



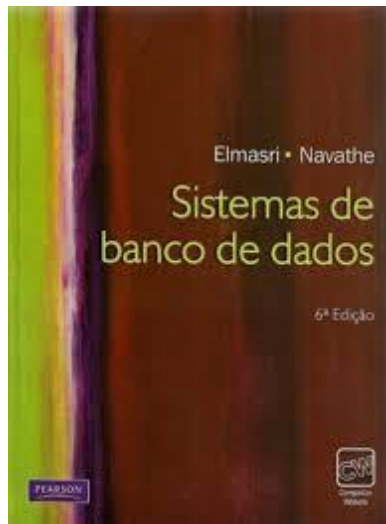
Unidade 18 – Programação SQL com chamadas CLI



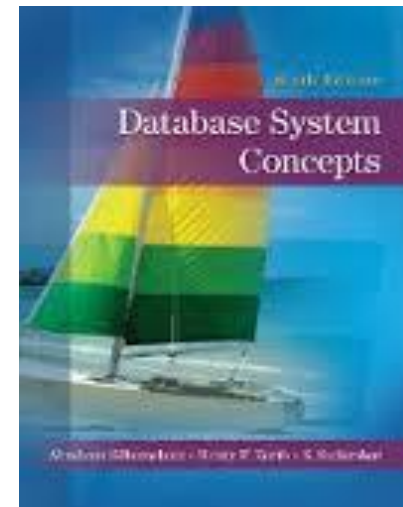
Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP



Bibliografia



Sistemas de Banco de Dados
Elmasri / Navathe 6ª edição



Sistema de Banco de Dados
Korth, Silberschatz – Sixth Edition



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;





Exemplo SQL Embutido

```

*-----
  IDENTIFICATION DIVISION.
  PROGRAM-ID.  PGMCOB07.
*-----
  ENVIRONMENT DIVISION.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
      EXEC SQL
          INCLUDE SQLCA
      END-EXEC.
*-----
  PROCEDURE DIVISION.
*-----
    100-INICIO.
      EXEC SQL
          WHENEVER SQLERROR GO TO 900-SQL-ERROR
      END-EXEC.
      PERFORM 200-PROCESSA.
      STOP RUN.
*-----
    200-PROCESSA.
      EXEC SQL
          INSERT INTO AVFREITAS1.TABCLIENTE
              (CODCLI ,NOMECLI ,ENDCLI)
              VALUES ('1' , 'Ana' , 'RUA 1')
      END-EXEC.
*-----
    900-SQL-ERROR.
      DISPLAY SQLCODE.
      DISPLAY SQLSTATE.
  
```

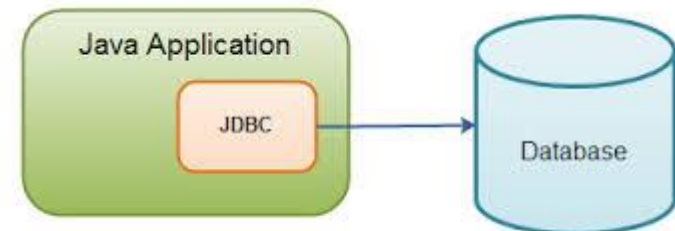




Programação SQL – CLI



- ✓ 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**;
- ✓ Exemplo: MySQL – C API – **MySQL Connector/C**
- ✓ 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**.





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**.





Exemplo: MySQL Connector/C



- ✓ **Documentação** – Capítulo 27 – Manual Referência – MySQL – Página 4111
- ✓ Provê acesso baixo nível ao servidor MySQL e habilita programas escritos na linguagem C a acessar conteúdos de banco de dados;
- ✓ A maioria dos clientes **API** usam a biblioteca **libmysqlclient** para se comunicar com o Servidor MySQL.
- ✓ **MySQL API C** é uma **API C-based** que pode ser usada para a comunicação com o **MySQL** server.





Exemplo: MySQL Connector/C



- ✓ A MySQL C API é definida por meio de um conjunto de headers files, que devem ser incluídos nos arquivos fontes para compilação;
- ✓ O header file **mysql.h** define a API cliente para MySQL.





Exemplo: MySQL Connector/C



```
//MySQL C API headers  
#include <my_global.h>  
#include <mysql.h>
```

```
int main(int argc, char **argv) {  
    //Print the libmysqlclient library version  
    printf("The MySQL client version is : %s\n", mysql_get_client_info());  
    return 0;  
}
```

```
Shell> gcc -o c_client c_client.c -I/usr/include/mysql/ -lmysqlclient
```





Exemplo: MySQL Connector/C



```
//Include the MySQL C API client header files
```

```
#include <my_global.h>
```

```
#include <mysql.h>
```

```
int main(int argc, char **argv) {
```

```
    //The Connection handle object
```

```
    MYSQL connection;
```

```
    //Initialize the connection handle
```

```
    if (mysql_init(&connection) != NULL) {
```

```
        printf("Connection handle initialized\n");
```

```
    } else {
```

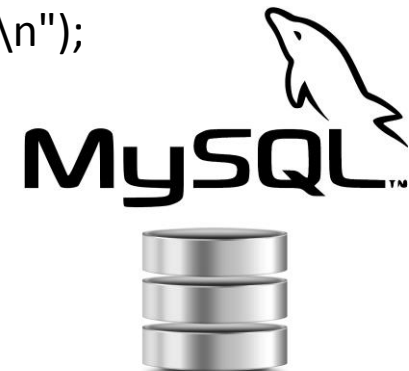
```
        printf("Connection handle initialization failed\n");
```

```
        exit(1);
```

```
    }
```

```
    return 0;
```

```
}
```





Exemplo: MySQL Connector/C



```
//Connect to MySQL server
```

```
if (mysql_real_connect(&connection, "localhost", "user", "user_password",  
"database_name", 3306, NULL, 0) != NULL) {  
    printf("Connection to remote MySQL server established\n");  
} else {
```

```
    printf("Connection attempt to remote MySQL failed !\n");  
    exit(1);
```

```
}
```

```
//Close the connection
```

```
mysql_close(&connection);  
return 0;
```

```
}
```





Exemplo: MySQL Connector/C



```
//Create the users_database database
```

```
if (mysql_query(&connection, "CREATE DATABASE users_database") == 0){  
    printf("Database created\n");  
} else{  
    printf("Database creation failed");  
    printf("MySQL error message: %s\n", mysql_error(&connection));  
    exit(1);  
}
```

```
//Switch to use the users_database database
```

```
mysql_query(&connection, "USE users_database");
```





Exemplo: MySQL Connector/C



```
//Create the users_table table
```

```
if (mysql_query(&connection,  
    "CREATE TABLE users_table(\n  
        Login VARCHAR(20) PRIMARY KEY,\n  
        FirstName VARCHAR(40) NOT NULL,\n  
        LastName VARCHAR(40) NOT NULL,\n  
        )" == 0) {\n    printf("Table created\n");\n} else{\n    printf("Table creation failed");\n    printf("MySQL error message: %s\n", mysql_error(&connection));\n    exit(1);\n}
```

```
//Close the connection
```

```
mysql_close(&connection);\nreturn 0;
```

```
}
```





Exemplo: MySQL Connector/C



//Execute an SQL statement that returns a result

```
mysql_query(&connection, "SELECT * FROM users_table");
```

//Retrieve the result set

```
MYSQL_RES * result = mysql_store_result(&connection);
```

//At this point the processing of the result set is to be done

//Free the result set

```
mysql_free_result(result);
```

//Close the connection

```
mysql_close(&connection);
```

```
return 0;
```

```
}
```





Exemplo: MySQL Connector/C



//Execute an SQL statement that returns a result

```
mysql_query(&connection, "SELECT * FROM users_table");
```

//Retrieve the result set

```
MYSQL_RES *result = mysql_store_result(&connection);
```

```
for (int i = 0; i < mysql_num_fields(result); i++) {
```

```
    MYSQL_FIELD *column = mysql_fetch_field(result);
```

```
    printf("Column number %i is named : %s\n", i + 1, column->name);
```

```
}
```

//Free the result set

```
mysql_free_result(result);
```

//Close the connection

```
mysql_close(&connection);
```

```
return 0;
```





MySQL Connector/C++

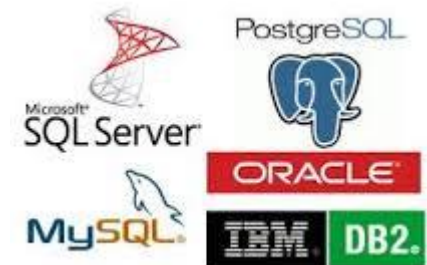




MySQL Connector/C++



- ✓ MySQL Connector/C++ é um driver C++ para acesso ao servidor de banco de dados MySQL;
- ✓ Oferece o suporte ao Paradigma Orientado a Objetos;
- ✓ Requer MySQL 5.1 ou maior;
- ✓ Requer Microsoft Visual Studio 2013 na plataforma Windows;
- ✓ Pode ser usada em uma aplicação como uma biblioteca estática ou dinâmica;
- ✓ O arquivo referente à biblioteca estática é: **mysqlcppcon-static.lib**. Esta biblioteca deve ser linkada estaticamente à aplicação;
- ✓ O arquivo referente à biblioteca dinâmica é: **mysqlcppcon.dll**



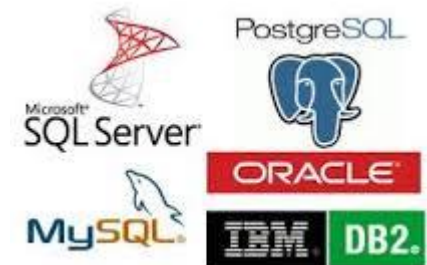


MySQL Connector/C++ Compilação Visual Studio



- ✓ Configuração da ide MS-Visual Studio disponível no MySQL API C++ user guide.

MySQL Connector/C++ Developer Guide



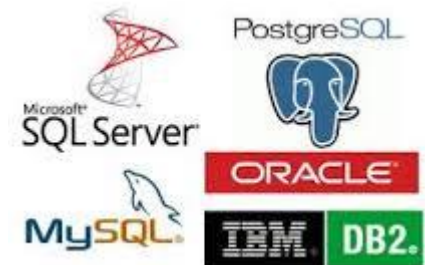


MySQL Connector/C++ Compilação NetBeans – Aplicações Linux



- ✓ Configuração da ide NetBeans disponível no MySQL API C++ user guide.

MySQL Connector/C++ Developer Guide

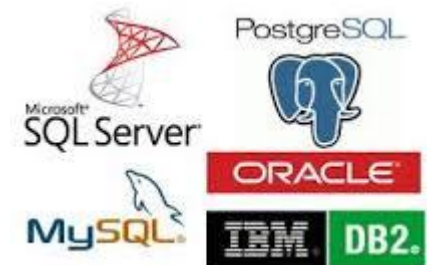




MySQL Connector/C++ Conexão



```
sql::mysql::MySQL_Driver *driver;  
sql::Connection *con;  
driver = sql::mysql::get_mysql_driver_instance();  
con = driver->connect("tcp://127.0.0.1:3306", "user", "password");  
delete con;
```

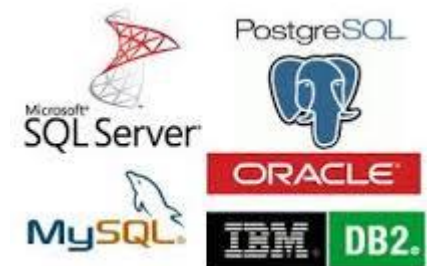




MySQL Connector/C++ Execução de Query



```
sql::mysql::MySQL_Driver *driver;  
sql::Connection *con;  
sql::Statement *stmt;  
driver = sql::mysql::get_mysql_driver_instance();  
con = driver->connect("tcp://127.0.0.1:3306", "user", "password");  
stmt = con->createStatement();  
stmt->execute("USE " EXAMPLE_DB);  
stmt->execute("DROP TABLE IF EXISTS test");  
stmt->execute("CREATE TABLE test(id INT, label CHAR(1))");  
stmt->execute("INSERT INTO test(id, label) VALUES (1, 'a')");
```

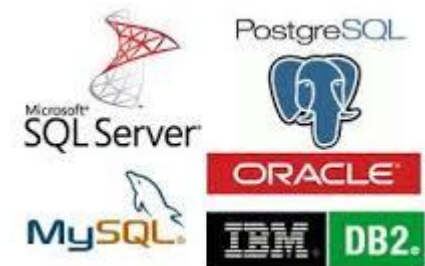




MySQL Connector/C++ Execução de Query



```
// ...  
sql::Connection *con;  
sql::Statement *stmt;  
sql::ResultSet *res;  
// ...  
stmt = con->createStatement();  
// ...  
res = stmt->executeQuery("SELECT id, label FROM test ORDER BY id ASC");  
while (res->next()) {  
    // You can use either numeric offsets...  
    cout << "id = " << res->getInt(1); // getInt(1) returns the first column  
    // ... or column names for accessing results.  
    // The latter is recommended.  
    cout << ", label = '" << res->getString("label") << "'" << endl;  
}
```





MySQL Connector/J 5.1

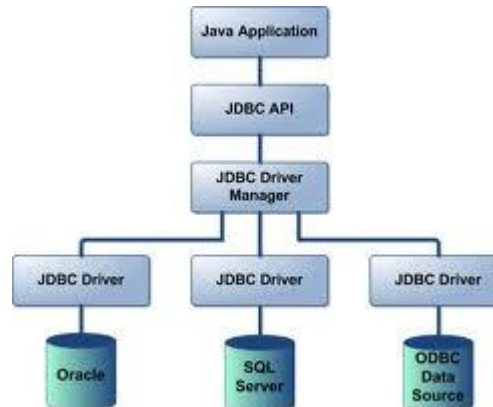




JDBC



- ✓ 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;
- ✓ Esta 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

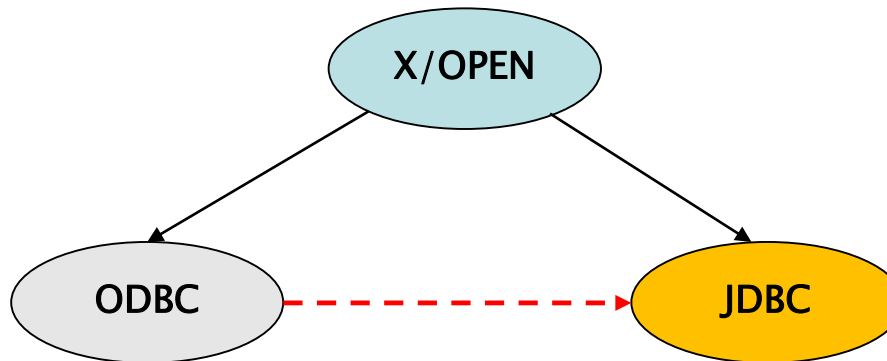
- ✓ API de acesso para executar comandos SQL;
- ✓ Implementado no **pacote padrão java.sql**;
- ✓ Envio para qualquer tipo de Banco de Dados relacional;
- ✓ Interface baseada no **X/OPEN SQL CLI**;
- ✓ Independente de **API**/Linguagem proprietária dos fabricantes de SGBD (IBM DB/2, Microsoft, Oracle,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





Objetivos JDBC

- ✓ Permitir que programadores Java possam escrever aplicações para acessar qualquer banco de dados;
- ✓ Permitir que fornecedores de SGBD possam fornecer drivers e otimizar estas liberações;
- ✓ Permitir a integração de driver **ODBC** com **JDBC** (bridge).





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).





JDBC é composta por um conjunto de interfaces.

Mas, o que são interfaces ?





Interfaces

- Imagine que você irá desenvolver uma aplicação no qual duas equipes irão desenvolver o software de forma simultânea;
- Cada equipe irá desenvolver seus códigos de forma independente.





Interfaces



- Ⓢ No entanto, deverá haver um “**contrato**” entre as equipes de tal modo que haja interação entre os códigos. Este contrato é conhecido por **interface**.
- Ⓢ Interface é uma forma de descrever o **quê** as classes devem fazer, sem especificar **como** elas devem fazê-lo.
- Ⓢ Uma classe pode implementar **uma** ou **mais interfaces**.
- Ⓢ Uma interface não é uma classe mas um conjunto de requisitos para as classes que quiserem implementá-la. Esses requisitos devem ser seguidos pelos fabricantes de sistemas gerenciadores de banco de dados, para garantir-se o padrão de acesso.
- Ⓢ Em Java, uma interface é uma definição de tipo, semelhante à classe, que pode conter apenas constantes e assinatura de métodos (protótipos).
- Ⓢ Numa interface não há corpo de definição de método. Não podem ser instanciadas. São implementadas por classes ou ainda estendidas em outras interfaces.





Interface

- ⌚ Todos os métodos de uma interface são automaticamente **public**;
- ⌚ Por esta razão não há necessidade de incluir a keyword **public** quando estivermos declarando um método em uma interface;
- ⌚ Tendo em vista que interfaces não são classes, nunca se pode usar o operador **new** para instanciar uma interface.





Interface – Exemplo

```
public interface A12 {  
    Integer func (Integer n) ;  
}
```

- ⌚ Isto significa que para qualquer classe que implementa a interface **A12** é requerido que se tenha a definição do método **func** e o método deve receber um argumento **Integer** e retornar um objeto do tipo **Integer**;
- ⌚ Assim, os fabricantes de **Sistemas Gerenciadores de Bancos de Dados** deverão respeitar a assinatura das funções presentes na interface, garantindo-se assim, o padrão de acesso ao banco de dados para qualquer **SGBD**.





Interfaces

- ② Uma interface é essencialmente uma coleção de **constantes** e **métodos abstratos**;
- ② Para fazer uso de uma interface, você implementa a interface em alguma classe;
- ② Ou seja, você declara que a classe implementa a interface e escreve o código para cada método declarado na interface;
- ② Quando uma classe implementa uma interface, quaisquer constantes que foram definidas na interface **são diretamente disponíveis na classe**, como se fossem herdados de uma classe base.



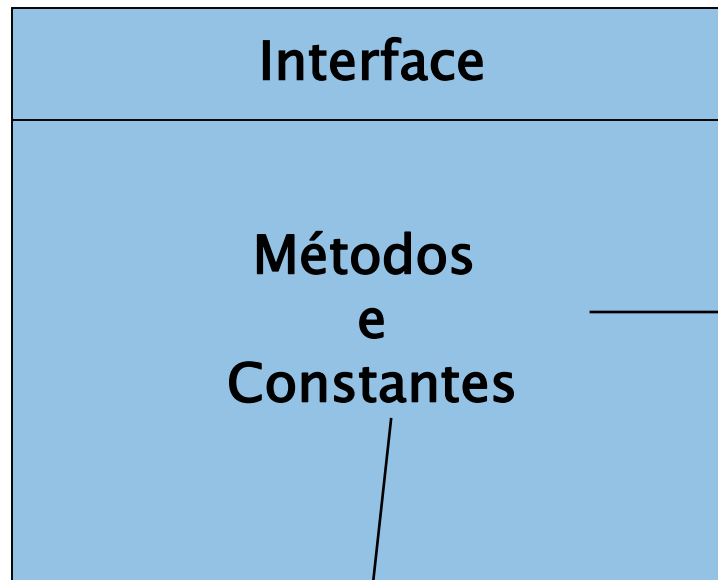


O que pode conter uma interface ?





Interface com métodos e constantes



São sempre **public** e **abstract** por **default**;

São sempre **public**, **static** e **final** por default;





Para que serve Interface?

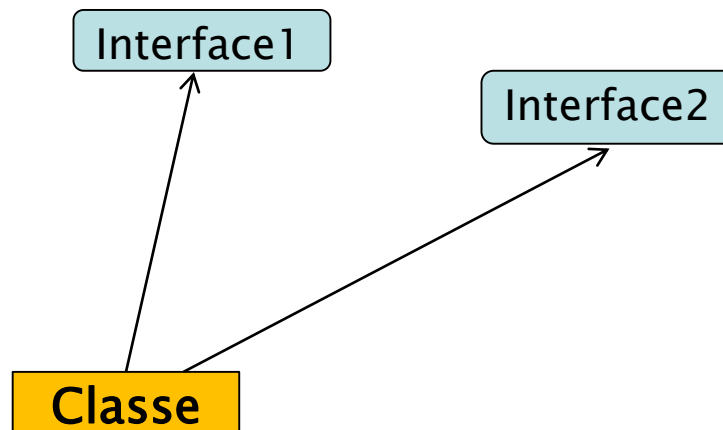
- @ Estabelecer um contrato entre a classe que implementa a função declarada na interface;
- @ No entanto, há outra finalidade. Em Java, uma classe **A** pode somente estender uma simples classe **B**;
- @ A classe **A** não pode estender uma segunda classe **C**. (**A** não pode ter dois pais...);
- @ Outras linguagens, tais como C++, permitem que uma classe tenha mais de uma superclasse;
- @ Esta feature, não suportada por Java, denomina-se **herança múltipla**;
- @ Java no entanto, por razões de eficiência e simplicidade oferece o mecanismo de **interface** para suportar herança múltipla.



Para que serve Interface?

```
public class C1 implements A12 {  
    Integer func (Integer n) {  
        // código com implementação  
    }  
}
```

- Java não permite herança múltipla, mas interfaces provêem uma alternativa;
- Em Java, uma classe pode ser herdada de somente uma classe mas ela pode implementar mais de uma interface.





Para que serve Interface?

- ⊕ É um meio de empacotar constantes;
- ⊕ Você pode usar uma interface contendo constantes em qualquer número de diferentes classes que tenham acesso à interface;
- ⊕ As constantes são static e assim são compartilhadas entre todos os objetos da classe.

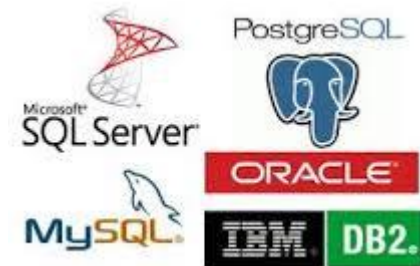
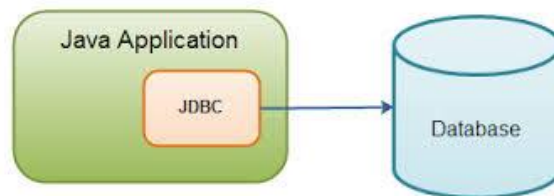




Interfaces JDBC



- ⊕ 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;

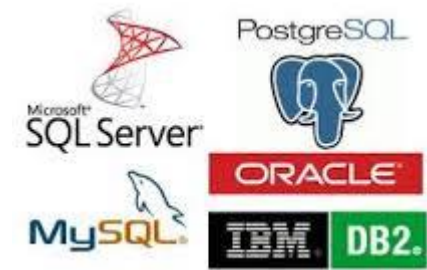
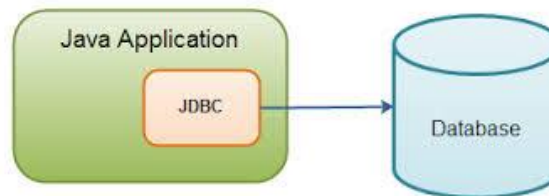




Arquitetura JDBC

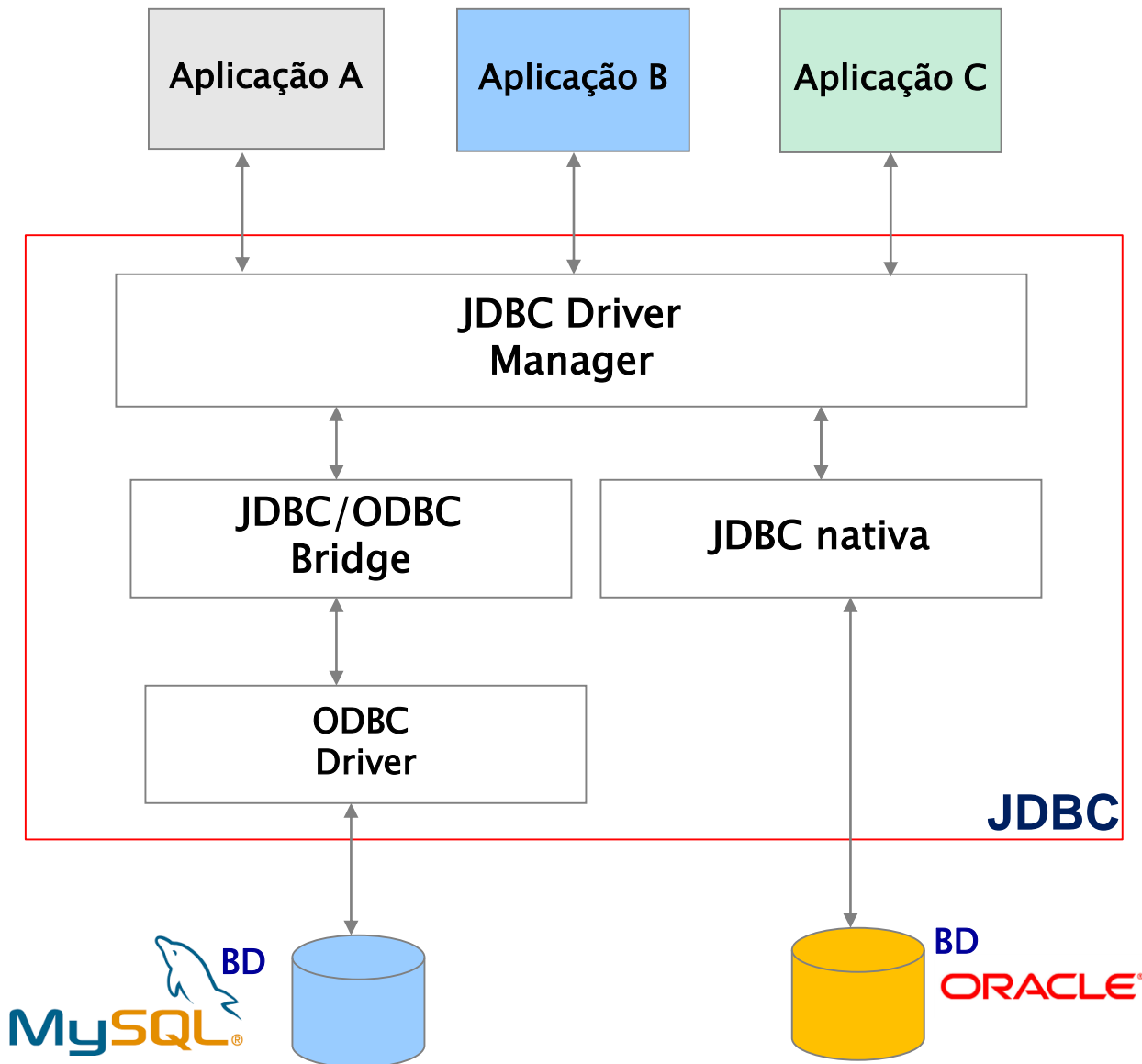


- ⊕ Aplicações Java “**conversam**” com o gerenciador de drivers JDBC (**Driver Manager**);
- ⊕ Este, por sua vez, se comunica com algum **driver carregado em tempo de execução**;
- ⊕ O programa de aplicação interage **apenas** com a API do gerenciador de drivers;
- ⊕ O gerenciador de drivers interage 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**.





Bridge ODBC-JDBC



- Tem a **vantagem** de permitir que as pessoas usem JDBC imediatamente (Não requer instalação de drivers);



- Tem a **desvantagem** de requerer uma outra camada entre o banco de dados e JDBC.





Conceitos JDBC

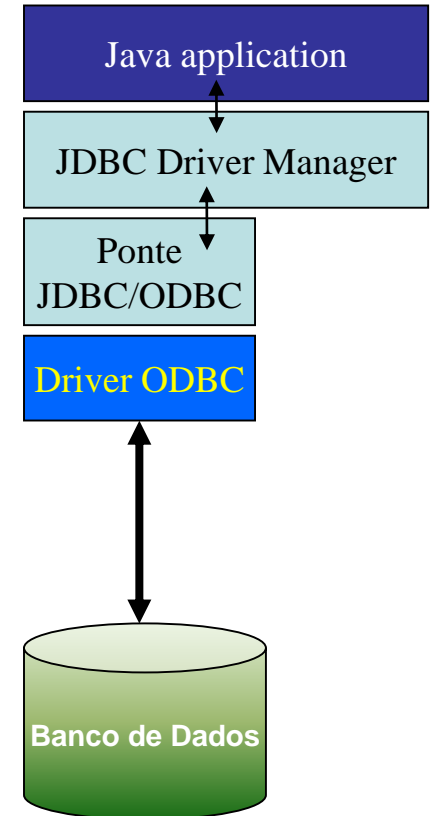
- A programação com interfaces **JDBC** não é diferente da programação Java usual;
- Basicamente constrói-se objetos a partir da **API**, empregando-se herança se necessário;
- O driver JDBC estabelece conexão com o Sistema Gerenciador de Banco de Dados;
- Requisições SQL são enviadas ao SGBD que processa a consulta;
- Existem 4 tipos de drivers JDBC.





Driver JDBC -Tipo 1

- Empregam tecnologia de **BRIDGE** para acesso ao banco de dados;
- Exemplo: JDBC-ODBC bridge incorporada ao **JDK**;
- Esta solução requer instalação de Driver ODBC no cliente.

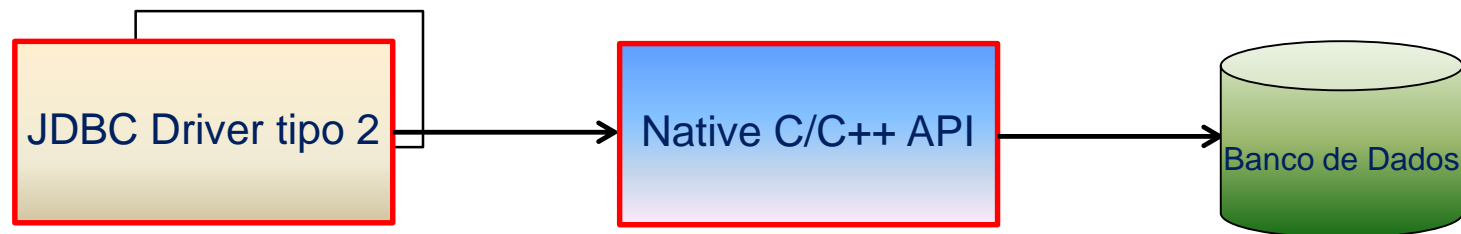




Driver JDBC -Tipo 2



- O driver contém código Java que efetua chamadas nativas nos métodos C ou C++ disponibilizados pelos fornecedores de BD;
- O driver JDBC se comunica com API C/C++ por meio de funções **JNI – Java Native Interface**.

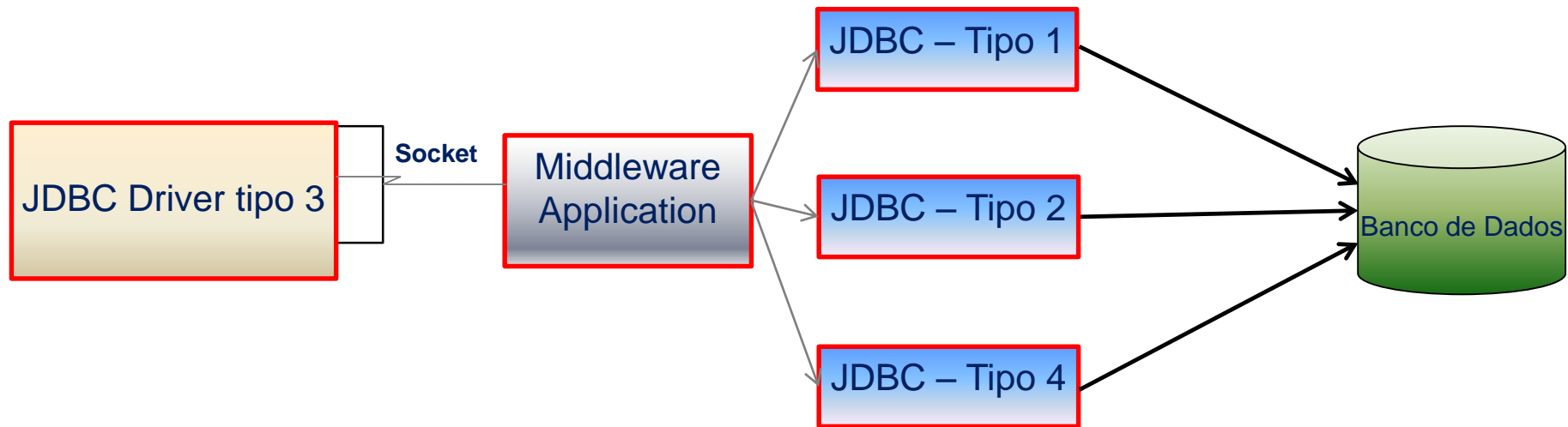




Driver JDBC -Tipo 3



- Driver no cliente faz uso de conexão **socket** para chamar uma aplicação middleware no servidor que transfere o request do cliente para uma API específica do driver desejado.
- Solução flexível pois não requer código instalado no cliente.

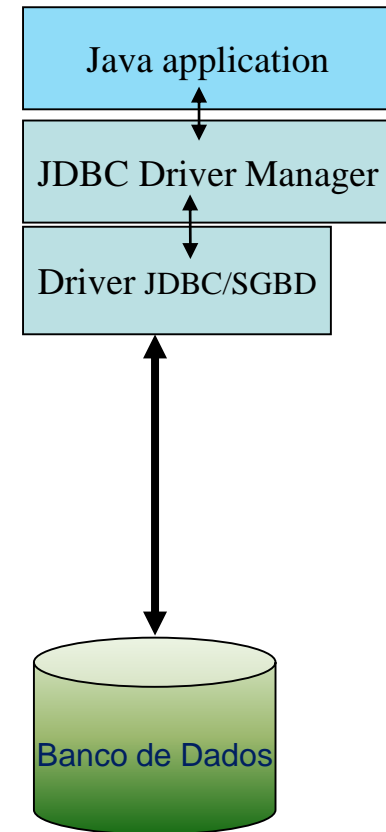
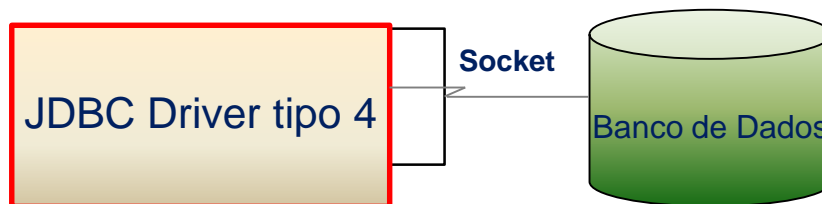




Driver JDBC -Tipo 4



- Drivers tipo 4 conversam diretamente ao Banco de Dados do fabricante.
- É uma solução puramente Java e emprega protocolo nativo do SGBD;





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 compõem 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>





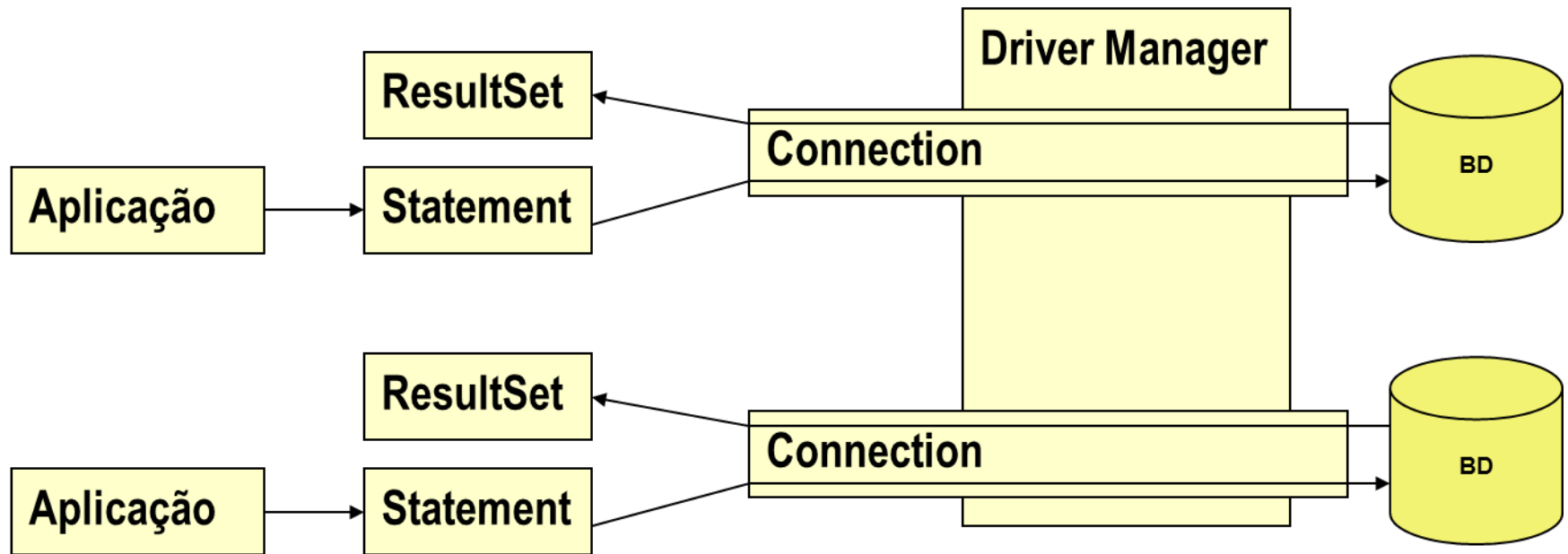
Classes Principais – JDBC

- ✓ **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`**

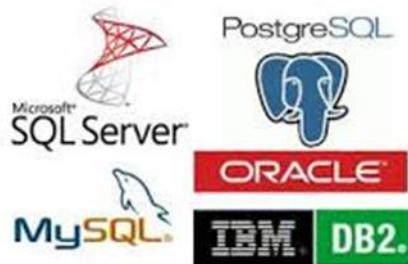
mySQL: **`com.mysql.jdbc.Driver`**

PostGresql: **`org.postgresql.Driver`**

Oracle: **`oracle.jdbc.driver.OracleDriver`**

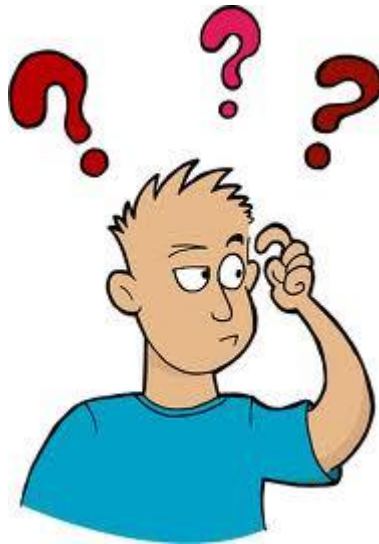
SqlServer: **`com.jnetdirect.jsql.JSQLDriver`**

DB2: **`com.ibm.db2.jdbc.app.DB2Driver`**





Como se registra o driver bridge JDBC-ODBC ?





Carga do JDBC-ODBC bridge driver

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```





Como se registra o driver nativo ORACLE ?





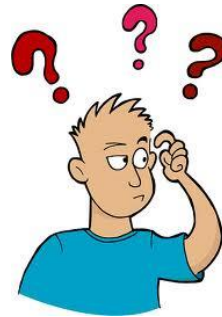
Carga do Oracle JDBC driver

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

ORACLE®



Como se registra o driver nativo MySQL?





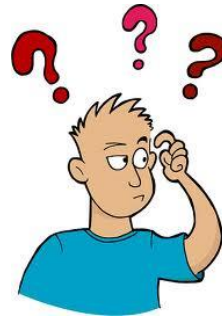
Carga do driver nativo MySQL

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





Como se registra o driver nativo DB/2?





Carga do IBM DB2 JDBC driver

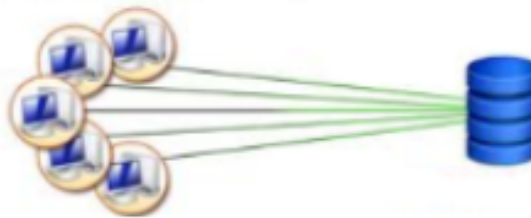
```
Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
```





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.





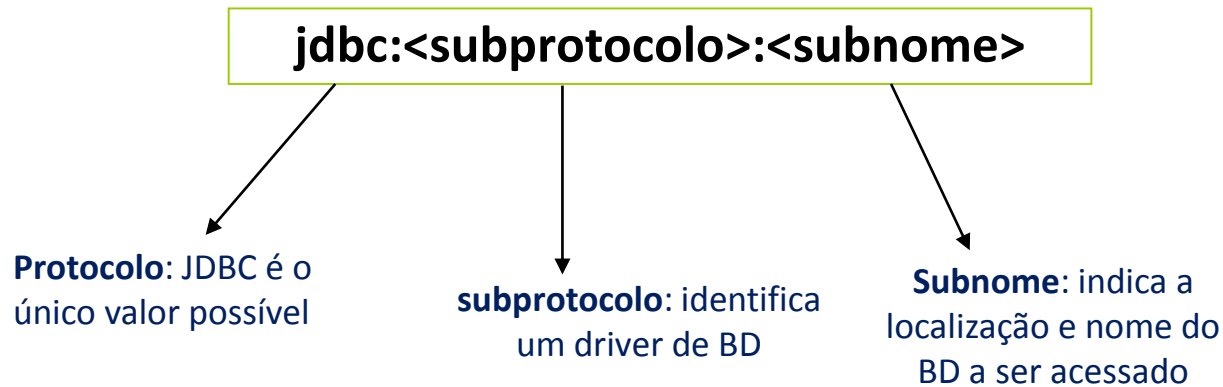
Como se especifica os parâmetros do método getConnection() ?





Parâmetros getConnection()

- Os parâmetros são URL, usuário e senha;
- A URL que emprega a seguinte sintaxe:

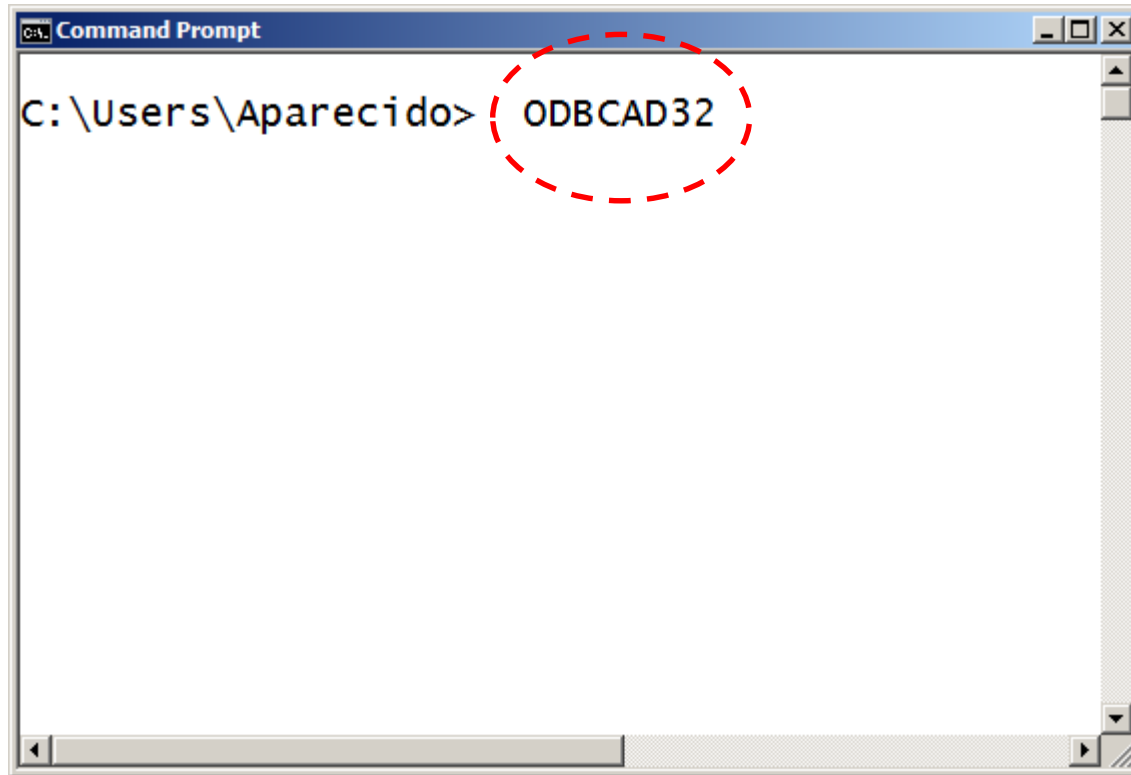




Exemplo – Configuração Bridge JDBC-ODBC



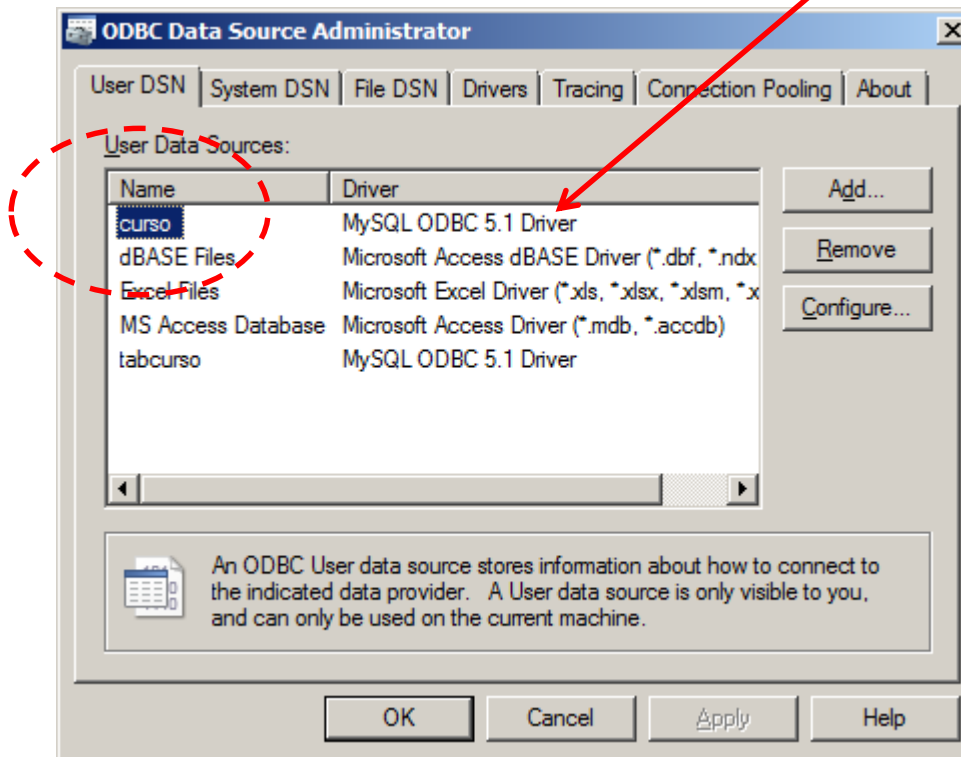
Configurando ODBC



Comando: ODBCAD32



Configurando ODBC





Exemplo getConnection() com Bridge JDBC-ODBC

- ODBC data source: **curso**
- DBMS login name: admin
- Password: secret
- Estabelecendo a conexão:

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



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 SGBD;
 - **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 executam 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

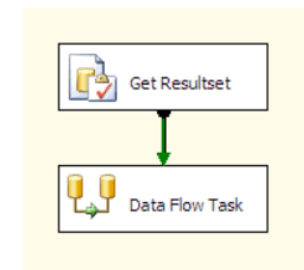
- Mantém o posicionamento do cursor em sua corrente linha de dados;
- Provê métodos para recuperar valores de colunas.

```
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.





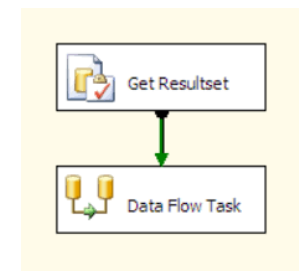
Manipulação de objetos ResultSet

✓ 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.





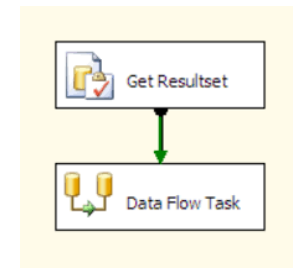
Manipulação de objetos ResultSet

✓ 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.





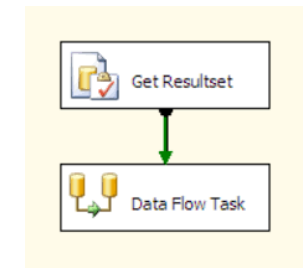
Acessores para tipos Java

- **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.





Exemplo de Consulta

```
import java.net.URL;
import java.sql.*;
import java.io.*;
public class Consulta{
    public static void main(String args[]) throws IOException{
        String comando="SELECT * FROM FONES" ;
        try{
            Connection con;
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con=DriverManager.getConnection
                ("jdbc:odbc:curso", "sa","");
            System.out.println("Conectado OK");
            Statement st = con.createStatement();
            ResultSet rs = st.executeQuery(comando);
```



Exemplo de Consulta

```
while (rs.next()) {  
    System.out.println("Nome: "+rs.getString(1)+" Fone:  
                        "+rs.getString(2));  
}  
st.close(); con.close();  
} catch(SQLException e){  
    System.out.println("Erro no SQL!");  
    return;  
} catch(ClassNotFoundException e){  
    System.out.println("Driver não Encontrada!");  
    return;  
}  
System.in.read();  
}  
}
```



Exemplo de Inserção

```
import java.net.URL;
import java.sql.*;
import java.io.*;
public class Consulta{
    public static void main(String args[]) throws IOException{
        String comando="INSERT INTO FONES
                        VALUES(“”+ args[0]+ “”,“”+args[1]+“”)”;
        try{
            Connection con;
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con=DriverManager.getConnection
                ("jdbc:odbc:curso", "sa","");
        }
    }
}
```




Exemplo de Inserção

```
System.out.println("Conectado OK");
Statement st = con.createStatement();
st.executeUpdate(comando);
System.out.println("INSERCAO OK");
st.close(); con.close();
} catch(SQLException e){
    System.out.println("Erro no SQL!");
    return;
} catch(ClassNotFoundException e){
    System.out.println("Driver não Encontrada!");
    return;
}
}
```



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 funcionarios = stat.executeQuery();
```

```
.....
```

```
// percorre os produtos
```

```
stat.setString(1, "Produtos");
```

```
ResultSet produtos = stat.executeQuery();
```

```
.....
```



Atividade 1 – JDBC

Escrever um **programa** desktop que faça uma conexão a um banco de dados e insira um registro. Utilizar a bridge para conexão JDBC/ODBC.

Acessar o Servidor de Banco de Dados MySQL.

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



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



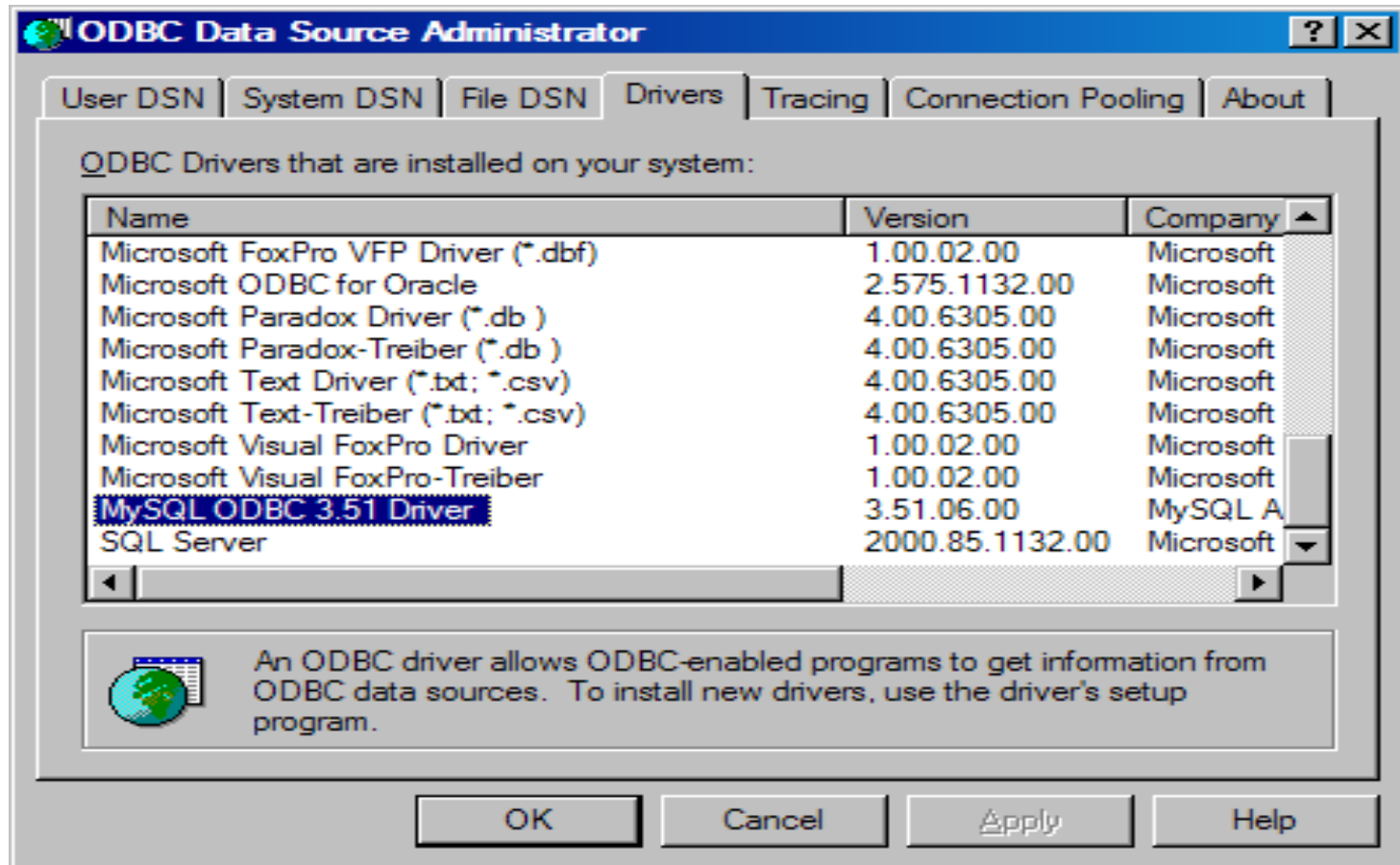
Criação do Banco de Dados – MySQL



```
mysql> create database curso;
mysql> use curso;
mysql> show tables;
mysql> create table tabcurso (
                codcurso      int(2) unsigned    zerofill not null,
                nomecurso      char(50),
                primary key (codcurso) ) ;

mysql> describe tabcurso;
mysql> show tables;
mysql> insert into tabcurso values(1,'Matematica');
mysql> select * from tabcurso;
```

Configuração ODBC => Comando **ODBCAD32** (Configurar Fonte de
Dados de Usuário)





MySQL ODBC 3.51 Driver - DSN Configuration, Version 3.51.06 [X]

This dialog helps you in configuring the ODBC Data Source Name, that you can use to connect to MySQL server

DSN Information

Data Source Name:

Description:

MySQL Connection Parameters

Host/Server Name(or IP):


Database Name:

User:

Password:

Port (if not 3306):

SQL command on connect:





```
package maua;
```

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



```
public class Atividade_01 { // Conexão com MySQL)
```

```
public static void main(String[] args) {  
    try {
```

```
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

```
        String url = "jdbc:odbc:curso";
```

```
        Connection con = DriverManager.getConnection(url, "root", null);
```

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

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

```
        String command = "INSERT INTO tabcurso VALUES(1,'Psicologia')";
```

```
        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);
```

```
}
```

```
}
```

```
}
```





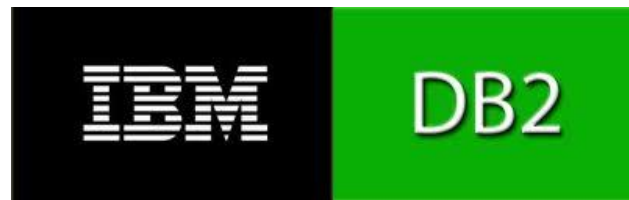
Atividade 2 – JDBC

Escrever um **programa** desktop que faça uma conexão a um banco de dados e insira um registro. Utilizar a bridge para conexão JDBC/ODBC.

Acessar o Servidor de Banco de Dados DB2 da plataforma IBM i.

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

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





package **maua**;

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



```
public class Atividade_2 { // Conexão DB2 – IBM i
```

```
public static void main(String[] args) {
```

```
try {
```

```
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

```
    String url = "jdbc:odbc:tabcurso";
```

```
    Connection con = DriverManager.getConnection(url, "db2maua", "maua");
```

```
    System.out.println("\nConexao no IBM i feita com sucesso...");
```

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

```
    String command = "INSERT INTO tabcurso VALUES(1,'Psicologia')";
```

```
    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);
```

```
}
```

```
}
```

```
}
```





Atividade 3 – 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

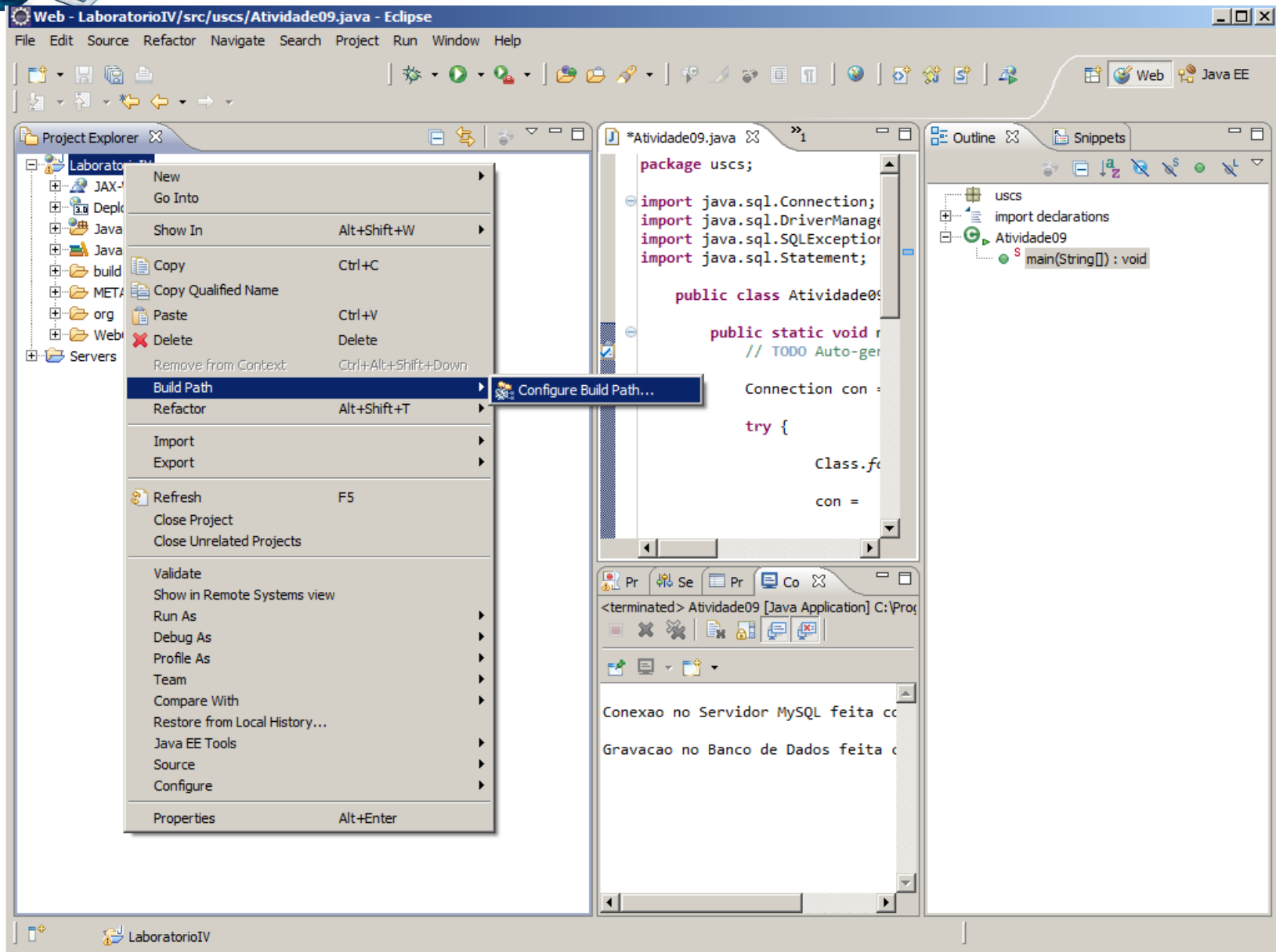
- ✓ Baixar o driver a partir do endereço:

<http://dev.mysql.com/downloads/connector/j/>

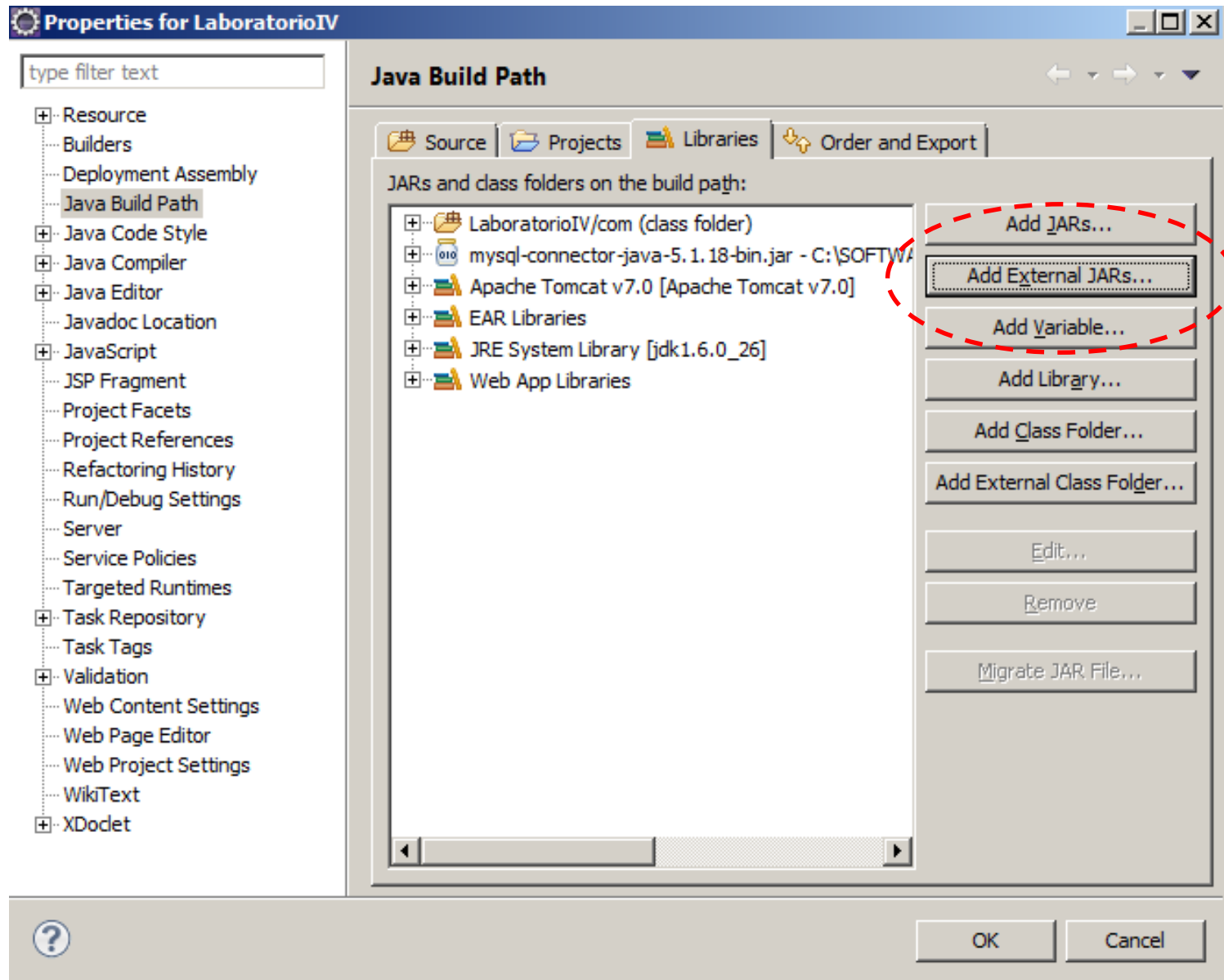
- ✓ Salvar em algum diretório do Servidor;
- ✓ Configurar o Path do Eclipse para que o projeto visualize o driver.



Configuração do Path – Eclipse



Configuração do Path – Eclipse





Parâmetros de Conexão

```
Connection con = null;
```

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

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





```
package maua;
```

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

```
public class Atividade_3 {
```

```
    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)

}

}

}

}
```

