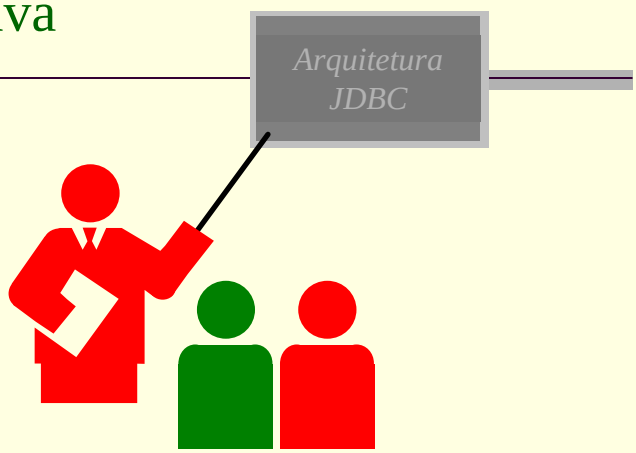
Módulo I
Interface com BancoDados JDBC

Prof. Ismael H F Santos

Ementa

- Modulo I – Interface com Banco Dados
  - Acesso a Bases de Dados com Java - Pacote JDBC
    - Arquitetura JDBC
    - Tipos de Drivers
    - Obtendo uma conexão – classe DriverManager
    - Criando comandos SQL
    - Enviando comandos SQL – classes
    - Manipulando resultados – classe ResultSet
  - Versões JDBC

POO-Java



Introdução

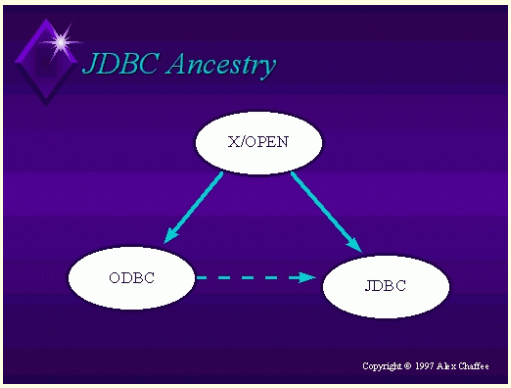
- Em 1996, a SUN lançou a primeira versão do kit JDBC.
- Esse kit possibilita ao programador de aplicações Java abrir conexões com um SGBD e consultar e modificar algum BD, utilizando a SQL.
- Baseado na abordagem da Microsoft para a sua API ODBC.
- Características:
  - Portabilidade
  - API independente do Banco de Dados subjacente
  - Estrutura em Camadas

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 (e futuramente OO)
- Interface baseada no X/OPEN SQL CLI
- Independente de API/Linguagem proprietária dos fabricantes de SGBD (Microsoft, Oracle, Informix, …)
- Uso de *drivers* específicos de fabricantes

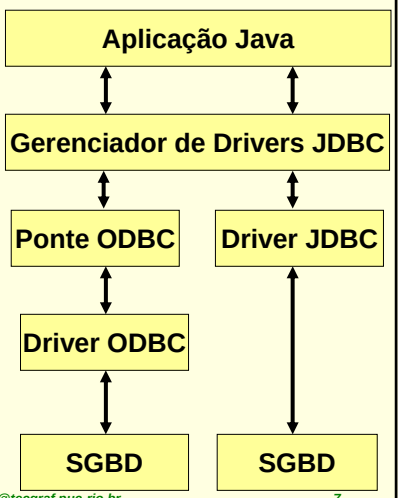
JDBC *versus* ODBC

- Padrão ODBC ≈ biblioteca C
- Problemas:
  - integração
  - segurança
  - portabilidade
- Solução:
  - ponte JDBC-ODBC

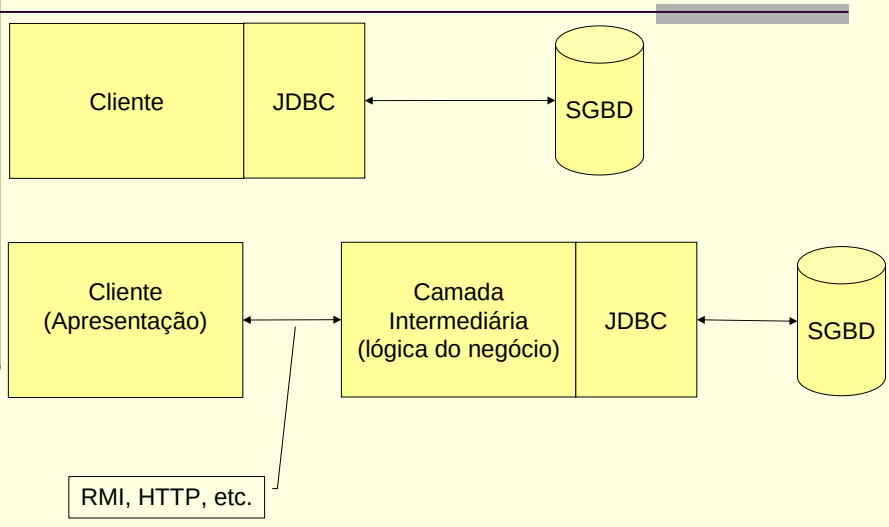


Arquitetura JDBC

- Aplicações Java “conversam” com o Gerenciador de Drivers JDBC (DriverManager)
- Este, por sua vez, se comunica com algum driver atualmente carregado.
- Programador se preocupa apenas com API do gerenciador de drivers.
- Drivers se ocupam da interface com o SGBD.
- Ponte JDBC pode ser usada se não existir um driver para um determinado SGBD.

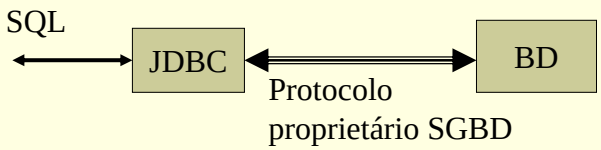


Arquiteturas de desenvolvimento e JDBC

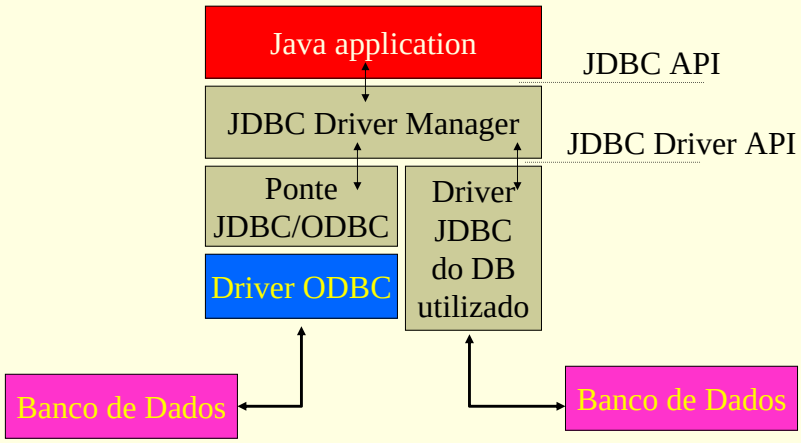


JDBC - O que faz

- Estabelece conexão com o Banco de Dados
- Envia requisições SQL
- Processa os resultados

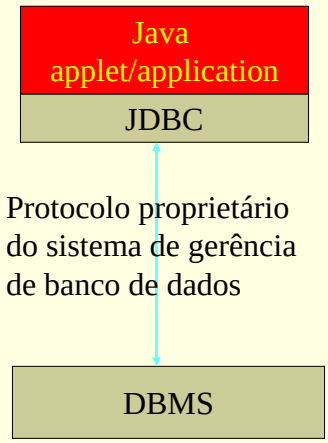


Caminho de Comunicação



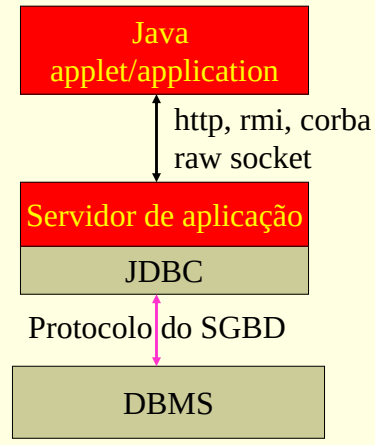
Modelo 2-camadas

- Aplicação se comunica diretamente com o BD
- Os comandos são enviados diretamente ao BD e as respostas ao cliente
- Utiliza o modelo Cliente/Servidor



Modelo 3-camadas

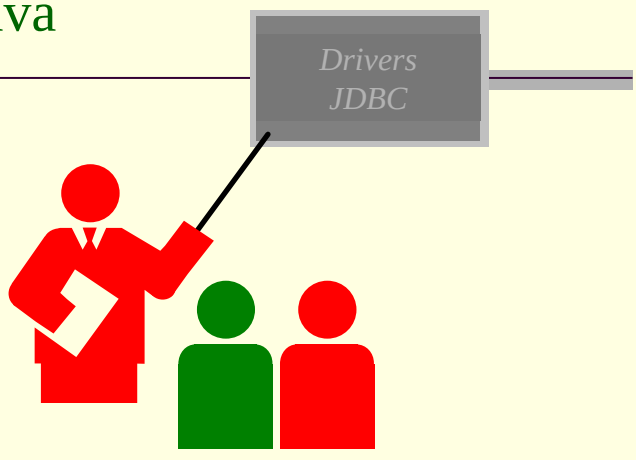
- Os comando são enviados para um intermediário (middle-tier)
- O intermediário faz a interface com BD
- Ganho de performance e de complexidade em longas distâncias



Padrão JDBC: passo-a-passo

1. Carga de um ou mais *drivers*
2. Obtenção de uma conexão
3. Criação de comandos SQL
4. Envio dos comandos
5. Obtenção de resultados

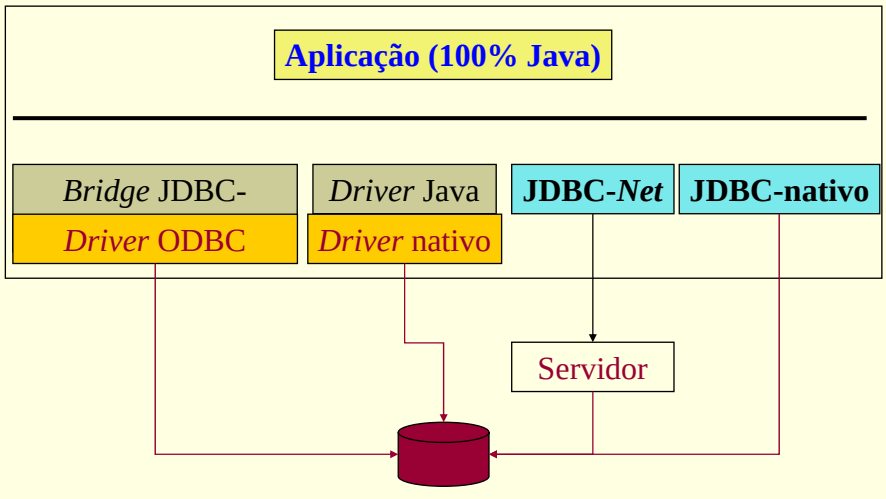
POO-Java



Tipos de *drivers*

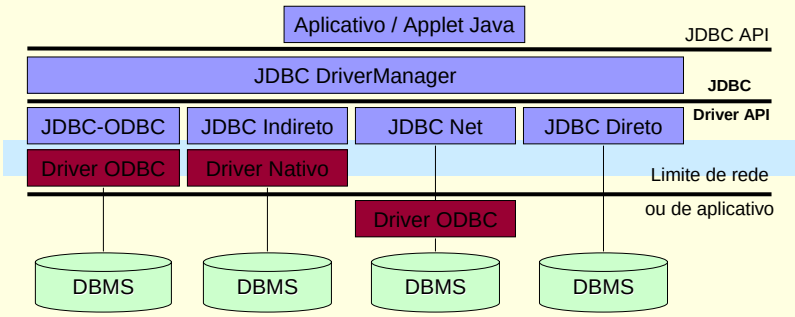
1. *Bridge* JDBC-ODBC + *Driver* ODBC
2. *Driver* Java + *Driver* da API nativa
3. *Driver* Java JDBC-*Net* + Servidor
4. *Driver* Java protocolo nativo

Funcionamento dos *drivers*



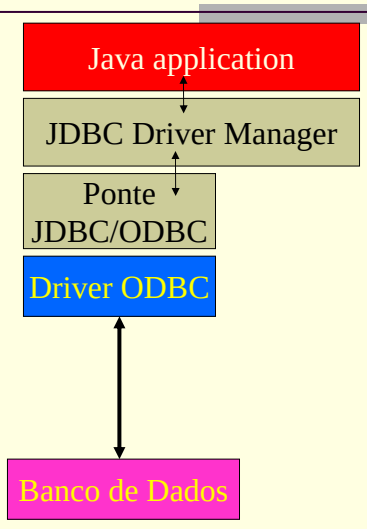
Funcionamento dos *drivers*

- Arquitetura



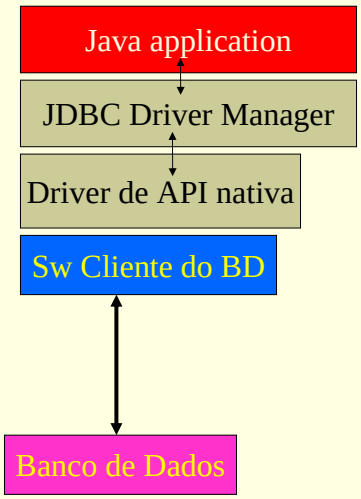
Tipos de Drivers

- Ponte JDBC-ODBC + driver ODBC
  - Driver ODBC deve estar em cada cliente



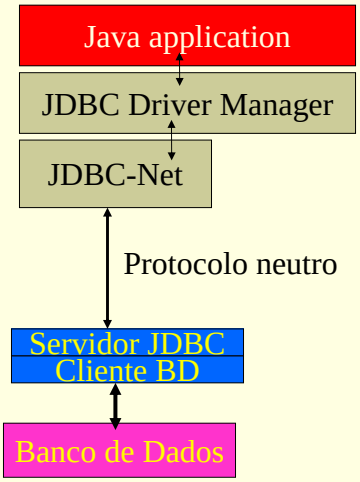
Tipos de Drivers

- API nativa
  - Driver que transforma chamadas JDBC em chamadas da API cliente de rede de um SGBD específico
  - Cada cliente deve possuir o driver instalado
  - Maior eficiência que a ponte ODBC



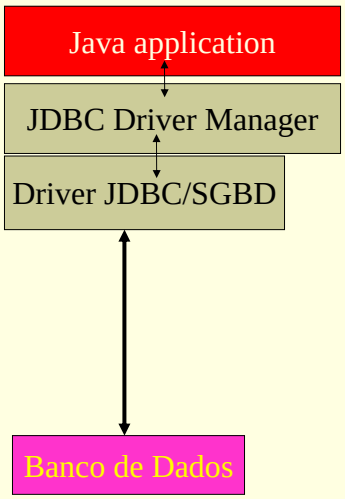
Tipos de Drivers

- JDBC-Net
  - Transforma as requisições JDBC em um protocolo de rede independente do SGBD. A conversão para o protocolo do SGBD é feita por um server.
  - Middleware para acesso a Banco de Dados
  - Solução flexível, que depende dos fabricantes colocarem driver JDBC em produtos middleware



Tipos de Drivers

- Protocolo Nativo- Java puro
  - Converte chamadas JDBC diretamente para o protocolo utilizado pelo SGBD.
  - Acesso direto do cliente ao BD
  - Fornecido pelo frabricante do SGBD
  - Apropriado para Intranet



Tipos de Drivers

Categoria	*100% Java*	*Comunicação com SGBD*
Ponte JDBC-ODBC	Não	Direto
API Nativa	Não	Direto
JDBC-Net	Sim	Necessita conector (servidor)
Protocolo Nativo	Sim	Direto

Drivers

- Implementam a interface **Driver**
- Como descrito anteriormente, implementam o padrão JDBC para uma base específica
- Todo *driver*, ao ser carregado, se cadastra junto ao **DriverManager** através de um inicializador estático
  - basta carregar a classe que o *driver* estará disponível para uso

Carregando um *driver*

- Todos os *drivers* que sejam listados na propriedade **jdbc.drivers** são carregados automaticamente
- Exemplo:

```
java –Djdbc.drivers=org.postgresql.Driver Teste
```

Outra forma...

- Como basta carregar a classe do *driver*, podemos fazer isso explicitamente
- Exemplo:

```
void carregaDrivers() throws ClassNotFoundException {
  Class.forName("org.postgresql.Driver");
  ...
}
```


Métodos de **Driver**

```
boolean acceptsURL(String url)
   throws SQLException
Connection connect(String url, Properties info) throws
   SQLException

int getMajorVersion()
int getMinorVersion()
boolean jdbcCompliant()
```

Implementações do 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.

Instalando JDBC

- O pacote JDBC vêm incluso com as distribuições Java
  - As classes que compões o kit JDBC estão nos pacotes java.sql e javax.sql.
- Entretantdo, deve-se obter um driver para o sistema de gerência de banco de dados a ser utilizado.
- O URL a seguir fornece uma lista de drivers JDBC atualmente disponíveis:
  - http://industry.java.sun.com/products/jdbc/drivers

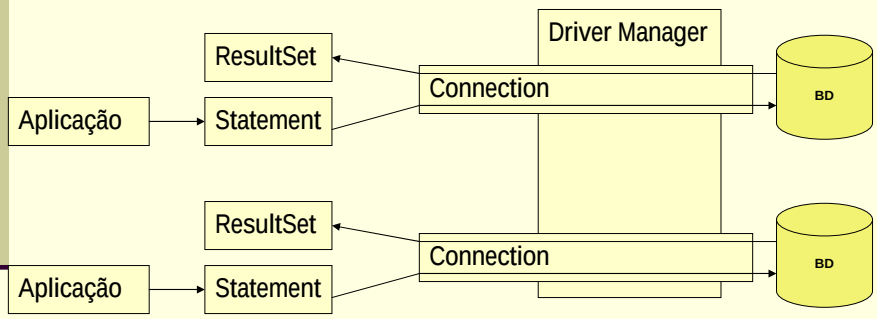
Classes principais do JDBC

- 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)

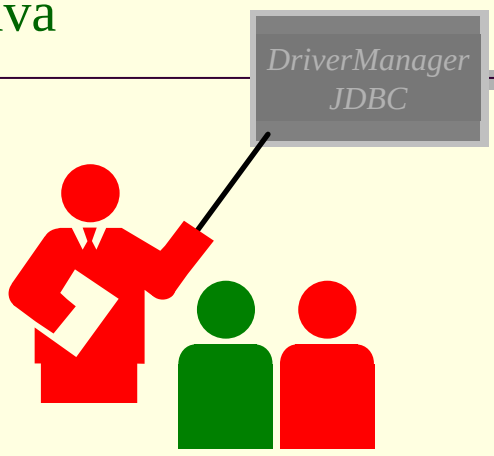
Classes principais do JDBC

- 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
    - Exemplo: tipo no banco DateTime, tipo de retorno String
  - Colunas de um objeto ResultSet podem ser referenciadas por um número posicional ou pelo nome da coluna do resultado.
    - ResultSet rs.getString("Nome") ou ResultSet rs.getString(1)

Classes principais do JDBC



POO-Java



Classe **DriverManager**

- É responsável por abrir uma conexão, especificada através de uma URL, com uma base de dados, utilizando o *driver* correto
- Possui um registro de todos os *drivers* já carregados
- Para abrir uma conexão, pergunta a cada *driver* se ele consegue abrí-la ou não, até encontrar um que consiga

Métodos de **DriverManager**

```
public static synchronized Connection getConnection( String
   url)throws SQLException

public static synchronized Connection getConnection( String
   url, Properties info)throws SQLException

public static synchronized Connection getConnection( String
   url, String user, String password)throws SQLException
```

Registrando um driver

- Para registrar um driver para um SGBD específico, utiliza-se o método estático Class.forName
  - Através desse método, é possível especificar qual o driver a ser utilizado.
  - Class.forName(nome-da-classe-do-driver);
  - O nome do driver consta na documentação do mesmo
- O argumento para o método forName especifica o driver a ser registrado.

Registrando um driver

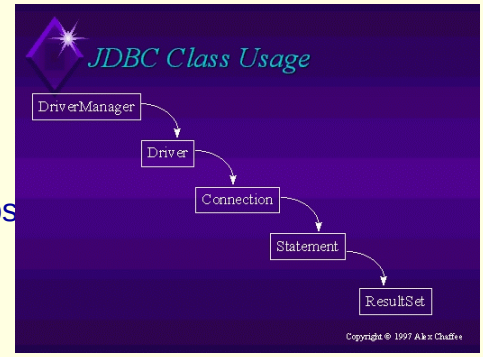
- Exemplos:
  - JDBC-ODBC: sun.jdbc.odbc.JdbcOdbcDriver
  - mySQL: com.mysql.jdbc.Driver
  - PostGresql: org.postgresql.Driver
  - Oracle: oracle.jdbc.driver.OracleDriver
  - SqlServer: com.jnetdirect.jsql.JSQLDriver
  - DB2: com.ibm.db2.jdbc.app.DB2Driver
- É possível registrar vários drivers em uma mesma aplicação.
  - O Driver Manager gerencia cada um deles.

Registrando um driver (cont.)

- O nome completo da classe correspondente ao driver a ser utilizado deve ser especificado no classpath.
- Alternativas:
  - Executar o interpretador (java) com a opção –classpath na linha de comando
    - java –classpath .;DriverPath programa.java
  - Modificar a variável de ambiente CLASSPATH.
    - set CLASSPATH=.;DriverPath

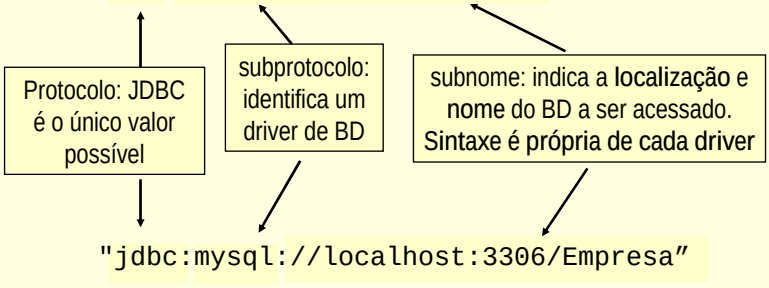
Padrão JDBC: passo-a-passo

1. ✓ Carga de um ou mais *drivers*
2. Obtenção de uma conexão
3. Criação de comandos SQL
4. Envio dos comandos
5. Obtenção de resultados



Conectando a um BD

- O argumento para o método getConnection tem um formato similar ao de uma URL
  - jdbc:<subprotocolo>:<subnome>



Conectando a um BD

- Para estabelecer uma conexão com um BD, use o método getConnection da classe DriverManager
  - DriverManager.getConnection(url, usuario, senha);
- Exemplos de URLs
  - jdbc:odbc:MinhaBase
  - jdbc:odbc:MinhaBase;UID=cassino;PWD=my_pass
  - jdbc:mysql://serverName/mydatabase?user=x&password=y
  - jdbc:postgresql://serverName/mydatabase
  - jdbc:oracle:thin:@serverName:portNumber:mydatabase
  - jdbc:JSQLConnect://serverName:portNumber/mydatabase

Abrindo uma conexão

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conn = DriverManager.getConnection(
   "jdbc:odbc:usuarios_db","es003","xpto");

Class.forName("org.gjt.mm.mysql.Driver");
Connection conn = DriverManager.getConnection(
   "jdbc:mysql://localhost/usuarios_db“,user,passwd);

Class.forName("org.postgresql.Driver");
Connection conn = DriverManager.getConnection(
   "jdbc:postgresql:usuarios_db“,user,passwd);

Class.forName("oracle.jdbc.driver.OracleDriver");
Connection conn = DriverManager.getConnection(
 "jdbc:oracle:thin:@127.0.0.1:1521:usuarios_db",
 user,passwd);
```


Conectando a um BD (cont.)

- Exemplo:

```
try {
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  Connection com = DriverManager.getConnection(
                   “jdbc:odbc:Northwind”,””,””);
  ...// Código de manipulação do BD
}
catch (ClassNotFoundException e) {
  System.out.println(“Classe não Encontrada!”);
}
catch (SQLException e) {
  System.out.println(“Erro na Conexão!”);
}
```

Outros métodos de **DriverManager**

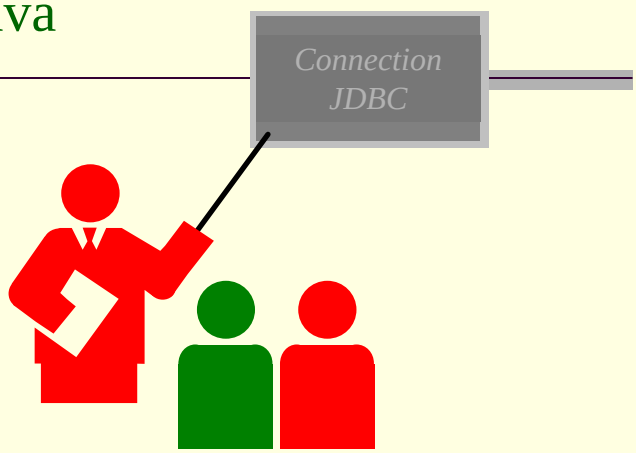
```
public static Driver getDriver(String url) throws
   SQLException
public static synchronized void registerDriver(Driver
   driver) throws SQLException
public static void deregisterDriver(Driver driver) throws
   SQLException
public static Enumeration getDrivers()

public static void setLoginTimeout(int seconds)
public static void setLogWriter(PrintWriter out)
public static void println(String message2log)
```


Padrão JDBC: passo-a-passo

- ✓ Carga de um ou mais *drivers*
- ✓ Obtenção de uma conexão
3. Criação de comandos SQL
4. Envio dos comandos
5. Obtenção de resultados

POO-Java



Conexões

- Implementam a interface **Connection**
- A partir de uma conexão, podemos:
  - criar comandos SQL (de diferentes formas)
  - configurar características da conexão, como:
    - controle de transações
    - registro de uso *read-only*

Métodos de **Connection** para criação de comandos

```
Statement createStatement()
   throws SQLException

PreparedStatement prepareStatement(String sql)
   throws SQLException

CallableStatement prepareCall(String sql)
   throws SQLException
```


Criando comandos SQL

```
Class.forName("org.postgresql.Driver");
Connection conn = DriverManager.getConnection(
   "jdbc:postgresql:usuarios");
Statement stat = conn.createStatement();
// stat pode ser utilizado para enviar comandos
// SQL à base de dados
```

Métodos de **Connection** para controle de transações

```
void setAutoCommit(boolean ac) throws SQLException

void commit() throws SQLException

void rollback() throws SQLException
```


Outros métodos de **Connection**

```
void setReadOnly(boolean ro) throws SQLException
boolean isReadOnly() throws SQLException
SQLWarning getWarnings() throws SQLException
void clearWarnings() throws SQLException

void close() throws SQLException
boolean isClosed() throws SQLException
```

Padrão JDBC: passo-a-passo

- ✓ Carga de um ou mais *drivers*
- ✓ Obtenção de uma conexão
- ✓ Criação de comandos SQL
4. Envio dos comandos
5. Obtenção de resultados

POO-Java

Tipos de comandos SQL

- **Statement**
  - envia texto SQL ao servidor
- **PreparedStatement**
  - pré-compila o texto SQL
  - posterior envio ao servidor
- **CallableStatement**
  - similar ao **PreparedStatement**
  - permite executar procedimentos SQL

Comandos diretos

- Comandos SQL podem ser diretamente enviados à base através de um objeto que implemente a interface **Statement**
- Comandos de definição de dados (DDL), de atualização da base e de consulta são aceitos
- O *driver* é responsável por traduzir o SQL para a sintaxe própria da base (note que o **Statement** é parte do *driver*!)

Statements

- Um objeto da classe 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.

Métodos de Statement

```
ResultSet executeQuery(String sql)
   throws SQLException
int executeUpdate(String sql) throws SQLException

boolean execute(String sql) throws SQLException
ResultSet getResultSet() throws SQLException
int getUpdateCount() throws SQLException
boolean getMoreResults() throws SQLException
```

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 e não retornem dados
  - Forma geral: executeUpdate(<comando>)
  - Ex: stmt.executeUpdate(“DELETE FROM Cliente”);
  - Esse método retorna um inteiro: quantas linhas foram atingidas.

Executando statements

- Consultas: executeQuery
  - Para comandos ”SELECT” ou outros que retornem dados
  - Forma geral: stmt.executeQuery(<comando>);
  - Esse método retorna um objeto da classe ResultSet
  - Ex: rs = stmt.executeQuery(“SELECT * FROM Cliente”);

Exemplo de **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");
```


Statement

- Permite o uso de escapes
- Exemplo:

```
{fn user()}
{d 'yyyy-mm-dd'}
{call procedure_name[(?, ?, . . .)]}
{? = call procedure_name[(?, ?, . . .)]}
```

Equivalência de Tipos JavaxSQL

Tipo SQL	Métodos para Recuperar Dados	Tipo Java
CHAR	getString()	String
VARCHAR	getString()	String
LONGCHAR	InputStream getAsciiStream()	String
INTEGER	getInt()	int
FLOAT	getFloat()	float
DOUBLE	getDouble()	double
BIGINT	getLong()	long
DATE	getDate()	java.sql.Date
TIME	getTime()	java.sql.Time
BIT	getBoolean()	boolean
TINYINT	getByte()	byte
TIMESTAMP	getTimestamp()	java.sql.Timestamp

Manipulando um objeto 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.

Manipulando um objeto ResultSet

- Métodos first(), last()
  - Posiciona o cursor no início ou no final do conjunto de dados.
- Métodos isFirst(), isLast()
  - Testa posição do cursor; retorna valor lógico.

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("com.ashna.jturbo.driver.Driver");
      con=DriverManager.getConnection
                ("jdbc:JTurbo://rubi/javadb", "sa","");
      System.out.println("Conectado OK");
      Statement st = con.createStatement();
      ResultSet rs = st.executeQuery(comando);
```

Exemplo de consulta (cont.)

```
 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 Insere {
  public static void main(String args[]) throws IOException{
    String fonte="jdbc:odbc:javadb";
    String comando="INSERT INTO FONES
                              VALUES('"+args[0]+"','"+args[1]+"')" ;
    System.out.println(comando+"\n");
    try {
      Connection con;
      Class.forName("com.ashna.jturbo.driver.Driver");
      con=DriverManager.getConnection
                        ("jdbc:JTurbo://rubi/javadb", "sa","");
```

Exemplo de inserção (cont.)

```
        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;
      }
    }
}
```


Comandos preparados

- É possível pré-compilar um comando SQL que precise ser executado repetidas vezes
- Tais comandos podem, inclusive, conter parâmetros a serem especificados no momento da execução
- Estes comandos pré-compilados são modelados pela interface **PreparedStatement**

Mais comandos preparados

- O fato do comando ser pré-compilado pode aumentar o desempenho em execuções sucessivas
- Para criarmos um **PreparedStatement**, utilizamos o método **prepareStatement** da conexão (**Connection**)
- **PreparedStatement** é sub-tipo de **Statement**

Métodos de **PreparedStatement**

```
ResultSet executeQuery() throws SQLException
int executeUpdate() throws SQLException

void setBoolean(int parameterIndex, boolean x) throws
  SQLException
void setByte(int parameterIndex, byte x) throws
  SQLException
void setAsciiStream(int parameterIndex, InputStream x,
  int length) throws SQLException
...
```

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.

PreparedStatement

- Em um comando SQL de um objeto PreparedStatement:
  - Parâmetros são simbolizados por pontos de interrogação.
  - Configuração dos valores dos parâmetros: métodos setXXX
- Exemplo1:

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

Exemplo de **PreparedStatement**

- Exemplo2:

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


Acesso a *stored procedures*

- Existe um comando especial, modelado pela interface **CallableStatement**, que facilita a chamada de *stored procedures*
- Um **CallableStatement** é um sub-tipo de **PreparedStatement** que, além de permitir a passagem de parâmetros, também permite o retorno de valores

Tipos dos valores

- Para obtermos um valor de uma *stored procedure*, devemos especificar o tipo deste valor
- A classe **Types** enumera todos os tipos SQL aceitos no padrão JDBC, conhecidos como “tipos JDBC”

Métodos de **CallableStatement**

```
public abstract void registerOutParameter(int
  parameterIndex, int sqlType) throws SQLException

public abstract boolean getBoolean(int
  parameterIndex) throws SQLException
public abstract byte getByte(int parameterIndex)
  throws SQLException
...
```

Exemplo de CallableStatement

```
CallableStatement call = conn.prepareCall(
   "{call getMinMaxIds(?,?)}");
call.registerOutParameter(1, java.sql.Types.INTEGER);
call.registerOutParameter(2, java.sql.Types.INTEGER);
call.executeQuery();
int min = call.getInt(1);
int max = call.getInt(2);
```


CallableStatement

- Objetos da classe CallableStatement
  - Usados para chamar Stored Procedures
  - Sintaxe abstrata
    - callableStatement cst=con.prepareCall("{call nome_da_SP>}");
  - Métodos setXXX, como nos objetos PreparedStatement
  - Podem receber parâmetros devolvidos pela StoredProcedure

CallableStatement

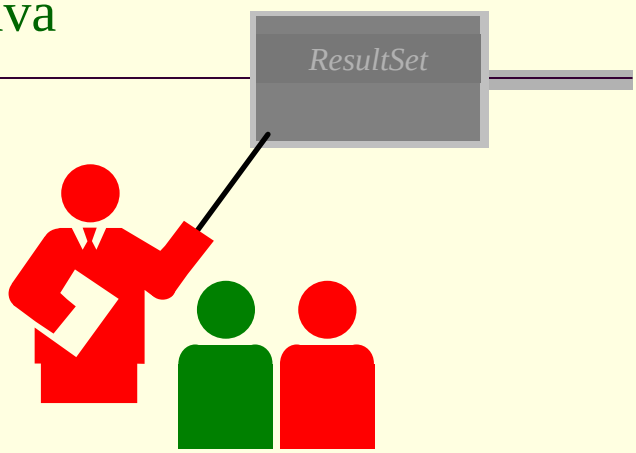
- Exemplo

```
CallableStatement csmt;
csmt = conexao.prepareCall("{call sp_interest(?,?)}");
csmt = conexao.registerOutParameter(2, Types.FLOAT);
csmt.execute();
float resultado=csmt.getFloat(2);
```


Padrão JDBC: passo-a-passo

1. ✓ Carga de um ou mais *drivers*
2. ✓ Obtenção de uma conexão
3. ✓ Criação de comandos SQL
4. ✓ Envio dos comandos
5. Obtenção de resultados

POO-Java



Resultados de pesquisas

- Representados por um **ResultSet**
- Linhas são acessadas em sequência
- Colunas são acessadas aleatoriamente
- São invalidados quando seu **Statement** for:
  - fechado
  - re-executado
  - usado para obter o próximo resultado de uma série

Métodos de **ResultSet**

```
public abstract boolean next() throws SQLException
public abstract void close() throws SQLException
public abstract boolean getBoolean(int columnIndex)
  throws SQLException
public abstract boolean getBoolean(String columnName)
  throws SQLException
...
public abstract InputStream getAsciiStream(int
  columnIndex) throws SQLException
public abstract int findColumn(String columnName)
  throws SQLException
```


Exemplo de **ResultSet**

```
ResultSet res = stat.executeQuery("SELECT Nome FROM
  Funcionarios");
while (res.next()) {
  System.out.println(res.getString(1));
}
```

Padrão JDBC: passo-a-passo

- ✓ Carga de um ou mais *drivers*
- ✓ Obtenção de uma conexão
- ✓ Criação de comandos SQL
- ✓ Envio dos comandos
- ✓ Obtenção de resultados

Conversões de tipos

- Um problema que não foi abordado até aqui é a necessidade de conversão entre os tipos de SQL e os de Java
- Como dissemos, os tipos SQL (ou tipos JDBC) são enumerados na classe **Types**
- As possíveis conversões entre estes tipos e os de Java são especificados pelo padrão

Exemplos

- Criar uma tabela
- Popular a tabela
- Listar a tabela
- Pesquisar na tabela
- Alterar a tabela
- Criar uma *stored procedure*
- Chamar uma *stored procedure*
- Controlar uma transação

Tabela “senhas”

- Todos os exemplos se referem à tabela “senhas”, cujo conteúdo é:

Coluna 1	Coluna 2
nome	senha

Criar uma tabela

```
import java.sql.*;

class CriaTabela {
  public static void main(String[] args) throws Exception {
    Class.forName("org.postgresql.Driver");
    String url = "jdbc:postgresql:teste";
    String user = "usuario";
    String pass = "senha";
    Connection db = DriverManager.getConnection(url,user,pass);
    Statement st = db.createStatement();
    int ok = st.executeUpdate("create table senhas ("+
                              "  nome varchar(80) not null unique,"+
                              "  senha varchar(16) not null"+
                              ")");
    System.out.println("ok.");
    st.close();
    db.close();
  }
}
```


Popular a tabela

```
String[][] registros = {{"joao","joao"},{"maria","maria"}};
Connection db = DriverManager.getConnection(url,user,pass);
Statement st = db.createStatement();
for (int i=0; i<registros.length; i++) {
  String nome = registros[i][0];
  String senha = registros[i][1];
  int ok = st.executeUpdate("insert into senhas "+
                            "(nome, senha) values ('"+
                             nome+"','"+senha+"')");
  System.out.println(nome+(ok==1 ? ": ok." : ": nok."));
}
st.close();
db.close();
```

Popular a tabela (2)

```
String[][] registros = {{"joao","joao"},{"maria","maria"}};
Connection db = DriverManager.getConnection(url,user,pass);
PreparedStatement ps = db.prepareStatement("insert into senhas "+
                                           "values(?,?)");
for (int i=0; i<registros.length; i++) {
  String nome = registros[i][0];
  String senha = registros[i][1];
  ps.setString(1, nome);
  ps.setString(2, senha);
  int ok = ps.executeUpdate();
  System.out.println(nome+(ok==1 ? ": ok." : ": nok."));
}
ps.close();
db.close();
```


Listar a tabela

```
Connection db = DriverManager.getConnection(url,user,pass);
Statement st = db.createStatement();
ResultSet rs = st.executeQuery("select * from senhas");
while (rs.next()) {
  System.out.println(rs.getString(1)+" "+rs.getString(2));
}
rs.close();
st.close();
db.close();
```

Pesquisar na tabela

```
String[][] registros = {{"joao","jj"},{"maria","maria"}};
Connection db = DriverManager.getConnection(url,user,pass);
Statement st = db.createStatement();
for (int i=0; i<registros.length; i++) {
  String nome = registros[i][0];
  String senha = registros[i][1];
  ResultSet rs = st.executeQuery("select nome from senhas"+
                                 "where nome='"+nome+"' and "+
                                 "senha='"+senha+"'");
  System.out.println(nome+(rs.next() ? ": ok." : ": nok."));
  rs.close();
}
st.close();
db.close();
```


Pesquisar na tabela (2)

```
String[][] registros = {{"joao","jj"},{"maria","maria"}};
Connection db = DriverManager.getConnection(url,user,pass);
PreparedStatement ps = db.prepareStatement("select nome from "+
                                           "senhas where nome=? "+
                                           "and senha=?");
for (int i=0; i<registros.length; i++) {
  String nome = registros[i][0];
  String senha = registros[i][1];
  ps.setString(1, nome);
  ps.setString(2, senha);
  ResultSet rs = ps.executeQuery();
  System.out.println(nome+(rs.next() ? ": ok." : ": nok."));
  rs.close();
}
ps.close();
db.close();
```

Alterar a tabela

```
String[][] registros = {{"maria","mary"},{"joao","jj"}};
Connection db = DriverManager.getConnection(url,user,pass);
Statement st = db.createStatement();
for (int i=0; i<registros.length; i++) {
  String nome = registros[i][0];
  String senha = registros[i][1];
  int ok = st.executeUpdate("update senhas set senha='"+senha+
                            "' where nome='"+nome+"'");
  System.out.println(nome+(ok==1 ? ": ok." : ": nok."));
}
st.close();
db.close();
```


Alterar a tabela (2)

```
String[][] registros = {{"maria","mary"},{"joao","jj"}};
Connection db = DriverManager.getConnection(url,user,pass);
PreparedStatement ps = db.prepareStatement("update senhas set "+
                                           "senha=? where "+
                                           "nome=?");
for (int i=0; i<registros.length; i++) {
  String nome = registros[i][0];
  String senha = registros[i][1];
  ps.setString(2, nome);
  ps.setString(1, senha);
  int ok = ps.executeUpdate();
  System.out.println(nome+(ok==1 ? ": ok." : ": nok."));
}
ps.close();
db.close();
```

Criar uma *stored procedure*

```
Connection db = DriverManager.getConnection(url,user,pass);
Statement st = db.createStatement();
int ok = st.executeUpdate("create function "+
                          "verifica (varchar,varchar) "+
                          "returns boolean as "+
                          "'select count(nome)=1 from senhas "+
                          " where nome = $1 and senha = $2;' "+
                          "language 'sql'");
System.out.println("ok.");
st.close();
db.close();
```


Chamar uma *stored procedure*

```
String[][] registros = {{"joao","jj"},{"maria","maria"}};
Connection db = DriverManager.getConnection(url,user,pass);
CallableStatement cs = db.prepareCall("{? = call verifica(?,?)}");
cs.registerOutParameter(1, Types.BIT);
for (int i=0; i<registros.length; i++) {
  String nome = registros[i][0];
  String senha = registros[i][1];
  cs.setString(2, nome);
  cs.setString(3, senha);
  ok = cs.executeUpdate();
  System.out.println(ok==1 ? "ok." : "nok.");
  boolean b = cs.getBoolean(1);
  System.out.println(nome+(b ? ": ok." : ": nok."));
}
cs.close();
db.close();
```

Versão alternativa

```
String[][] registros = {{"joao","jj"},{"maria","maria"}};
Connection db = DriverManager.getConnection(url,user,pass);
PreparedStatement os =db.prepareStatement("select verifica(?,?)");
for (int i=0; i<registros.length; i++) {
  String nome = registros[i][0];
  String senha = registros[i][1];
  ps.setString(1, nome);
  ps.setString(2, senha);
  ResultSet rs = ps.executeQuery();
  if (!rs.next())
    System.out.println("nok.");
  else
    System.out.println(nome+(rs.getBoolean(1)?": ok." : ": nok."));
  rs.close();
}
ps.close(); db.close();
```


Transações

- Modo padrão: Compromissamento automático (auto_commit)
- Pode ser mudado para compromissamento manual
- método setAutoCommit(<true|false>), da classe Connection
- Métodos commit e rollback na conexão para compromissamento manual

Transações (Exemplo)

```
preparedStatement pst;
pst=con.prepareStatement("insert into fones (nome,fone) values (?,?)");
callableStatement cst;
cst=con.prepareCall("{call insert_endereco}");
con.setAutoCommit(false);
try {
    pst.setString(1,"Carlos");
    pst.setString(2,"222-2222");
    cst.setString(1,"Futino");
    cst.setString(2,"Rua das Casas, 222");
    con.commit();
}
catch (SQLException e) {
    con.roolback();
    System.out.println("Erro: transação abortada");
}
```


Controlar uma transação

```
Connection db = DriverManager.getConnection(url,user,pass);
db.setAutoCommit(false);
Statement st = db.createStatement();
int ok = st.executeUpdate("update senhas set senha='x' "+
                  "where nome='joao'");
System.out.println(ok==1 ? "ok." : "nok.");
db.rollback(); // ou db.commit();
db.setAutoCommit(true);
ResultSet rs = st.executeQuery("select * from senhas");
while (rs.next()) {
  System.out.println(rs.getString(1)+" "+rs.getString(2));
}
rs.close();  st.close();  db.close();
```

Conclusões

- Curiosidade: alguns SGBD estão implementando a JVM internamente a seus produtos, de tal forma que se possa construir procedimentos armazenados em Java.
  - Essa tecnologia é parte da especificação SQLJ.
  - Para mais detalhes sobre SQLJ: http://www.sqlj.org

Conclusões

- Originalmente, o kit JDBC foi bastante utilizado em aplicações cliente/servidor.
- Contudo, o mundo da computação está se afastando cada vez mais da arquitetura cliente/servidor e se aproximando da arquitetura em três camadas.
- Vide próximo slide...

Passos para programar com JDBC

- Importar java.sql.*
- Carregar driver JDBC
- Especificar BD
- Abrir conexão com BD
- Criar um objeto comando (statement)
- Submeter o comando SQL
- Receber o resultado
- Utilizar os resultados no programa

Anatomia de um aplicativo JDBC

```
// Conexão com BD
Connection com=DriverManager.getConnection(
     “jdbc:odbc:bd”,”user”,”passwd”);
Statement stmt= com.createStatement();
// Envia SQL query
ResultSet rs=smtm.executeQuery(
               “SELECT a,b c FROM Table”);
// Manipula resultados
while (rs.next()){
     int x=rs.getInt(“a”);
     String s= rs.getString(“b”);
     float f= rs.getFloat(“c”);
}
```

API - Carga do driver JDBC

- `Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");`
- `new sun.jdbc.odbc.JdbcOdbcDriver`
- `java -Djdbc.drivers=sun.jdbc.JdbcOdbcDriver <pgm>`

Especificar o banco de Dados

- Através de um string no formato *URL*
  jdbc:<subprotocol>:<fonte_dos_dados>
- Ponte jdbc-odbc
  jdbc:odbc:arquivo.mdb
- mSQL
  jdbc:msql://server.unifil.br:5344/BD
- Oracle thin (Puro Java)
  jdbc:oracle:thin:@server.unifil.br:2345/BD

API JDBC

- Métodos para Conectar ao banco de dados
  - getConnection(*url*,"user","senha")
- Métodos para preparar SQL
  - Statements
- Métodos de acesso ao BD
  - executeQuery()
  - executeUpdate()
- Tratamento dos Resultados
  - resultSet()

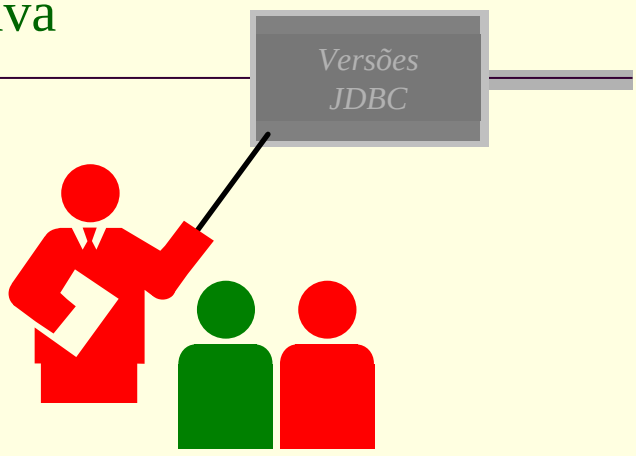
Tipos de *Statements*

- Statements
  - SQL que pode ser executada imediatamente
- PreparatedStatement
  - Instrução compilada
  - Aceita parâmetros variáveis
  - No momento da compilação pode otimizar a procura
- CallableStatement
  - Procedimento armazenado

Exemplo - Livraria Virtual

- Banco de dados de uma livraria
  - arquivo mdb (access) com autor e título
- Java application para consulta no BD
- Exclusão mútua provida pelo SGBD (no access não existe conceito de multiusuário)
- BookStore.java
- Configurar ponte ODBC, Compilar, Executar

POO-Java



Versões do padrão JDBC

JDBC 1.0

- Além do exposto até aqui, o padrão JDBC em sua versão 1.0 define meta-dados do banco e dos resultados das consultas, através das interfaces DatabaseMetaData e ResultSetMetaData

JDBC 2.0

- Define a extensão padrão **javax.sql** que:
  - cria o conceito de um *DataSource*
  - permite o uso de *pools* de conexões
  - permite o uso de transações distribuídas
  - cria o conceito de um *RowSet*

JDBC 2.1

- Estende o padrão inicial provendo:
  - movimentação livre do *cursor* no **ResultSet**
  - atualizações programáticas, via **ResultSet**
  - atualizações em lotes (*batch updates*)
  - compatibilidade com tipos SQL3 (SQL-99)
- Extensões Padrão (javax.sql)
  - Rowset Beans
  - JNDI – Java Naming and Directory Interface
  - JTS – Java Transaction Service

JDBC 3.0

- Estende o pacote e a extensão padrão provendo:
  - conceito de *savepoints* em transações
  - configuração do *pool* de conexões
  - obtenção de valores de colunas de preenchimento automático
  - manutenção de resultados abertos após a conclusão da transação
  - e outras funcionalidades

JDBC 4.0

- Estende o pacote e a extensão padrão provendo: