfaces**, cada qual implementada diferentemente pelos fornecedores;
- ✓ O conjunto de classes que implementam as interfaces JDBC para um particular banco de dados é chamada **JDBC driver**;
- ✓ Os detalhes de como esta implementação foi feita é irrelevante (**encapsulamento**).

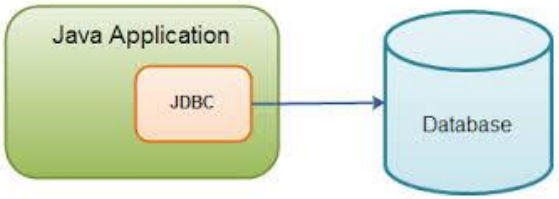
Interfaces JDBC

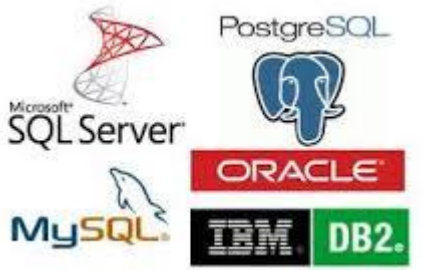
- Assim, **JDBC** é composto por um conjunto de **interfaces**, cada qual implementada diferentemente pelos fornecedores;
- O conjunto de classes que implementam as interfaces **JDBC** para um particular banco de dados é chamada **JDBC driver**;
- Os detalhes de como esta implementação foi feita é **irrelevante** (**encapsulamento**), uma vez que cada fornecedor SGBD implementou as classes definidas na interface de forma a atender as necessidades de seu SGBD em particular;



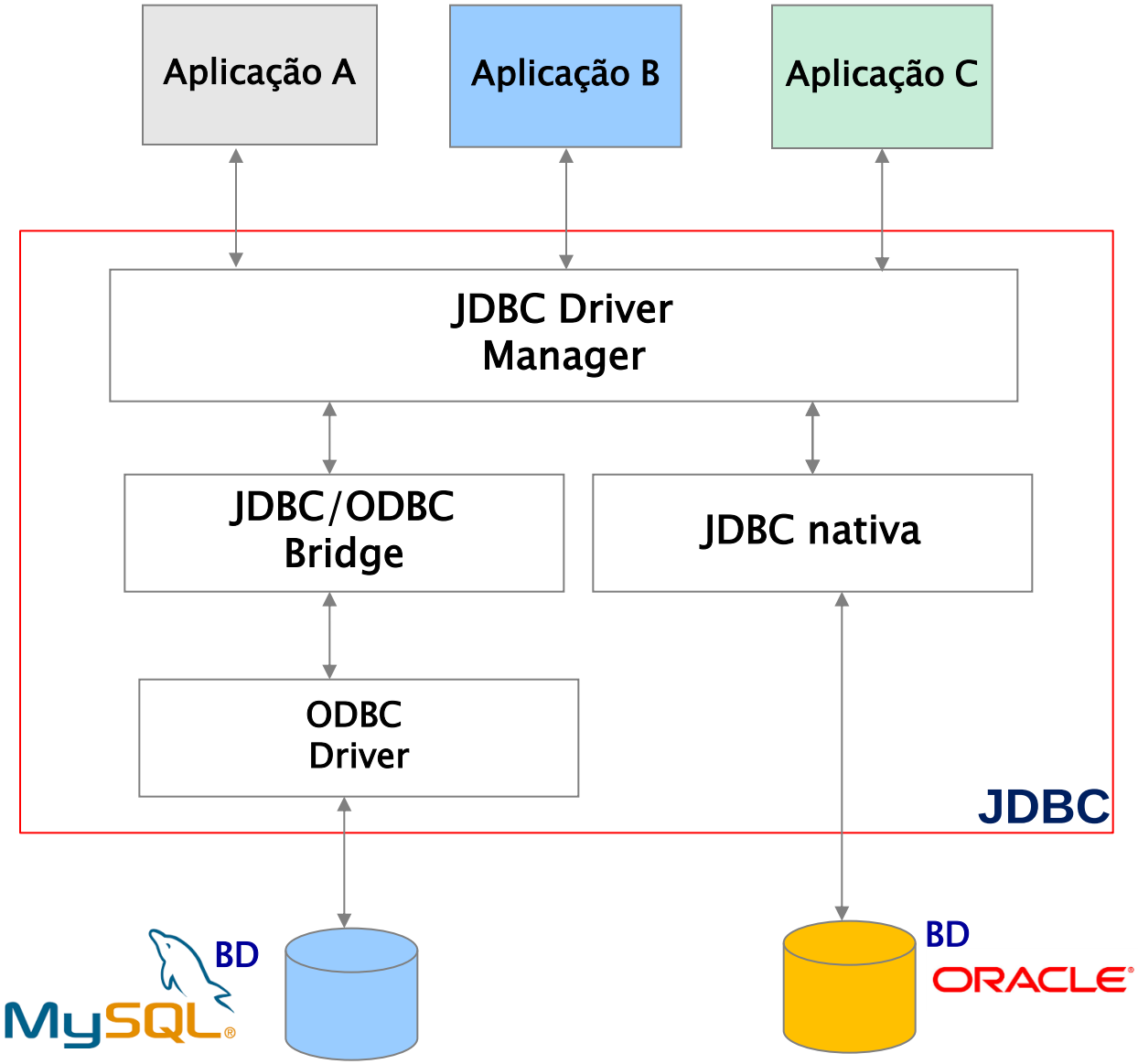
Arquitetura JDBC

- Aplicações Java "**conversam**" com o gerenciador de drivers JDBC (Driver Manager);
- Este, por sua vez, se comunica com algum **driver carregado em tempo de execução**;
- O programa de aplicação interage **apenas** com a API do gerenciador de drivers;
- O gerenciador de drivers interage com o driver específico do SGBD, que por sua vez interage com o SGBD;
- A API permite também que se use uma **BRIDGE** para o driver **ODBC**.





Arquitetura JDBC



Bridge ODBC-JDBC

- A maior parte dos fornecedores de Bancos de dados têm drivers **JDBC nativos** e assim pode-se instalá-los, seguindo as orientações do fabricante;
- Uma vez que **ODBC** existe para a maioria dos fabricantes de Bancos de Dados, uma **bridge JDBC-ODBC** também é disponibilizada pela API;
- Importante para se aprender **JDBC**, mas para o desenvolvimento em **ambientes de produção** recomenda-se utilizar drives **nativos**.

Pacote java.sql

- Na maioria definida por meio de **interfaces**;
- A implementação destas interfaces é feita pelo fornecedor do driver do banco de dados;
- Deste modo, a implementação destas interfaces fica por conta de quem entende de banco de dados (por exemplo: **DB2**, **Oracle**, etc);
- Com isso, a Sun padronizou o modo de se conectar ao banco de dados, liberando o driver para ser implementado pelos fornecedores que na verdade é que são especialistas em banco de dados.

Implementações JDBC

- O JDBC pode ser visto como um conjunto de interfaces cuja implementação deve ser fornecida por fabricantes de SGBD;
- Cada fabricante deve fornecer implementações de:
  - java.sql.Connection
  - java.sql.Statement
  - java.sql.PreparedStatement
  - java.sql.CallableStatement
  - java.sql.ResultSet
  - java.sql.Driver

- O objetivo é que fique transparente para o programador qual a implementação JDBC está sendo utilizada.

Instalação JDBC

- O pacote **JDBC** vêm incluso com as distribuições Java;
- As classes e interfaces que compões o kit **JDBC** estão nos pacotes **java.sql** e **javax.sql**;
- Entretanto, deve-se obter um **driver** para o sistema de gerência de banco de dados a ser utilizado.

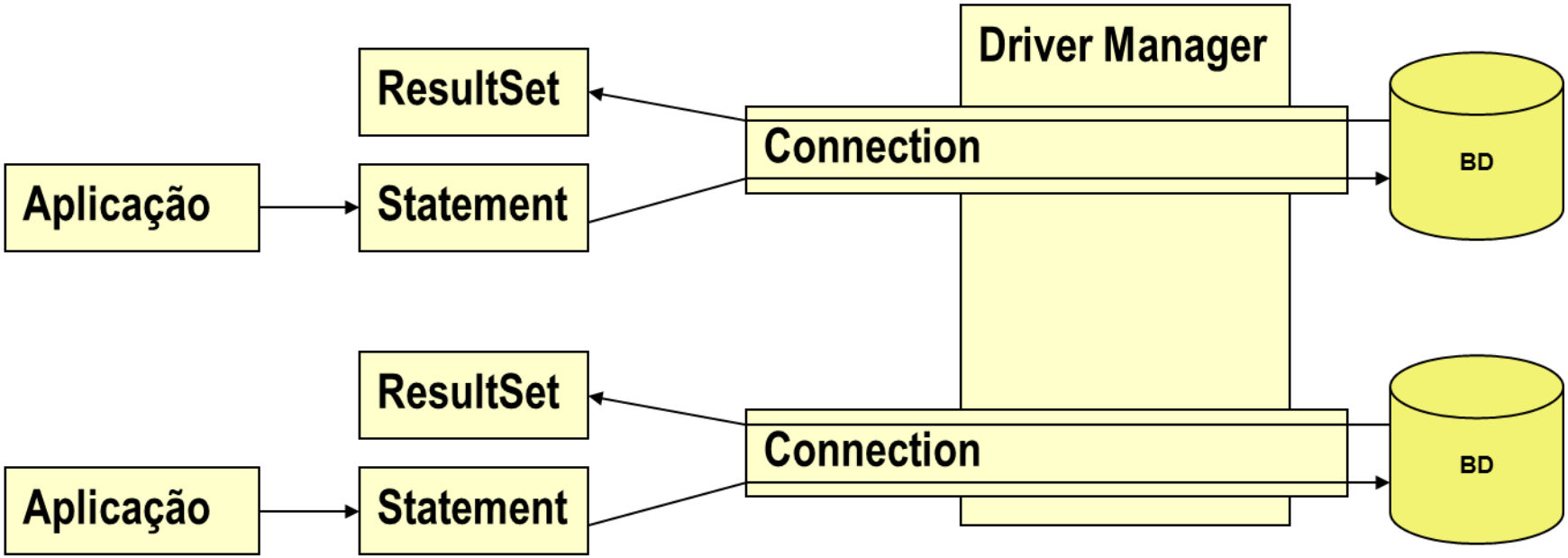
- Lista de drivers JDBC disponíveis:

http://www.oracle.com/technetwork/java/index-136695.html

Classes Principais - JDBC

- ✓ **java.sql.DriverManager**
  - Provê serviços básicos para gerenciar diversos drivers JDBC;
- ✓ **java.sql.Connection**
  - Representa uma conexão estabelecida com o BD;
- ✓ **java.sql.Statement**
  - Representa sentenças onde são inseridos os comandos SQL;
  - Permite realizar todo o tratamento das consultas (select) e dos comandos de atualizações (insert, delete, update)
- ✓ **java.sql.ResultSet**
  - Representa o conjunto de registros resultante de uma consulta;
  - Permite manipular os resultados;
  - Permite realizar coerção (cast) entre tipos Java e SQL;

Classes Principais - JDBC



Processamento de aplicação JDBC

- ✓ Definição de **qual driver** será utilizado na aplicação;
- ✓ **Carga** do driver;
- ✓ Criação do objeto **Connection** que será responsável pelas atividades de conexão banco de dados;
- ✓ Criação dos objetos **Statement** e **ResultSet** para envio de queries;
- ✓ Execução das queries;
- ✓ Processamento dos resultados;
- ✓ Fechamento (Close) da conexão.

Como efetuar conexão com o servidor de banco de dados ?

Classe DriverManager

- ✓ Responsável por **abrir uma conexão**, especificada através de uma URL, com uma base de dados, utilizando um determinado driver;
- ✓ Possui registro de todos os drivers já carregados;

Definição do Driver

- ✓ Numa aplicação **Java**, podemos ter vários drivers trabalhando ao mesmo tempo;
- ✓ A definição do driver é feita por meio de um String de conexão;
- ✓ O driver é um arquivo .jar e devemos tê-lo em um **classpath**, do contrário a aplicação não o encontrará.
- ✓ Deve-se anexar o driver no **classpath**, no instante da execução da aplicação Java.

```
java -classpath diretorio/meudriver.jar Minhaclasse
```

Carga do Driver

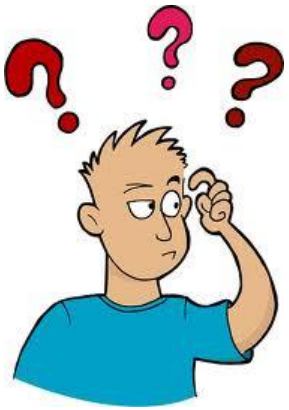
- Feita pelo método static **forName()** da classe **Class**, em tempo de run-time.
- Neste procedimento, o Class Loader tenta inicializar a classe que representa o driver.
- O driver possui um inicializador estático que irá registrar a classe que está sendo carregada como um driver JDBC, avisando o **java.sql.DriverManager** por meio do método **RegisterDriver**();
- O argumento para o método **Class.forName()** especifica o driver a ser registrado;
- O nome do driver definido como parâmetro consta na documentação do driver.

Registrando o Driver

- Exemplos:

JDBC-ODBC:	**sun.jdbc.odbc.JdbcOdbcDriver**
mySQL:	**com.mysql.jdbc.Driver**
PostGresql:	**org.postgresql.Driver**
Oracle:	**oracle.jdbc.driver.OracleDriver**
SqlServer:	**com.jnetdirect.jsql.JSQLDriver**
DB2:	**com.ibm.db2.jdbc.app.DB2Driver**

Como se registra o driver nativo ORACLE ?

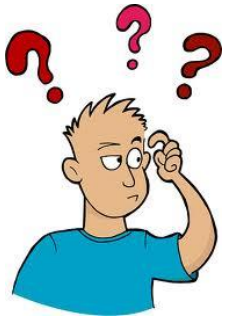


ORACLE®

Carga do Oracle JDBC driver

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Como se registra o driver nativo MySQL?

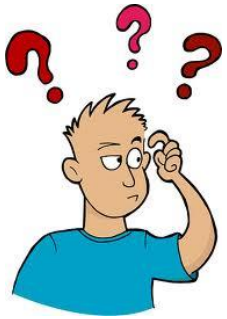


Carga do driver nativo MySQL

```
Class.forName("com.mysql.jdbc.Driver");
```

ORACLE
MySQL

Como se registra o driver nativo DB/2?



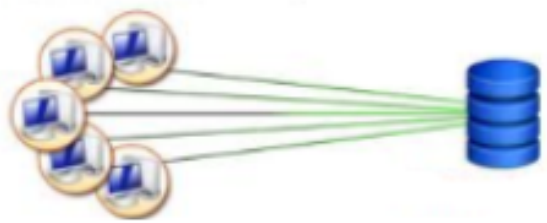
Carga do IBM DB2 JDBC driver

```
Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
```

Conexão ao Banco de Dados

- Após a carga e registro do driver como sendo um driver **JDBC**, pode-se abrir uma conexão com o banco criando-se um objeto do tipo **Connection** através da chamada do método **getConnection()** da classe **DriverManager**.

- O método **getConnection()** recebe como parâmetros informações relativas ao **data-source (URL)**, **usuário** e **senha** para autenticação.



Exemplo getConnection() com Driver nativo MySQL

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

public class TestConnection {

        public static Connection createConnection() throws SQLException {

                String url = "jdbc:mysql://localhost:3306/loja";
                String user = "root";
                String password = "root";

                Connection conexao = null;

                conexao = DriverManager.getConnection( url, user, password );

                return conexao;
        }
}
```

Uma vez conectado ao BD, como enviar comandos para o SGBD?

Comandos SQL

- Comandos SQL podem ser diretamente enviados ao SGBD por meio de um objeto instanciado por uma classe que implemente a interface **Statement**;
- Comandos de definição de dados (**DDL**) e de consultas são aceitos;
- Há dois tipos básicos de comandos SQL:
  - **Statement**: Envia texto SQL ao SQGD;
  - **PreparedStatement**: Pré-compila o texto SQL, com posterior envio ao SGBD;

Statements

- ✓ Um objeto **Statement** é uma espécie de canal que envia comandos SQL através de uma conexão;
- ✓ O mesmo **Statement** pode enviar vários comandos;
- ✓ Para se criar um **Statement**, é preciso ter criado anteriormente um objeto Connection;
- ✓ A partir de uma conexão, pode-se criar diversos objetos **Statement**.

Criação do objeto Statement

- O objeto **Statement** será responsável pelo envio dos comandos **SQL** ao **DBMS**.
- Este objeto é criado pelo método **createStatement()** executado pelo objeto Connection.

```
Connection dbCon = DriverManager.getConnection (
                              "jdbc:odbc:curso",
                              "admin",
                              "secret" ) ;

Statement     stmt = dbCon.createStatement();
```

Como se executa os comandos SQL ?

Executando Statements

- Há dois métodos da classe Statement para envio de comandos ao SGBD.
- **Modificações**: **executeUpdate()**

- ✓ Para comandos SQL "**INSERT**", "**UPDATE**", "**DELETE**", ou outros que alterem a base de dados e não retornem dados;
- ✓ Forma geral: **executeUpdate(<comando>);**
- ✓ Exemplo: **stmt.executeUpdate("DELETE FROM Cliente");**

Executando Statements

- **Consultas**: **executeQuery()**

- ✓ Para comandos SQL "**SELECT**" ou outros retornem tuplas;
- ✓ Forma geral: **executeQuery(<comando>);**
- ✓ Esse método retorna um objeto da Classe **ResultSet**;
- ✓ Exemplo: **stmt.executeQuery("SELECT * FROM Cliente");**

Exemplo - Statement


```
Class.forName("org.postgresql.Driver");

Connection conn = DriverManager.getConnection( "jdbc:postgresql:usuarios");

Statement stat = conn.createStatement();

ResultSet nomes = stat.executeQuery("SELECT nomes FROM pessoas");
```

Exemplo - executeQuery()

- O método **executeQuery()** executa comandos **SQL** do tipo **SELECT**;
- Retorna um objeto do tipo **ResultSet**.

```
Connection    dbCon = DriverManager.getConnection (
                          "jdbc:odbc:curso",
                          "admin",
                          "secret" ) ;

Statement     stmt = dbCon.createStatement();

ResultSet     rs = stmt.executeQuery(
        "SELECT nome_curso FROM curso");
```

Exemplo - executeUpdate()

- O método executeQuery() é usado para submeter statements SQL do tipo DML/DDL;
- DML é usado para manipular dados existentes em objetos (por meio de UPDATE, INSERT, DELETE statements).
- DDL é usado para manipular objetos database (CREATE, ALTER, DROP).

```
Statement     stmt = dbCon.createStatement();

stmt.executeUpdate("INSERT INTO tabcurso
VALUES(1,'Psicologia')" );
```

O Objeto ResultSet

- Mantém o posicionamento do cursor em sua corrente linha de dados;
- Provê métodos para recuperar valores de colunas.

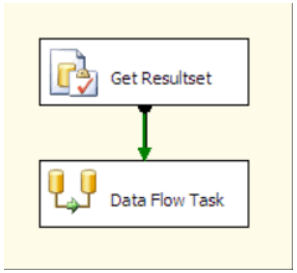
```
ResultSet     rs = stmt.executeQuery(
       "SELECT nome_curso FROM curso");

while    (rs.next() ) {

       String nome_curso = rs.getString("nome_curso");
       double valor = rs.getDouble("preco");
}
```

Funções de acesso ao ResultSet

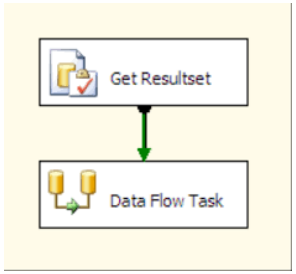
- Métodos de acesso aos dados têm duas formas: Uma forma tem um argumento numérico e outra com argumento **String**.
- Quando se fornece um argumento numérico, está se referindo à coluna que corresponde àquele valor.
- Quando se fornece um argumento String se refere à coluna cujo nome corresponde ao String fornecido.



Manipulação de objetos ResultSet

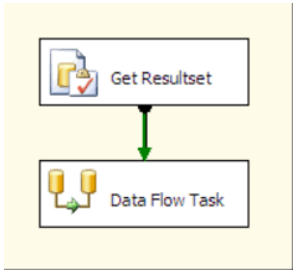
- ✓ Métodos **getXXX**
  - Recuperam um dado de acordo com o tipo;
  - Formas: rs.**getXXX**(<nome do campo>) ou rs.**getXXX**(<posição do campo >);
  - Exemplo:rs. **getString**(“nm_cliente”) ou rs.**getString**(2);

- ✓ Método **next()** , **previous()**
  - Retorna para o próximo registro no conjunto ou para o anterior;
  - Retornam valor lógico;
  - Valor de retorno **true** indica que há outros registros para serem processados.



Manipulação de objetos ResultSet

- ✓ Métodos **first() , last()**
  - Posicionam o cursor no início ou no final do conjunto de dados;
- ✓ Método **next()** , **previous()**
  - Testam a posição do curso;
  - Retornam valor lógico.

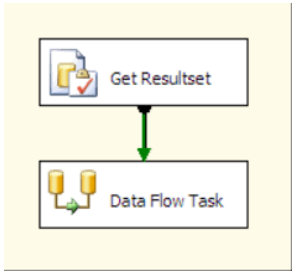


Acessores para tipos Java

- **rs.getString(1)** retorna o valor da primeira coluna na linha corrente.
- **rs.getDouble(“Price”)** retorna o valor da coluna com nome “Price”.

Encerramento da conexão

- ✓ Explicitamente fecham a conexão, por meio da função **close**() aplicada aos objetos **Connection**, **Statement** e **ResultSet**.
- ✓ Este procedimento irá liberar os recursos que não são mais necessários à aplicação.



PreparedStatement

- ✓ Os métodos **executeQuery** e **executeUpdate** da classe **Statement** não recebem parâmetros;
- ✓ **PreparedStatement** é uma subinterface de Statement cujos objetos permitem a passagem de parâmetros;
- ✓ Em um comando **SQL** de um objeto **PreparedStatement**:
  - Parâmetros são simbolizados por pontos de interrogação;
  - Configuração dos valores dos parâmetros: métodos **setXXX**

```
PreparedStatement pst =
    con.prepareStatement("INSERT INTO Clientes (codigo, nome) VALUES  (?,?)");
      pst.setInt(1,10);
      pst.setString(2,"Eduardo");
```

PreparedStatement - Exemplo

```
PreparedStatement stat = conn.prepareStatement("SELECT * FROM ?");
// percorre os funcionários
stat.setString(1, "Funcionarios");
ResultSet funcionários = stat.executeQuery();
.....
// percorre os produtos
stat.setString(1, "Produtos");
ResultSet produtos = stat.executeQuery();
.....
```

Atividade - JDBC

Escrever um **programa** desktop que faça uma conexão a um banco de dados e insira um registro. Acessar o Servidor de Banco de Dados MySQL. Utilizar o Driver JDBC nativo.

Obs. a) Nome do database: CURSO
b) Nome da tabela: TABCURSO

Código do Curso	Nome do Curso
CODCURSO int(2)	NOMECURSO char(50)

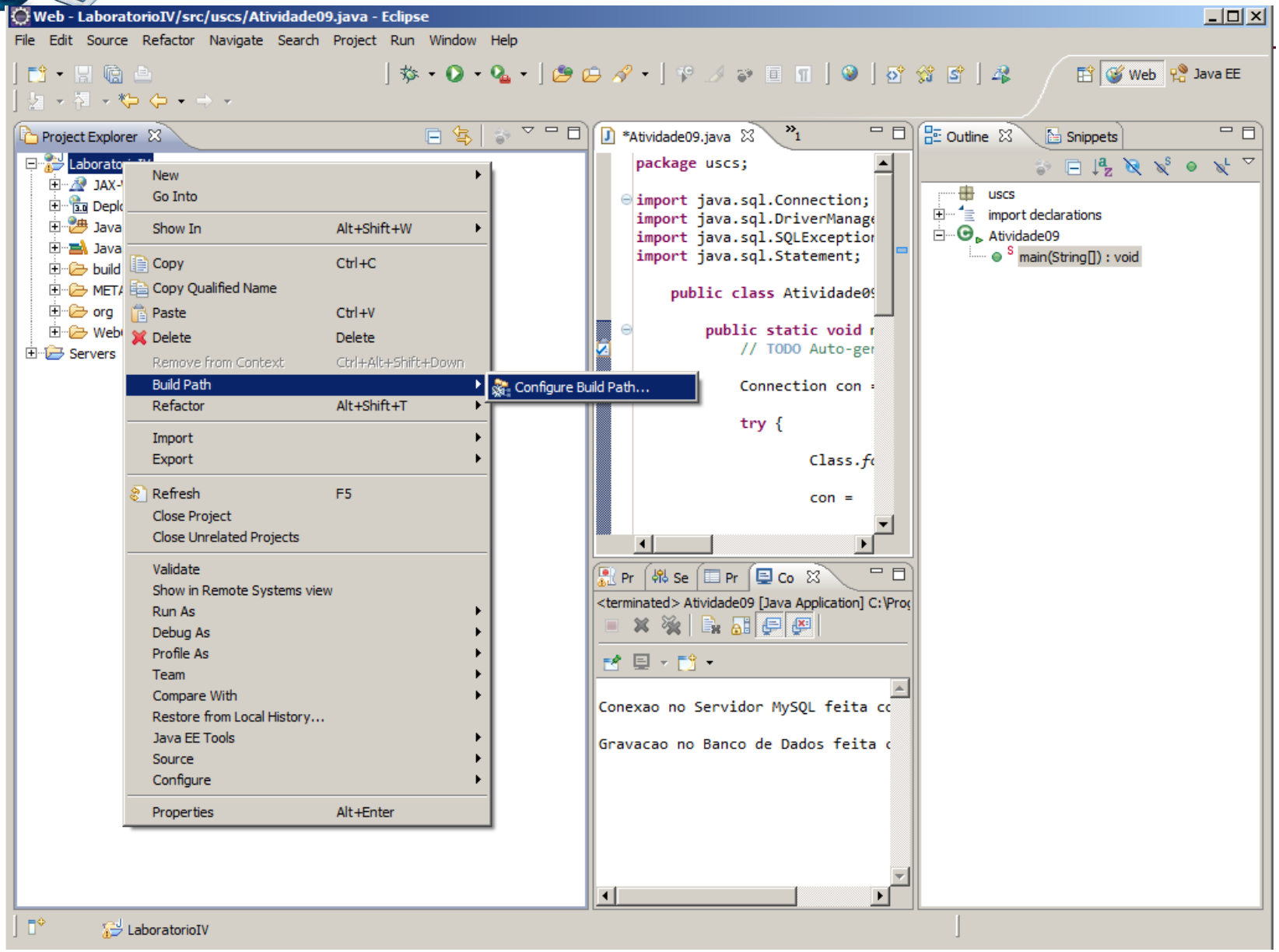
Driver JDBC MySQL nativo

- ✓ Baixar o driver a partir do endereço:

https://dev.mysql.com/downloads/connector/j/

- ✓ Salvar em algum diretório do Servidor;
- ✓ Configurar o Path do Eclipse para que o projeto visualize o driver.

Configuração do Path - Eclipse



Configuração do Path - Eclipse

Properties for LaboratorioIV

type filter text

- Resource
- Builders
- Deployment Assembly
- Java Build Path
- Java Code Style
- Java Compiler
- Java Editor
- Javadoc Location
- JavaScript
- JSP Fragment
- Project Facets
- Project References
- Refactoring History
- Run/Debug Settings
- Server
- Service Policies
- Targeted Runtimes
- Task Repository
- Task Tags
- Validation
- Web Content Settings
- Web Page Editor
- Web Project Settings
- WikiText
- XDoclet

Java Build Path

Source | Projects | Libraries | Order and Export

JARs and class folders on the build path:

- LaboratorioIV/com (class folder)
- mysql-connector-java-5.1.18-bin.jar - C:\SOFTW
- Apache Tomcat v7.0 [Apache Tomcat v7.0]
- EAR Libraries
- JRE System Library [jdk1.6.0_26]
- Web App Libraries

Add JARs...
Add External JARs...
Add Variable...
Add Library...
Add Class Folder...
Add External Class Folder...
Edit...
Remove
Migrate JAR File...

OK Cancel

Parâmetros de Conexão

```
Connection con = null;

Class.forName("com.mysql.jdbc.Driver").newInstance();

con = DriverManager.getConnection("jdbc:mysql://localhost/curso?" +
                " user=root&password="+ "" );
```

```
package uscs;

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

public class Atividade_MySQL {

public static void main(String[] args) {
// TODO Auto-generated method stub

Connection con = null;

try {

        Class.forName(" com.mysql.jdbc.Driver " ).newInstance();

        con = DriverManager.getConnection( " jdbc:mysql://localhost/curso?"  +
                " user=root&password="+ "" );

        System.out.println("\nConexao no Servidor MySQL feita com sucesso...");
```

```
Statement stmt = con.createStatement();
String command = "INSERT INTO tabcurso VALUES(2,'Matematica')" ;

stmt.executeUpdate(command);
System.out.println("\nGravacao no Banco de Dados feita com sucesso...");

}
catch  (SQLException ex) {

        System.out.println ("**** ERRO DE ACESSO AO BANCO DE DADOS...\n");
        System.out.println ("****SQLException: " + ex);

}

catch (Exception ex) {

        System.out.println("*****Exception: " + ex)

}

}

}
```