Universidade Estadual da Paraíba

Banco de Dados

Programação com banco de dados (JDBC)

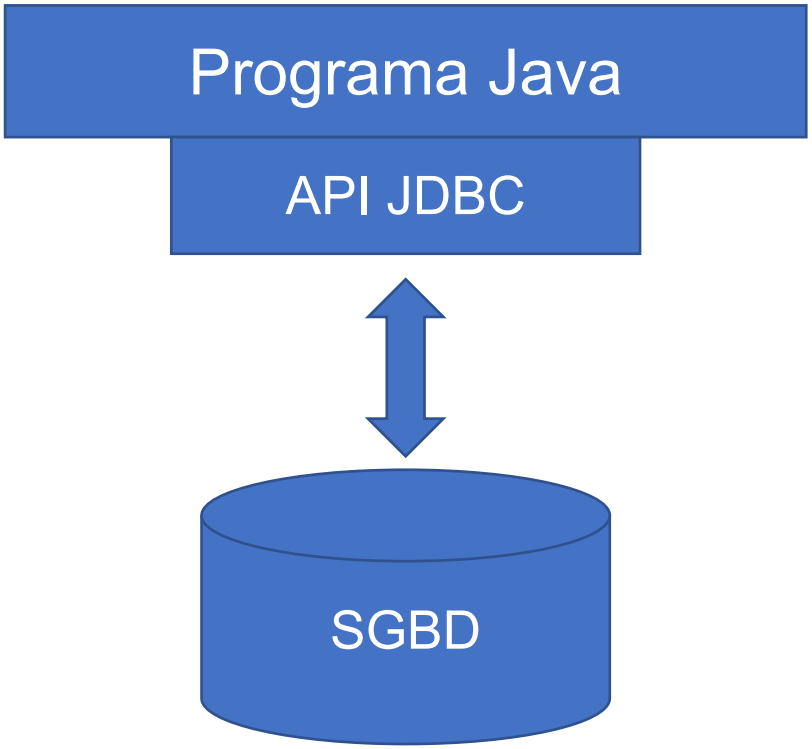
Prof. Dr. Vladimir Costa Alencar

valencar@gmail.com

https://www.valencar.com/

Programação com Banco de Dados (JDBC)

- **JDBC** é uma **API** (Application Program Interface) para acesso a **SGBD** relacionais por meio de comandos **SQL**



JDBC Características

- Fácil mapeamento objeto para relacional

- Independência de banco de dados (oracle, mysql, postgres, ...)

- Independente de plataforma (windows, linux, mac, android, ...)

Programação com Banco de Dados (JDBC)

- Java Database Connectivity ou JDBC é um conjunto de classes e interfaces (API) escritas em Java que fazem o envio de instruções SQL para qualquer banco de dados relacional
- Api de baixo nível e base para api's de alto nível
- Amplia o que você pode fazer com Java
- Possibilita o uso de bancos de dados já instalados

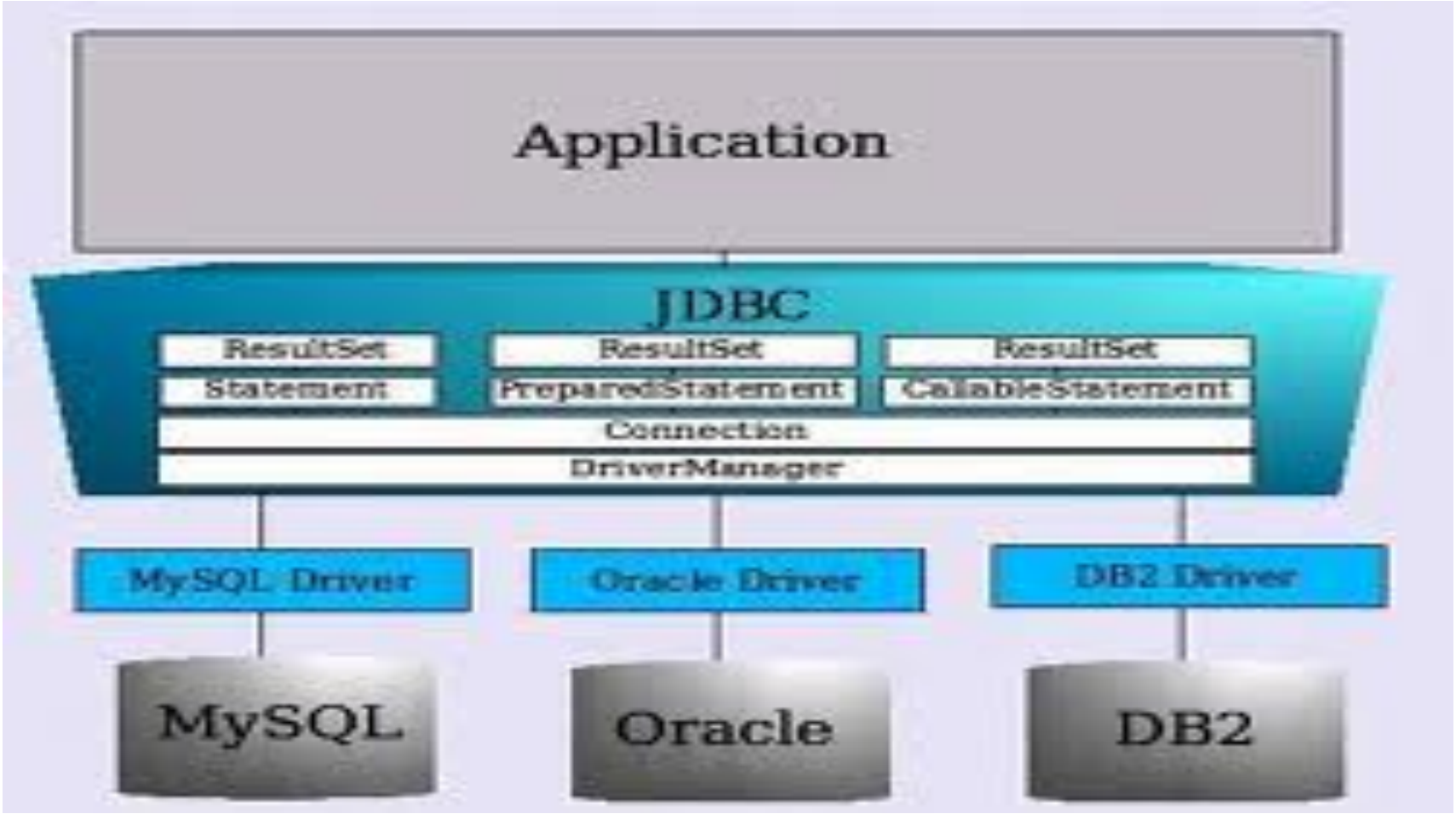
Programação com BD (JDBC)

- Torna fácil enviar statements SQL para os BDs, mas vai além do SQL pois permite interagir com outros tipos de fontes de dados, como arquivos

- Interoperabilidade: o mesmo programa Java acessa Oracle, Sybase ou MySQL

Programação com Banco de Dados (JDBC)

- Possibilita o uso de bancos de dados já instalados



Programação com BD (JDBC)

- Serve de base para outras APIs, como:

1. EJB: fornecer um rápido e simplificado desenvolvimento de aplicações Java baseado em componentes distribuídos, transacionais, seguros e portáveis (Java Beans)

2. JPA: é uma API padrão do java para persistência que deve ser implementada por frameworks que queiram seguir o padrão (ex. Hibernate – Mapeamento Objeto-Relacional)

3. SQLJ: criada pela Oracle, para facilitar o uso de JDBC

4. JDO: Persistência de Objetos

Vantagens (JDBC)

- a API para programação do sistema é a mesma para qualquer SGBD, não há necessidade de se desenvolver aplicações voltadas (e amarradas) para um BD específico
- permite a construção de páginas WWW que acessam BD pois dispensa a configuração da máquina cliente
- mantém a independência de plataforma da linguagem Java
- “A aplicação roda em qualquer sistema operacional acessando qualquer BD”

JDBC

A API encapsula:

1. o estabelecimento da conexão com o BD
2. o envio de comandos SQL
3. o processamento dos resultados

Comparação do JDBC com Microsoft ODBC

- ODBC tem o mesmo propósito e existe para várias plataformas, por que não usar ODBC?
- Os drivers ODBC são escritos em C, o que limita a portabilidade e auto-instalação dos programas Java
- O ODBC tem que ser instalado e configurado na máquina do Cliente
- Imagine instalar e configurar o ODBC em 40.000.000 computadores da Internet!
- Entretanto, bancos de dados que utilizam ODBC podem ser utilizados em aplicações Java via a ponte JDBC-ODBC.

Tipos de drivers JDBC

1. **Ponte com ODBC**
2. **Acesso ao driver nativo**

- a API JDBC chama procedimentos do driver nativo do SGBD instalado na máquina local

3. **Driver com protocolo proprietário escrito em Java**

- a comunicação entre o cliente e a SGBD dá-se no protocolo do SGBD, contudo, como o driver é escrito em Java, dispensa a instalação/configuração do driver no cliente.

4. **Driver independente de SGBD**

- utilizado pelo cliente para conectar-se com o middleware

- os dois primeiros tipos são recomendados para Intranets, pois exigem configuração da máquina cliente
- os outros dois podem ser utilizados na Internet, preferencialmente o último, que tem menor tempo de download.

Programação com BD (JDBC)

JDBC realiza as seguintes tarefas:

- Estabelece uma conexão com um banco de dados
- Executa comandos SQL DDL e DML
- Recebe um conjunto de resultados
- Executa stored procedures
- Obtém informações sobre o banco de dados (metadados)
- Executa transações

Ex. Programação com BD (JDBC)


```
Connection con =
DriverManager.getConnection("jdbc:meuDriver:meuDB",
"meuLogin", "minhaPassword");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(
                        "Select código, nome from Cliente");
While (rs.next()) {
        int cod = rs.getInt("código");
        String nome = rs.getString("nome");
        System.out.println( cod + "," + nome );
}
rs.close();
...
```

Estabelecendo uma conexão

- Carregando o driver:
  - Cria uma instância do driver e registra com o DriverManager

- Driver JDBC do mysql:

Class.forName ("org.gjt.mm.mysql.Driver");

- Ponte JDBC-ODBC:

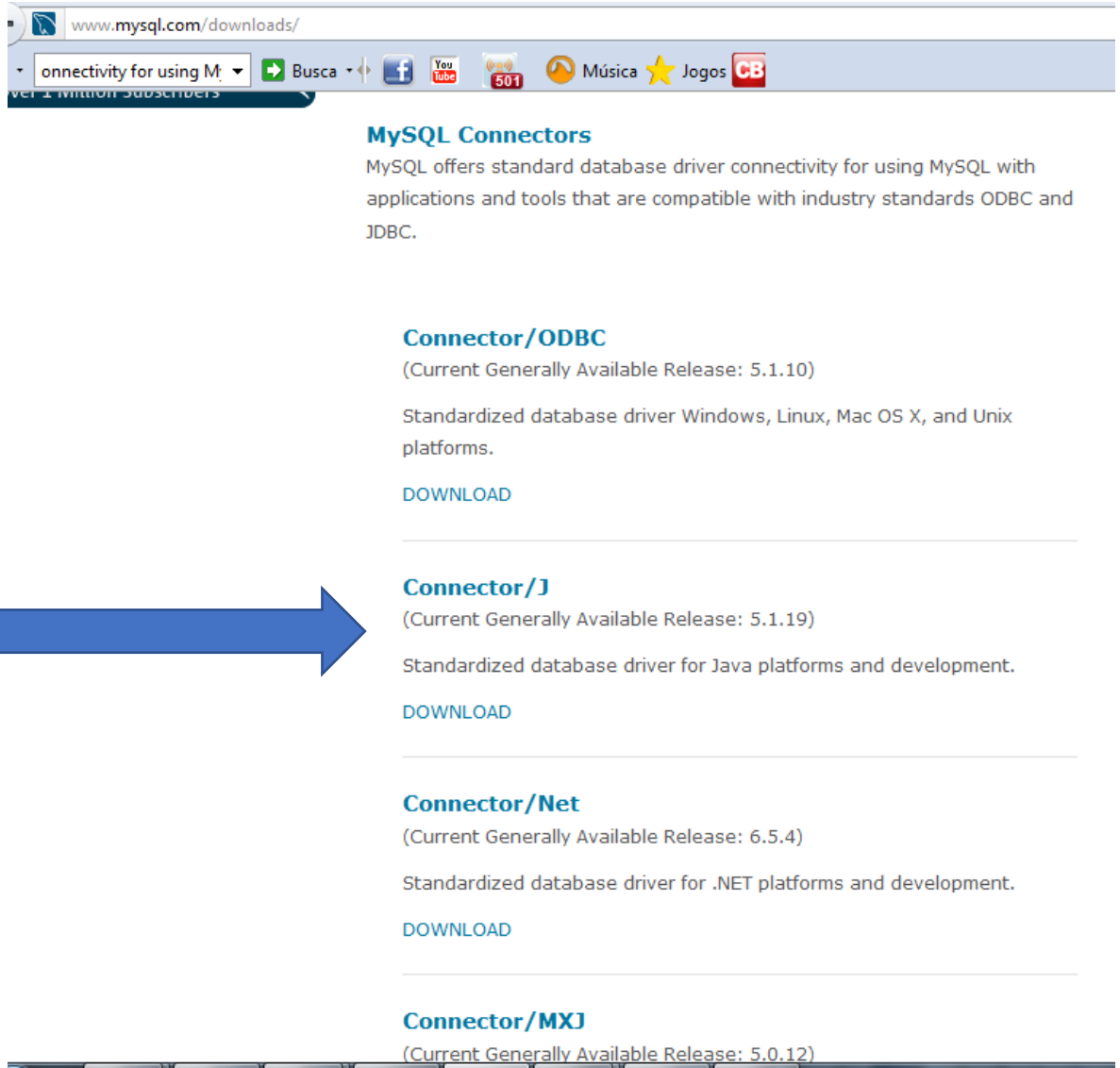
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

- A JVM deve poder encontrar estas classes para aplicações, a variável de ambiente CLASSPATH deve incluir os drivers

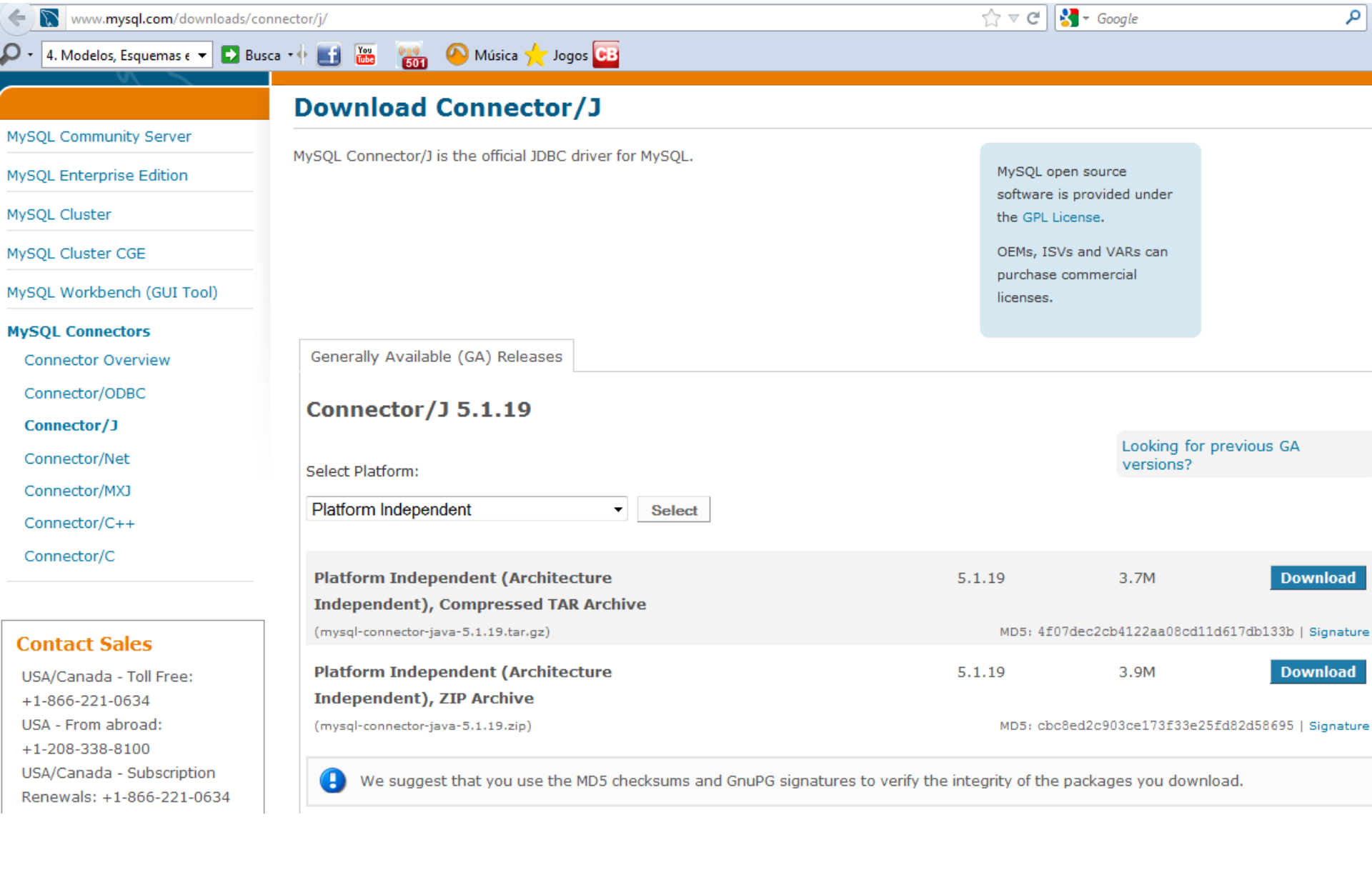
Onde achar o driver JDBC

- Cada SGDB implementa os drivers JDBC

Ex. Mysql



Onde achar o driver JDBC



Adicionando o driver JDBC

Ex. Mysql

- No CLASSPATH do java é só incluir:

Java –classpath .; c:\ mysql-connector-java-5.1.19.jar

- A JVM deve poder encontrar estas classes para aplicações, a variável de ambiente CLASSPATH deve incluir os drivers

Mapeamento de Tipos (SQL e Java)

SQLtype	JavaType	Tamanho
CHAR	String	0 até 255 caracteres
VARCHAR	String	armazena uma cadeia de longitude variável
LONGVARCHAR	String	
NUMERIC	java.math.BigDecimal	Número em vírgula flutuante desempacotado. O número armazena-se como uma cadeia.
DECIMAL	java.math.BigDecimal	O mesmo que NUMERIC
BIT	boolean	0,1
TINYINT	byte	-128 até 127 ou 0..255(sem sinal)
SMALLINT	short	-32768 até 32767

Mapeamento de Tipos (SQL e Java)

SQLtype	JavaType	Tamanho
INTEGER	int	-2.147.483.648 até 2.147.483.647
BIGINT	long	-9.223.372.036.854.775.808 até 9.223.372.036.854.775.807
REAL	float	-3.402823466E+38 até - 1.175494351E-38 e 175494351E-38 até 3.402823466E+38
FLOAT	double	-1.7976931348623157E+308 até -2.2250738585072014E-308, 0 e desde 2.2250738585072014E-308 até 1.7976931348623157E+308
BINARY	byte[]	

Mapeamento de Tipos (SQL e Java)

SQLtype	JavaType	Tamanho
DATE	java.sql.Date	1 de Janeiro de 1001 ao 31 de dezembro de 9999
TIME	java.sql.Time	-838 horas, 59 minutos e 59 segundos
TIMESTAMP	java.sql.Timestamp	1 de Janeiro de 1970 ao ano 2037 (Combinação de data e hora)
TEXT	String	um texto com um máximo de caracteres de 4GB
BLOB	blob	Um arquivo binário (imagem, som, vídeo) com no máximo de 4 GB

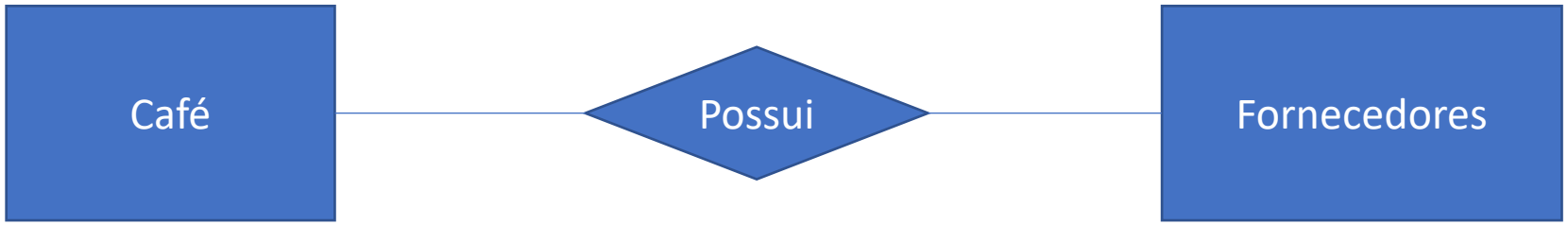
Criando uma conexão:


```
Import java.sql.*;

String serverName = "localhost";
String mydatabase = "aulas";
String url = "jdbc:mysql://" + serverName + "/" +
mydatabase;

Connection con = DriverManager.getConnection(
url, “meulogin”, “minhaPassword”);
```

Manipulando Tabelas

Vamos criar 2 tabelas:

1. tipos de café
2. fornecedores



Atributos

Nome_café
Id_fornecedor
Preço
Vendas
Total

Atributos

Id_fornecedor
Nome_fornecedor
Endeço
Cidade
Estado
CEP

Manipulando Tabelas

cafe

nome_cafe	id_fornecedor	preco	vendas	total
Colombiano	101	7,99	0	0
Francês_torrado	49	8,99	0	0
Espresso	150	9,99	0	0
Colombiano_Descafeinado	101	8,99	0	0
Francês_torrado_descafeinado	49	9,99	0	0

fornecedor

id_fornecedor	nome_fornecedor	endereco	cidade	estado	CEP
101	Nestlé	Av. Floriano Peixoto, 99	Campina Grande	PB	58128-087
49	Melita	Rua 25 de Março, 123	São Paulo	SP	95460-876
150	São Braz	Av. Júlio Prestes, 100	Curitiba	PR	93966-123

Criando Tabelas


```
String createTabCafe = "CREATE TABLE Cafe " +
"(nome_cafe VARCHAR(32),
id_fornecedor INTEGER,
 preco FLOAT, " +
"vendas INTEGER,
total INTEGER)";
```

```
String createTabFornecedores = "CREATE TABLE
Fornecedores " +
"(id_fornecedor INTEGER, nome_fornecedor
VARCHAR(40), endereco VARCHAR(40), " +
"cidade VARCHAR(20), estado VARCHAR(2), CEP
CHAR(9))";
```

Criando JDBC Statements

O envio de um comando SQL é feito por meio de um **Statement**

- Um statement pode ser criado a partir de três classes:

1. **Statement**
é utilizado para enviar comandos SQL simples

2. **PreparedStatement**
o comando SQL é pré-compilado e utilizado posteriormente, sendo mais eficiente nos casos onde o mesmo comando é utilizado várias vezes

3. **CallableStatement**
utilizado para chamar procedimentos SQL armazenados no BD

Criando Tabelas

```
String driverName = "org.gjt.mm.mysql.Driver";
String url = "jdbc:mysql://localhost/aulas“;

Class.forName(driverName);
Connection con = DriverManager.getConnection(
url, “root”, “root”);

Statement stmt = con.createStatement();
String createTabCafe = "CREATE TABLE cafe " +
"(nome_cafe VARCHAR(32), id_fornecedor INTEGER,
preco FLOAT, " + "vendas INTEGER, total INTEGER)";

stmt.executeUpdate(createTabCafe);
```

Criando Tabelas

- Tabela café (tipos de dados)

Field	Type	Null	Key	Default	E
nome_cafe	varchar(32)	YES			
id_fornecedor	int(11)	YES			
preco	float	YES			
vendas	int(11)	YES			
total	int(11)	YES			

Adicionando dados na tabela

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

stmt.executeUpdate("INSERT INTO cafe " +
"VALUES ('Colombian', 101, 7.99, 0, 0)");

stmt.executeUpdate("INSERT INTO cafe " +
"VALUES ('French_Roast', 49, 8.99, 0, 0)");
```

Values for the remaining rows can be inserted as follows:

```
stmt.executeUpdate("INSERT INTO cafe " +
"VALUES ('Espresso', 150, 9.99, 0, 0)");
stmt.executeUpdate("INSERT INTO cafe " +
"VALUES ('Colombian_Decaf', 101, 8.99, 0, 0)");
stmt.executeUpdate("INSERT INTO cafe " +
"VALUES ('French_Roast_Decaf', 49, 9.99, 0, 0)");
```

Adicionando dados na tabela

cafe

nome_cafe	id_fornecedor	preco	vendas	total
Colombiano	101	7,99	0	0
Francês_torrado	49	8,99	0	0
Espresso	150	9,99	0	0
Colombiano_Descafeinado	101	8,99	0	0
Francês_torrado_descafeinado	49	9,99	0	0

Recuperando valores com ResultSets

A API JDBC retorna resultados de consultas SELECT de um programa Java em um banco de dados num objeto do tipo **ResultSet**.

Ex.

```
ResultSet  rs = stmt.executeQuery(
"Select nome_cafe, preco from cafe”);
```

Obtenção do Resultado com ResultSets

- A classe ResultSet oferece à aplicação a tabela resultante de um Select e mantém um cursor posicionado em uma linha da tabela.

- Inicialmente este cursor está antes da primeira linha e o método **next()** movimenta o cursos para frente.

- Os dados das colunas da linha corrente são obtidos através do método getXXX(<cloluna>). Onde:
  XXX é o tipo da coluna
  <coluna> é o nome da coluna ou sua posição (a partir de 1)

Ex.

```
ResultSet rs = stmt.executeQuery(" Select nome_cafe, preco from cafe ");

while (rs.next()) {
String nome = rs.getString("nome_cafe");      // ou   rs.getstring(1);
float preco = rs.getFloat(2);                 //ou    rs.getFloat("preco");
System.out.println(nome + " - " + preco );
}
```

Obtenção do Resultado com ResultSets

- A classe ResultSet oferece à aplicação a tabela resultante de um Select e mantém um cursor posicionado em uma linha da tabela.

- Inicialmente este cursor está antes da primeira linha e o método **next()** movimenta o cursos para frente.

- Os dados das colunas da linha corrente são obtidos através do método getXXX(<cloluna>). Onde:
  XXX é o tipo da coluna
  <coluna> é o nome da coluna ou sua posição (a partir de 1)

Ex.

```
ResultSet rs = stmt.executeQuery(" Select nome_cafe, preco from cafe ");

while (rs.next()) {
String nome = rs.getString("nome_cafe");      // ou   rs.getstring(1);
float preco = rs.getFloat(2);                 //ou    rs.getFloat("preco");
System.out.println(nome + " - " + preco );
}
```

Obtenção do Resultado com ResultSets

Obtendo o nomes das colunas de uma tabela

```
int coluna = 1;
while ( coluna <= r.getColumnCount()) {
        System.out.println(coluna  +" - "+ coluna +
" : " +        r.getColumnLabel(coluna) );
        coluna++;
}
```

Obtenção do Resultado com ResultSets

```
Class.forName(driverName);
con = DriverManager.getConnection(url, username, password);
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(" Select nome_cafe, preco from cafe ");

while (rs.next()) {
String nome = rs.getString("nome_cafe");      // ou   rs.getstring(1);
float preco = rs.getFloat(2);                 //ou    rs.getFloat("preco");
System.out.println(nome + " - " + preco );
}
```

Saída:

```
Colombiano - 7.99
Francês_torrado - 8.99
Espresso - 9.99
Colombiano_Descafeinado - 8.99
Francês_torrado_descafeinado - 9.99
```

Atualizando as Tabelas

```
String updateString = "UPDATE cafe " +
"set vendas = 75 " +
"where nome_cafe = 'Colombiano' ";
stmt.executeUpdate(updateString);

ResultSet rs = stmt.executeQuery(" Select nome_cafe, preco, vendas from cafe ");
System.out.format("Nome                    Preco  Vendas \n");
while (rs.next()) {
String nome = rs.getString("nome_cafe");
float preco = rs.getFloat("preco");
float vendas = rs.getFloat("vendas");
System.out.format("%-30s  %-5.2f  %5.2f \n",nome, preco, vendas); }
```

```
Nome                           Preco  Vendas
Colombiano                     7,99   75,00
Francês_torrado                8,99    0,00
Espresso                       9,99    0,00
Colombiano_Descafeinado        8,99    0,00
Francês_torrado_descafeinado   9,99    0,00
```

Usando Prepared Statements

- Usados para enviar parâmetros para comandos SQL

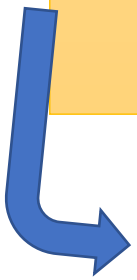
```
PreparedStatement atualizaVendas = con.prepareStatement(
"UPDATE cafe set vendas = ? where nome_cafe = ?");
atualizaVendas.setInt(1, 78);
atualizaVendas.setString(2, "Colombiano");
atualizaVendas.executeUpdate();
```

UPDATE cafe set vendas = **78** where nome_cafe = **"Colombiano"**

Nome	Preco	Vendas
Colombiano	7,99	78,00
Francês_torrado	8,99	0,00
Espresso	9,99	0,00
Colombiano_Descafeinado	8,99	0,00
Francês_torrado_descafeinado	9,99	0,00

Usando um loop

```
String str_atualizaVendas = "update cafe set vendas = ? where nome_cafe = ?";
PreparedStatement  atualizaVendas = con.prepareStatement(str_atualizaVendas);

int [] vendasPorSemana = {175, 150, 60, 155, 90};
String [] cafe = {"Colombiano", "Francês_torrado", "Espresso",
"Colombiano_Descafeinado", "Francês_torrado_descafeinado"};
int len = cafe.length;
for(int i = 0; i < len; i++) {
atualizaVendas.setInt(1, vendasPorSemana[i]);
atualizaVendas.setString(2, cafe[i]);
atualizaVendas.executeUpdate();
}
```



Nome	Preco	Vendas
Colombiano	7,99	175,00
Francês_torrado	8,99	150,00
Espresso	9,99	60,00
Colombiano_Descafeinado	8,99	155,00
Francês_torrado_descafeinado	9,99	90,00

Utilizando Joins

```
Class.forName(driverName);
con = DriverManager.getConnection(url, username, password);
Statement stmt = con.createStatement();

String query = "SELECT cafe.nome_cafe " +
" FROM cafe, fornecedores " +
" WHERE fornecedores.nome_fornecedor = 'Nestlé'    and " +
" fornecedores.id_fornecedor = cafe.id_fornecedor";
ResultSet rs = stmt.executeQuery(query);
System.out.println("Cafes comprados da Empresa Nestlé: ");
while (rs.next()) {
String nomeCafe = rs.getString("nome_cafe");
System.out.println(" " + nomeCafe);
 }
```

```
Cafes comprados da Empresa Nestlé:
 Colombiano
 Colombiano_Descafeinado
```

Liberando recursos

Após o uso precisamos fechar o statement e a conexão usando os métodos:

- rs.close();
- stmt.close();
- con.close();

Metadados de um ResultSet

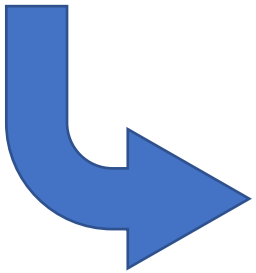
Os Metadados são informações sobre os dados da tabela.

- Quais os nomes e os tipos das colunas
- Se a coluna do tipo é numérico com sinal
- Se a coluna pode ser alterada, etc...

Utilizamos o objeto ResultSetMetaData para obter essas informações.

Metadados de um ResultSet

```
ResultSet result = stmt.executeQuery(
"SELECT * FROM cafe");
ResultSetMetaData meta = result.getMetaData();
int columns = meta.getColumnCount();
System.out.println("Campo          TIPO      Tamanho Aceita_Nulos?");
for (int i=1; i<=columns; i++) {
  String campo = meta.getColumnLabel(i);
  String tipo = meta.getColumnTypeName(i);
  int tamanho = meta.getColumnDisplaySize(i);
  int aceitaNulos = meta.isNullable(i);
  System.out.format("%-15s  %-7s  %-6d      %d \n", campo, tipo,
                    tamanho, aceitaNulos);
}
```



```
Campo           TIPO     Tamanho  Aceita_Nulos?
nome_cafe       VARCHAR  32         1
id_fornecedor   INT      11         0
preco           FLOAT    12         1
vendas          INT      11         1
total           INT      11         1
```

MetaDados de um Banco de Dados

```
String mydatabase = "aulas";
String url = "jdbc:mysql://localhost/" + mydatabase;
Class.forName(driverName);
con = DriverManager.getConnection(url, username, password);

DatabaseMetaData dbmd = con.getMetaData();

String   catalog        = null;
String   schemaPattern  = null;
String   tableNamePattern = null;
String[] types          = null;

ResultSet rs = dbmd.getTables(
   catalog, schemaPattern, tableNamePattern, types );

System.out.println("Tabelas do banco de dados: " + mydatabase);
while(rs.next()) {
   System.out.println(rs.getString(3));
}
```

MetaDados de um Banco de Dados

```
ResultSet rs = dbmd.getTables(
   catalog, schemaPattern, tableNamePattern, types );

System.out.println("Tabelas do banco de dados: " + mydatabase);
while(rs.next()) {
   System.out.println(rs.getString(3));
}
```

```
Tabelas do banco de dados: aulas
agencia
cafe
cliente
departamento
empregado
fornecedores
funcionario
funcionário
item_do_pedido
pedido
produto
vendedor
produtos_metro
salário_médio
```

Transações

- Uma transação é um conjunto de Statements que são validados no BD com **commit** ou cancelados com **rollbabk**
- Por default, todos os comandos no JDBC são auto-commit.

```
con.setAutoCommit(false); // muda o default
Statement s = con.createStatement();
try {
        s.executeUpdate("SQL statement 1");
        s.executeUpdate("SQL statement 2");
        con.commit(); // valida os 2 updates
} catch (Exception e) {
        con.rollback(); // senão, desfaz os updates
}
```

Transações - commit

```
con.setAutoCommit(false);

PreparedStatement atualizaVendas = con.prepareStatement(
"UPDATE Cafe SET vendas = ? WHERE nome_cafe = ?");
atualizaVendas.setInt(1, 50);
atualizaVendas.setString(2, "Colombiano");
atualizaVendas.executeUpdate();

PreparedStatement atualizaTotal = con.prepareStatement(
"UPDATE cafe SET TOTAL = TOTAL + ? WHERE nome_cafe = ?");
atualizaTotal.setInt(1, 50);
atualizaTotal.setString(2, "Colombiano");
atualizaTotal.executeUpdate();

con.commit();  // executa (confirma) a transação
con.setAutoCommit(true);
```

Quando fazer um RollBack numa Transaction

A ativação do método **rollback()** aborta a transação e retorna aos valores anteriores (antes do início da transação)

Se tentantarmos executar um ou mais statements na transação e obter uma exceção **SQLException**, nós deveremos chamar o método rollback para abortar a transação e restaurar os valores.

A exceção **SQLException** nos diz que alguma coisa deu errado, mas não especifica o quê deu errado.

Quando fazer um RollBack numa Transaction

```
...
try {
        con.setAutoCommit(false);
        PreparedStatement atualizaVendas = con.prepareStatement(
        "UPDATE Cafe SET vendas = ? WHERE nome_cafe = ?");
        atualizaVendas.setInt(1, 50);
        atualizaVendas.setString(2, "Colombiano");
        atualizaVendas.executeUpdate();
....
} catch(SQLException ex) {
        System.err.println("SQLException: " + ex.getMessage());
        if (con != null) {
          try {
                con.rollback();
                System.err.print("Transação revertida (rollback)");
          } catch(SQLException excep) {
                System.err.print("SQLException: ");
                System.err.println(excep.getMessage());
          }
        }
}
```

O que são Savepoints

Os Savepoints permitem ao desenvolvedor uma maior controle sobre as transações.

Uma transação pode conter vários comandos, e quando a transação é confirmada (commit), todos os comandos são confirmados (gravados) no Banco de Dados.

Se a transação é revertida (rollback), NENHUM dos comandos são gravados no BD.

O que são Savepoints

Um Objeto Savepoint serve para fazer um **ponto de marcação** intermediária dentro da transação, tornado possível a reversão (rollback) da transação até o ponto de marcação, ou invés de reverter (rollback) a transação inteira.

Em outras palavras, se o método rollback é chamado dentro com um objeto Savepoint, TUDO que foi feito depois do Savepoint é **desfeito**.

Usando um Savepoint

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();
int numLinhas = stmt.executeUpdate(
                        "insert into cafe values (‘Zambia', 101, 7.99, 0, 0)";
Savepoint save1 = con.setSavepoint("SAVEPOINT_01");
numLinhas = stmt.executeUpdate(
                        "insert into cafe values ('Paulista', 101, 7.99, 0, 0)");
//.... outros comandos SQL
con.rollback(save1);
con.commit();
```

Nome Café	Id_fornecedor	Preço
Amaretto	49	9,99
Amaretto_descafeinado	49	10,99
Colombiano_Descafeinado	101	8,99
Espresso	150	9,99
Fiomiccino	49	14,99
Francês_torrado	49	8,99
Francês_torrado_descafeinado	49	15,50
Mineirinho	150	10,99
Zambia	101	7,99

Stored Procedures

É um grupo de comandos SQL para realizar uma tarefa específica.

São usados para encapsular um conjunto de operações ou queries (consultas) com vistas de ser executado num SGBD.

Os procedimentos são armazenados no Banco de Dados.

Ex. Stored Procedures (Sintaxe)

```
Create procedure mostrar_fornecedor()
BEGIN
SELECT   fornecedores.nome_fornecedor, café.nome_cafe
FROM     cafe, fornecedor
WHERE   fornecedores.id_fornecedores = cafe.id_fornecedores;
END
```

Stored Procedure

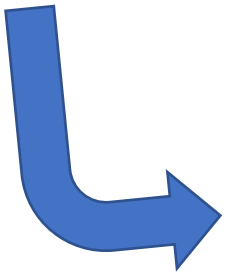
```
String removaProcedure = "drop procedure if exists mostrar_fornecedores;";

String createProcedure = "create procedure mostrar_fornecedores() " +
" begin " +
" select fornecedores.nome_fornecedor, cafe.nome_cafe " +
" from fornecedores, cafe " +
" where fornecedores.id_fornecedor = cafe.id_fornecedor;" +
" end ";

Statement stmt = con.createStatement();
stmt.executeUpdate(removaProcedure);
stmt.executeUpdate(createProcedure);
```

Chamando uma Stored Procedure

```
 String removaProcedure = "drop procedure if exists
mostrar_fornecedores;";

String createProcedure = "create procedure
mostrar_fornecedores() " +
" begin " +
" select fornecedores.nome_fornecedor, cafe.nome_cafe " +
" from fornecedores, cafe " +
" where fornecedores.id_fornecedor = cafe.id_fornecedor;" +
" end ";

Statement stmt = con.createStatement();
stmt.executeUpdate(removaProcedure);
stmt.executeUpdate(createProcedure);
```

Chamando uma Stored Procedure

```
Class.forName(driverName);
con = DriverManager.getConnection(url, username, password);
CallableStatement cs = con.prepareCall("{call mostrar_fornecedores()}");

ResultSet rs = cs.executeQuery();
System.out.format("Nome_fornecedor  Nome_Café\n");
while (rs.next()) {
        String nome_fornecedor = rs.getString("nome_fornecedor");
        String nome_cafe = rs.getString("nome_cafe");
        System.out.format("%-10s \t %s \n",nome_fornecedor, nome_cafe);
}
con.close();
```



```
Nome_fornecedor  Nome_Café
Nestlé            Colombiano
Melita            Francês_torrado
São Braz          Espresso
Nestlé            Colombiano_Descafeinado
Melita            Francês_torrado_descafeinado
```

Recuperando exceções no JDBC

```
try {
        Class.forName("myDriverClassName");
}  catch (java.lang.ClassNotFoundException e) {
        System.err.print("ClassNotFoundException: ");
        System.err.println(e.getMessage());
}

try {
// O Código que pode gerar a exceção está neste bloco.
// Se uma exceção for gerada, o bloco catch abaixo irá
// imprimir informações sobre ela
        <comandos JDBC e JAVA>
} catch(SQLException ex) {
        System.err.println("SQLException: " +
        ex.getMessage());
}
```

Recuperando exceções no JDBC

```
 try {
Class.forName(driverName);
con = DriverManager.getConnection(url, username, password);
String createTabCafe = "CREATE TABLE Cafe " +
"(nome_cafe VARCHAR(32), id_fornecedor INTEGER, preco FLOAT, " +
"vendas INTEGER, total INTEGER)";
stmt = con.createStatement();
stmt.executeUpdate(createTabCafe);
con.close();
 } catch(SQLException ex) {
    System.err.println("SQLException: " + ex.getMessage());
 }
```

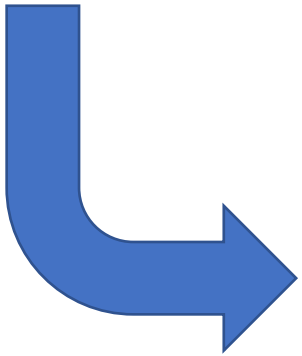
SQLException: Table 'cafe' already exists

Recuperando exceções no JDBC

```
 try {
Class.forName(driverName);
con = DriverManager.getConnection(url, username, password);
String createTabCafe = "CREATE TABLE Cafe " +
"(nome_cafe VARCHAR(32), id_fornecedor INTEGER, preco FLOAT, " +
"vendas INTEGER, total INTEGER)";
stmt = con.createStatement();
stmt.executeUpdate(createTabCafe);
con.close();
 } catch(SQLException ex) {
        System.err.println("SQLException: " + ex.getMessage());
        System.out.println("\n--- SQLException caught ---\n");
        while (ex != null) {
                System.out.println("Message: " + ex.getMessage ());
                System.out.println("SQLState: " + ex.getSQLState ());
                System.out.println("ErrorCode: " + ex.getErrorCode ());
                ex = ex.getNextException();
                System.out.println("");
        }
}
```

Recuperando exceções no JDBC

```
} catch(SQLException ex) {
        System.err.println("SQLException: " + ex.getMessage());
        System.out.println("\n--- SQLException caught ---\n");
        while (ex != null) {
                System.out.println("Message: " + ex.getMessage ());
                System.out.println("SQLState: " + ex.getSQLState ());
                System.out.println("ErrorCode: " + ex.getErrorCode ());
                ex = ex.getNextException();
                System.out.println("");
        }
}
```



```
--- SQLException caught ---

SQLException: Table 'cafe' already exists
Message: Table 'cafe' already exists
SQLState: 42S01
ErrorCode: 1050
```

Recuperando Warnings (advertências)

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("select nome_cafe from cafe");
SQLWarning warning = stmt.getWarnings();
if (warning != null) {
        System.out.println("\n---Warning---\n");
        while (warning != null) {
                System.out.println("Message: " + warning.getMessage());
                System.out.println("SQLState: " + warning.getSQLState());
                System.out.print("Vendor error code: “);
                System.out.println(warning.getErrorCode());
                System.out.println("");
                warning = warning.getNextWarning();
        }
}
```

OBS: Os Warnings atualmente não são muito usados

Cursor

Aplicativos, especialmente aplicativos online interativos, não podem sempre trabalhar efetivamente com todo o conjunto de resultados como uma unidade. Estes aplicativos precisam de um mecanismo para trabalhar com **uma linha** ou um **bloco pequeno de linhas por vez**.

Os **cursores** são uma extensão dos conjuntos de resultados que provêem esse mecanismo

Cursor é um recurso em bancos de dados que permite que seus códigos SQL façam uma **varredura** de uma tabela ou consulta **linha-por-linha**, realizando mais de uma operação se for o caso.

Pode ser visto como um arquivo temporário que armazena e controla as linhas retornadas de um comando SQL.

Movendo o cursor

A API JDBC possui a funcionalidade de mover um cursor de um **Resultset** para frente ou para trás.

São métodos que movem o cursor para uma linha em particular e que verificam a posição de um cursor.

“Resultset Roláveis” tornam mais fácil criar uma **interface grafica** para visualizar os dados, entre outras coisas.

Outro importante uso é **mover o cursor** para uma linha que você vai realizar atualizações nela.

Manipulando dados com o ResultSet

```
Class.forName(driverName);
con = DriverManager.getConnection(url, username, password);
String Query = "select nome_cafe from cafe";
Statement stmt = con.createStatement(ResultSet.TYPE_FORWARD_ONLY,
ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery(Query);
```

- Esta seqüência de comandos cria o ResultSet através do qual se poderá manipular o resultado da consulta.
- Um ResultSet pode ter milhares de registros mas normalmente somente um deles é manipulado de cada vez o Registro Corrente
- Este registro é manipulado através de um Cursor que pode ser movimentado utilizando-se diversos métodos da classe para tornar outro registro o registro corrente
- O **método next** de ResultSet torna o registro seguinte o registro corrente e se o registro corrente for o último, retorna false. (Este valor pode ser usado em uma estrutura de controle para percorrer todos os registros)
- Cada registro é composto de diversos campos de diferentes tipos de dados
- Os objetos ResultSet têm portanto diferentes métodos para ler os diferentes tipos de dados.

Métodos de um ResultSet

Os objetos da interface ResultSet são centrais na utilização de JDBC Verificar na API Java a lista completa de campos e métodos do objeto
A interface é muito rica e entre seus métodos se pode citar:

– absolute: permite que se acesse uma linha específica de uma tabela;
– first: torna a primeira linha de uma tabela o registro corrente;
– last: torna a última linha de uma tabela o registro corrente;
– next: posiciona o cursor no registro seguinte ao registro corrente;
– previous: posiciona o cursor no registro anterior ao registro corrente;
– relative: posiciona o cursor em um registro com posição relativa (+/-) ao registro corrente;

Métodos de um ResultSet

– updateRow: permite que se atualize uma linha de uma tabela;
– deleteRow: permite que se suprima uma linha de uma tabela;
– updateRow: permite que se atualize uma linha de uma tabela;
– isFirst: se o registro corrente for o primeiro, retorna true, caso contrário retorna false;
– isLast: se o registro corrente for o primeiro, retorna true, caso contrário retorna false;
– isBeforeFirst: se o registro corrente for anterior ao primeiro, retorna true, caso contrário retorna false;
– isAfterLast: se o registro corrente for seguinte ao último, retorna true, caso contrário retorna false;
– getType: sendo Type um dos tipos java como int, long, string, etc. permite que se recupere um atributo de um tipo específico de uma linha de uma tabela.

Navegação no ResultSet

Um ResultSet pode funcionar com diferentes tipos de cursor
O cursor padrão (forward_only) só permite um movimento para a frente
Somente o método next funciona.
Os diferentes tipos de cursor:

– TYPE_FORWARD_ONLY: cria um objeto ResultSet cujo cursor só pode se movimentar para a frente. Se nada for especificado um cursor deste tipo será criado.

– TYPE_SCROLL_INSENSITIVE: cria um objeto ResultSet cujo cursor é bidirecional
As mudanças feitas no registro não serão visíveis enquanto o ResultSet estiver aberto.

– TYPE_SCROLL_SENSITIVE: cria um objeto ResultSet cujo cursor é bidirecional
As mudanças feitas no registro serão visíveis enquanto o ResultSet estiver aberto.

Navegação no ResultSet

Quando se declara o tipo do ResultSet deve-se obrigatoriamente definir como os dados do registro corrente poderão ser manipulados pelos outros usuários da base:

– CONCUR_READ_ONLY: cria um objeto ResultSet que não pode ser atualizado enquanto estiver sendo utilizado.

– CONCUR_UPDATABLE: cria um objeto ResultSet que pode ser atualizado enquanto estiver sendo utilizado.

Algumas vezes mesmo após especificar o tipo de cursor não se pode movimentar para frente e para trás

O driver do Banco de Dados sendo utilizado deve implementar esta funcionalidade, caso contrário de nada adiante definir um cursor não padrão.

Um driver pode oferecer esta característica mesmo que o Banco de dados não a ofereça, mas normalmente se o banco não oferece a possibilidade de movimentação bidirecional o driver não é obrigado a apresentar a característica.

Navegação no ResultSet

O seguinte código pode ser utilizado para verificar o tipo de cursor utilizado pelo ResultSet:

```
Class.forName(driverName);
con = DriverManager.getConnection(url, username, password);

String Query = "select nome_cafe from cafe";

Statement stmt = con.createStatement(ResultSet.TYPE_FORWARD_ONLY,
ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery(Query);
int type = rs.getType();
System.out.println("TYPE_FORWARD_ONLY: " + type);
....
```

```
TYPE_FORWARD_ONLY: 1003
TYPE_SCROLL_INSENSITIVE: 1004
TYPE_SCROLL_SENSITIVE: 1005
```

Movendo o Cursor para frente

```
String Query = "select * from cafe";
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                     ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery(Query);
srs.beforeFirst();
System.out.format("Nome Café \t \t \t Preço \n");
while (srs.next()) {
        String nome_cafe = srs.getString("nome_cafe");
        float preco = srs.getFloat("preco");
        System.out.format("%-30s \t %-5.2f \n", nome_cafe, preco);
}
```

```
Nome Café                       Preço
Colombiano                       7,99
Francês_torrado                  8,99
Espresso                         9,99
Colombiano_Descafeinado          8,99
Francês_torrado_descafeinado     9,99
```

Movendo o Cursor para tras

```
String Query = "select * from cafe";
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery(Query);
System.out.format("Nome Café \t \t \t Preço \n");
srs.afterLast(); // posiciona o cursor após o último registro
while (srs.previous()) {  // Posiciona no registro anterior
        String nome_cafe = srs.getString("nome_cafe");
        float preco = srs.getFloat("preco");
        System.out.format("%-30s \t %-5.2f \n", nome_cafe, preco);
}
```

```
Nome Café                          Preço
Colombiano                          7,99
Francês_torrado                     8,99
Espresso                            9,99
Colombiano_Descafeinado             8,99
Francês_torrado_descafeinado        9,99
```

Movendo o Cursor para uma linha específica

```
...
String Query = "select * from cafe";
Statement stmt = con.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE,
                                      ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery(Query);

System.out.format("    Posição do cursor \t Nome Café \t\t    Preço \n");
srs.absolute(4); // cursor está na 4a linha
String nome_cafe = srs.getString("nome_cafe");
float preco = srs.getFloat("preco");
System.out.format("4a linha: ");
System.out.format("%-25s  %-5.2f \n", nome_cafe, preco);

srs.relative(-3); // cursor está na 1a linha
....
srs.relative(2); // cursor está na 3a linha
...
```

Nome Café	Preço
Colombiano	7,99
Francês_torrado	8,99
Espresso	9,99
Colombiano_Descafeinado	8,99
Francês_torrado_descafeinado	9,99

Posição do cursor	Nome Café	Preço
4a linha	Colombiano_Descafeinado	8,99
1a linha	Colombiano	7,99
3a linha	Espresso	9,99

Obtendo a posição do cursor

```
String Query = "select * from cafe";
Statement stmt = con.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE,
                                      ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery(Query);
System.out.format("Posição do cursor \tNome Café \t\t   Preço \n");
srs.absolute(4); // cursor está na 4a linha
int NumeroLinha = srs.getRow();
String nome_cafe = srs.getString("nome_cafe");
float preco = srs.getFloat("preco");
System.out.format("%-2d  \t\t\t%-25s  %-5.2f \n", NumeroLinha, nome_cafe, preco);
...
srs.relative(-3); // cursor está na 1a linha
NumeroLinha = srs.getRow();
...
srs.relative(2); // cursor está na 3a linha
NumeroLinha = srs.getRow();
```

Nome Café	Preço
Colombiano	7,99
Francês_torrado	8,99
Espresso	9,99
Colombiano_Descafeinado	8,99
Francês_torrado_descafeinado	9,99

Posição do cursor	Nome Café	Preço
4	Colombiano_Descafeinado	8,99
1	Colombiano	7,99
3	Espresso	9,99

Obtendo a posição do cursor

Quatro métodos adicionais permitem verificar o cursor em uma determinada posição.
São eles: isFirst(), isLast(), isBeforeFirst(), isAfterLast().

Estes métodos retornam um booleano e podem ser usados numa statement condicional.

```
Statement stmt = con.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE,
                                      ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("select * from cafe“);
srs.absolute(4); // cursor está na 4a linha
if (! srs.isAfterLast()) { // não atingiu o final dos Resultados
        String nome_cafe = srs.getString("nome_cafe");
        float preco = srs.getFloat("preco");
        System.out.println(nome_cafe + " " + preco);
}
```

Criando um ResultSet Atualizável

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("select * from cafe");

uprs.beforeFirst();
System.out.format("Nome Café \t \t \t Preço \n");
while (uprs.next()) {
        String nome_cafe = uprs.getString("nome_cafe");
        float preco = uprs.getFloat("preco");
        System.out.format("%-30s \t %-5.2f \n", nome_cafe, preco);
}
```

- ***ResultSet.CONCUR_UPDATABLE*** → *Habilita o Resultset para Consulta, Atualização e Remoção*

- Alguns Drivers não suportam atualizações no ResultSet

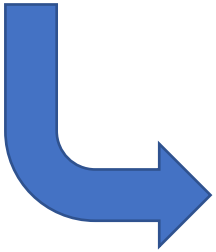
Verificando se um ResultSet é Atualizável

```
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                    ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("select * from cafe");

int tipo = uprs.getConcurrency();
System.out.println("tipo = " + tipo);
```

A variável tipo pode assumir os valores:
1007 indica ResultSet.CONCUR_READ_ONLY
1008 indica ResultSet.CONCUR_UPDATABLE

Atualizando um ResultSet com SQL

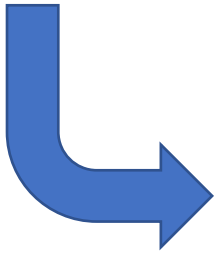
```
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                        ResultSet.CONCUR_UPDATABLE);
int numRegistrosAtualizados = stmt.executeUpdate("update cafe set preco =
                                                    preco + 2");
System.out.println("Número de Registros Aualizados = " +
                                                  numRegistrosAtualizados);
ResultSet uprs = stmt.executeQuery("select nome_cafe, preco from cafe");
```



```
Número de Registros Aualizados = 5
Nome Café                          Preço
Colombiano                         9,99
Francês_torrado                    10,99
Espresso                           11,99
Colombiano_Descafeinado            10,99
Francês_torrado_descafeinado       11,99
```

Atualizando um ResultSet diretamente do java

```
...
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("select * from cafe");
uprs.last(); // último registro
uprs.updateDouble("preco", 15.50);
uprs.updateRow(); // Atualiza no Banco de Dados
...
```



Nome Café	Preço
Colombiano	7,99
Colombiano_Descafeinado	8,99
Espresso	9,99
Francês_torrado	8,99
Francês_torrado_descafeinado	15,50

Inserindo Linhas

```
stmt.executeUpdate("INSERT INTO COFFEES " +
"VALUES ('Mineirinho', 150, 10.99, 0, 0)");
```

Semelhante

```
...
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("select * from cafe");

uprs.moveToInsertRow(); // posiciona o cursor para inserção
uprs.updateString("nome_cafe", "Mineirinho");
uprs.updateInt("id_fornecedor", 150);
uprs.updateDouble("preco", 10.99);
uprs.updateInt("vendas", 0);
uprs.updateInt("total", 0);
uprs.insertRow(); // Atualiza no Banco de Dados
...
```

Inserindo Linhas

```
...
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("select * from cafe");

uprs.moveToInsertRow(); // posiciona o cursor para inserção
uprs.updateString("nome_cafe", "Mineirinho");
uprs.updateInt("id_fornecedor", 150);
uprs.updateDouble("preco", 10.99);
uprs.updateInt("vendas", 0);
uprs.updateInt("total", 0);
uprs.insertRow(); // Atualiza no Banco de Dados
...
```

Nome Café	id_fornecedor	Preço	Vendas	Total
Colombiano	101	7,99	175	50
Colombiano_Descafeinado	101	8,99	155	0
Espresso	150	9,99	60	0
Francês_torrado	49	8,99	150	0
Francês_torrado_descafeinado	49	15,50	90	0
Mineirinho	150	10,99	0	0

Removendo linhas

```
...
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("select * from cafe");

uprs.absolute(6); // último registro
uprs.deleteRow(); // Atualiza no Banco de Dados
...
```

Nome Café	id_fornecedor	Preço	Vendas	Total
Colombiano	101	7,99	175	50
Colombiano_Descafeinado	101	8,99	155	0
Espresso	150	9,99	60	0
Francês_torrado	49	8,99	150	0
Francês_torrado_descafeinado	49	15,50	90	0

Usando Batch Updates (atualizações em lote)

Um Batch update é um conjunto de múltiplos statements de atualizações que é submetido ao banco de dado para ser processado no modo batch (lote de comandos).

Enviando batch apdates podemos, em algumas situações, termos uma maior eficiência que executar os comandos em separado.

Um exemplo é quando queremos inserir algumas tuplas (e assegurar que TODAS foram inseridas sem problemas)

Usando Batch Updates (atualizações em lote)

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.addBatch("INSERT INTO cafe " +  "VALUES('Amaretto', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO cafe " +  "VALUES('Pilar', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO Cafe " +  "VALUES('Amaretto_descafeinado', 49, 10.99, 0, 0)");
stmt.addBatch("INSERT INTO cafe " +  "VALUES('Pilar_descafeinado', 49, 10.99, 0, 0)");
int [] updateCounts = stmt.executeBatch();
con.commit();
con.setAutoCommit(true);
```

Nome Café	id_fornecedor	Preço	Vendas	Total
Amaretto	49	9,99	0	0
Amaretto_descafeinado	49	10,99	0	0
Colombiano	101	7,99	175	50
Colombiano_Descafeinado	101	8,99	155	0
Espresso	150	9,99	60	0
Francês_torrado	49	8,99	150	0
Francês_torrado_descafeinado	49	15,50	90	0
Mineirinho	150	10,99	0	0
Pilar	49	9,99	0	0
Pilar_descafeinado	49	10,99	0	0

Usando Batch Updates Parametrizáveis

```
con.setAutoCommit(false); // desabilita o modo de gravação no BD automático
PreparedStatement pstmt = con.prepareStatement("INSERT INTO cafe VALUES(?, ?, ?, ?, ?)");
pstmt.setString(1, "Fiomiccino");
pstmt.setInt(2, 49);
pstmt.setDouble(3, 14.99);
pstmt.setInt(4, 0);
pstmt.setInt(5, 0);
pstmt.addBatch();

pstmt.setString(1, "Camponesa");
pstmt.setInt(2, 49);
pstmt.setDouble(3, 12.99);
pstmt.setInt(4, 0);
pstmt.setInt(5, 0);
pstmt.addBatch();

pstmt.setString(1, "Cubano");
pstmt.setInt(2, 49);
pstmt.setDouble(3, 11.99);
pstmt.setInt(4, 0);
pstmt.setInt(5, 0);
pstmt.addBatch();

int [] updateCounts = pstmt.executeBatch(); // executa os comandos em Lote
con.commit();
con.setAutoCommit(true);
```

Usando Batch Updates Parametrizáveis

```
con.setAutoCommit(false); // desabilita o modo de gravação no BD automático
PreparedStatement pstmt = con.prepareStatement("INSERT INTO cafe VALUES(?, ?, ?, ?, ?)");
pstmt.setString(1, "Fiomiccino");
pstmt.setInt(2, 49);
pstmt.setDouble(3, 14.99);
pstmt.setInt(4, 0);
pstmt.setInt(5, 0);
pstmt.addBatch();
...
int [] updateCounts = pstmt.executeBatch(); // executa os comandos em Lote
con.commit();
con.setAutoCommit(true);
```

Nome Café	id_fornecedor	Preço	Vendas	Total
Amaretto	49	9,99	0	0
Amaretto_descafeinado	49	10,99	0	0
Camponesa	49	12,99	0	0
Colombiano	101	7,99	175	50
Colombiano_Descafeinado	101	8,99	155	0
Cubano	49	11,99	0	0
Espresso	150	9,99	60	0
Fiomiccino	49	14,99	0	0
Francês_torrado	49	8,99	150	0
Francês_torrado_descafeinado	49	15,50	90	0
Mineirinho	150	10,99	0	0
Pilar	49	9,99	0	0
Pilar_descafeinado	49	10,99	0	0