

---

# **JDBC**

---

*Sistemas de Informação I*

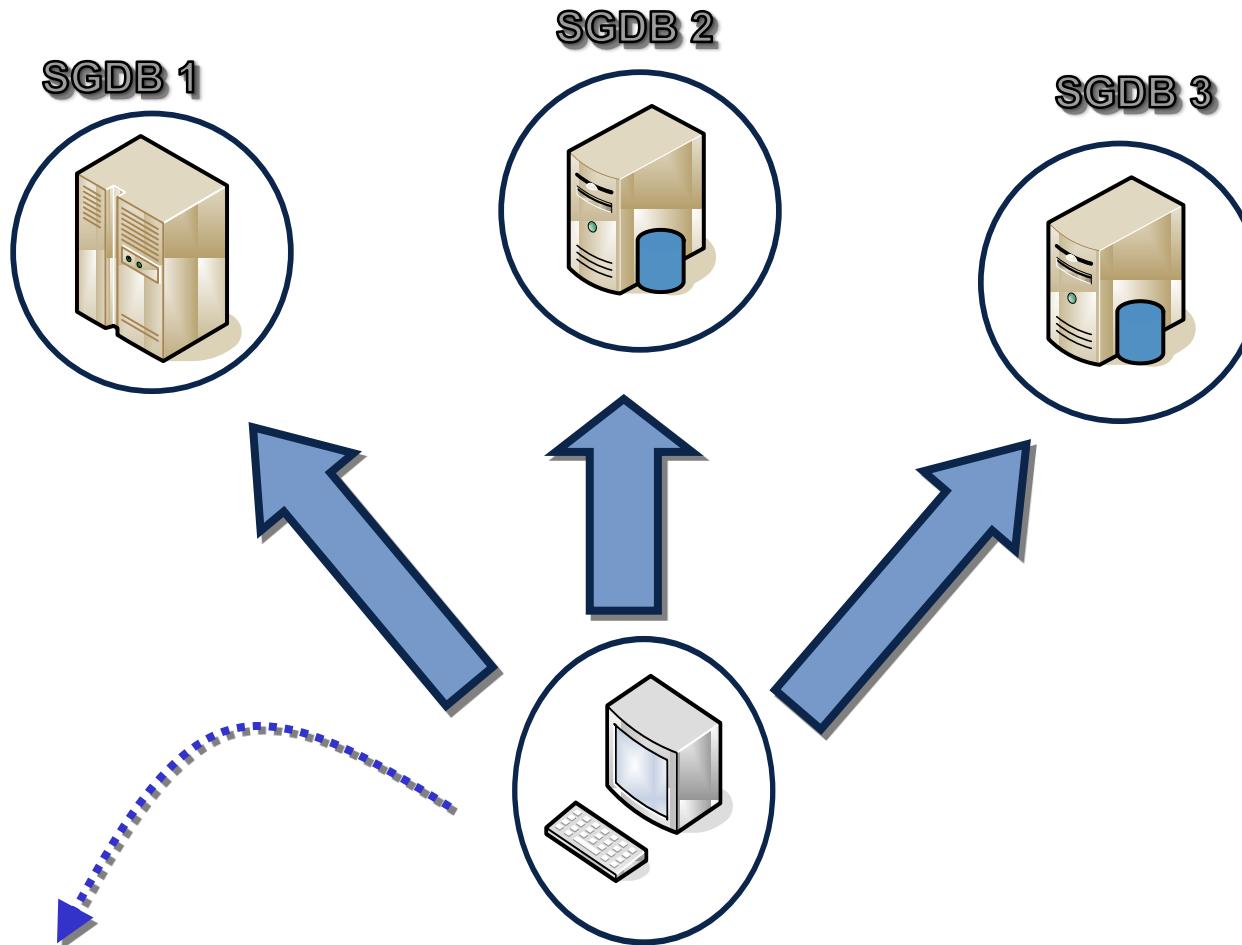
---

# *Agenda*

---

- Evolução tecnológica no acesso a dados
  - ODBC
  - Vista geral sobre o JDBC
  - Principais tipos
  - Execução de comandos
  - Manipulação dos resultados
  - Controlo Transaccional
-

# Evolução da tecnologia de acesso a dados



**Objectivo:** Desenvolver código de acesso a dados independente do SGDB

**Cenário:** Década de 80

# Evolução da tecnologia de acesso a dados

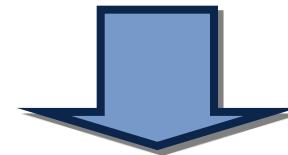
1980

1985

1990

2000

- Não existia uma infra-estrutura de acesso a dados independente do SGBD
- O acesso a diferentes fontes de dados necessitava algum código e muito esforço
- As ferramentas escolhidas eram, principalmente, SQL embutido e pré-processadores



- Tecnologia proprietária
- Flexibilidade limitada

# Evolução da tecnologia de acesso a dados

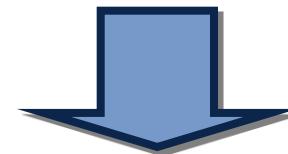
1980

- Como resposta a estes problemas, foi fundado em 1989 um consórcio, *SQL Access Group* (SAG), com o objectivo de fomentar a interoperabilidade e portabilidade entre diferentes SGBD

1990

- Em 1992 surge a primeira definição do ODBC (*Open DataBase Connectivity*) que viria a revolucionar o acesso a dados no mundo PC

1992



2000

- Permitiu que os PC's se tornassem “os clientes” para as aplicações empresariais
- O ponto de viragem

# Evolução da tecnologia de acesso a dados

1980

1990

1996

2000

- O ODBC teve várias versões, a última em 1996 – ODBC 3.0
- Existem várias implementações da especificação
  - *Microsoft ODBC*
  - *iODBC*
  - *UnixODBC*
- O ODBC está demasiado ligado à linguagem SQL, o que levantou problemas no acesso a fontes de dados não relacionais (e.g. Multidimensionais)
- A API não era adequada aos novos paradigmas de programação que estavam a emergir

# Evolução da tecnologia de acesso a dados

1980

- Mas paralelamente ao envolvimento com o *ODBC*, a Microsoft foi lançando API's de acesso a dados, com o objectivo dar suporte a novas necessidades – *DAO* (*DataAccess Objects*) e *RDO* (*Remote Data Objects*)

1990

- *Surge em 1996 o JDBC, como forma standard de aceder a dados em Java*
- O desenvolvimento de aplicações baseada em componentes começava a emergir

1996

2000

# Evolução da tecnologia de acesso a dados

1980

- A primeira versão do JDBC fornecia os mecanismos básicos de acesso a dados
- Em 1999 surge a versão 2 do JDBC, que foi dividida em dois pacotes
  - Funcionalidades *core*
  - Funcionalidades opcionais
- Este lançamento teve em conta principalmente questões de desempenho e adição de funcionalidades às classes existentes, nomeadamente:
  - Cadeias de caracteres com suporte a internacionalização
  - Suporte a diferentes *Timezones*

1990

1999  
2000

# Evolução da tecnologia de acesso a dados

1980

- A versão 3.0 do JDBC é lançada
- Suporta a maior parte da especificação SQL99, além de outras funcionalidades nomeadamente:
  - API de METADATA
  - Alteração de CLOBS (*Character Large OBject*) e BLOBS (*Binary Large Object*)
  - Obter chaves geradas automaticamente na sequência do comando
  - Suporte a Múltiplos resultados
  - *Connection pooling*

1990

2000  
2001

# Evolução da tecnologia de acesso a dados

1980

- A versão 4.0 do JDBC é lançada
- Suporte muitas das funcionalidades presentes no standard SQL2003

1990

- Das novas funcionalidades, destacam-se :
  - Carregamento automático dos drivers
  - Gestão melhoradas das ligações
  - Suporte ao tipo de dados RowId
  - Suporte para o tipo de dados XML
  - Suporte ao conceito de Dataset, com recurso a anotações.

2000

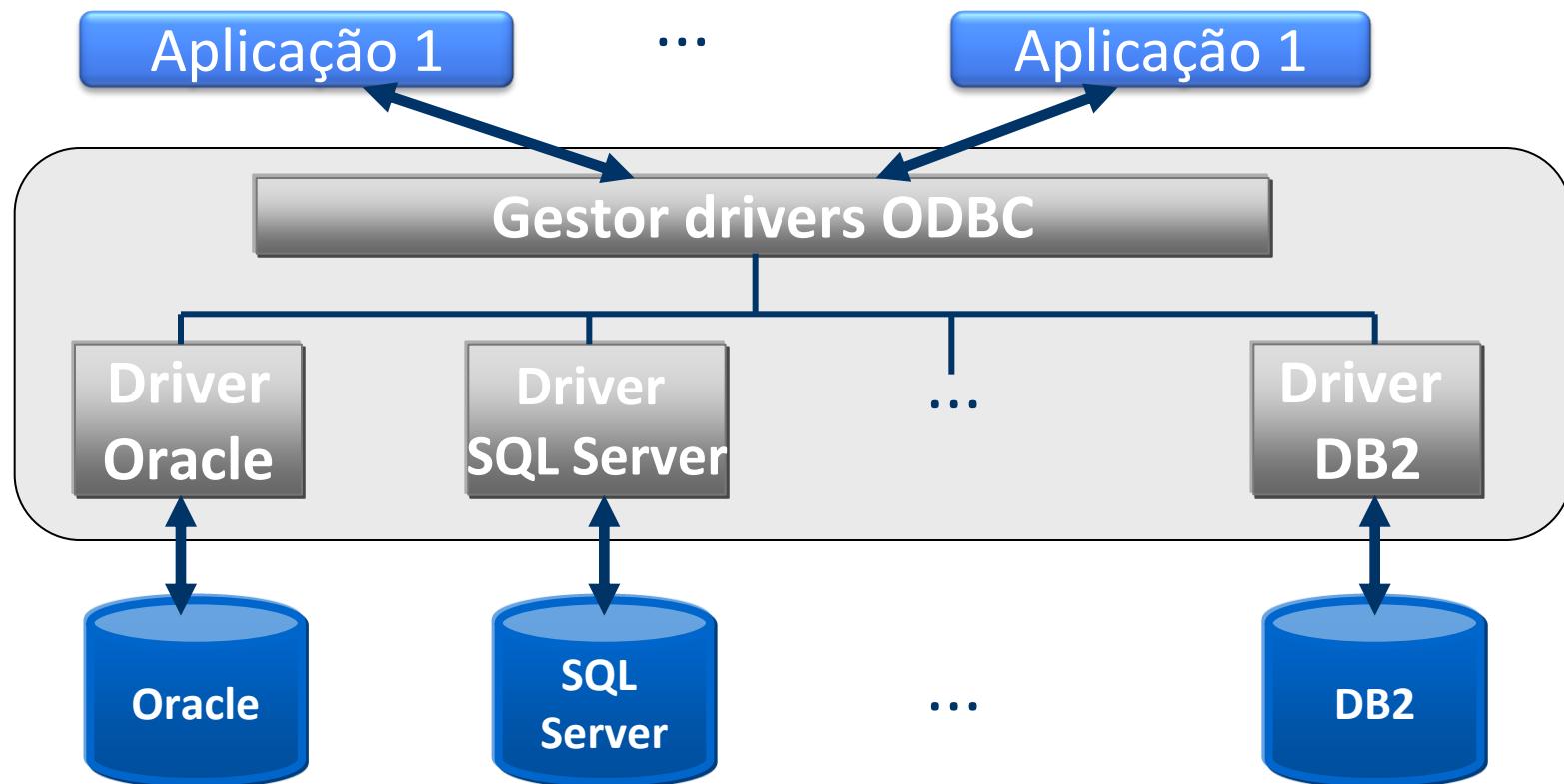
2006

## *ODBC*

---

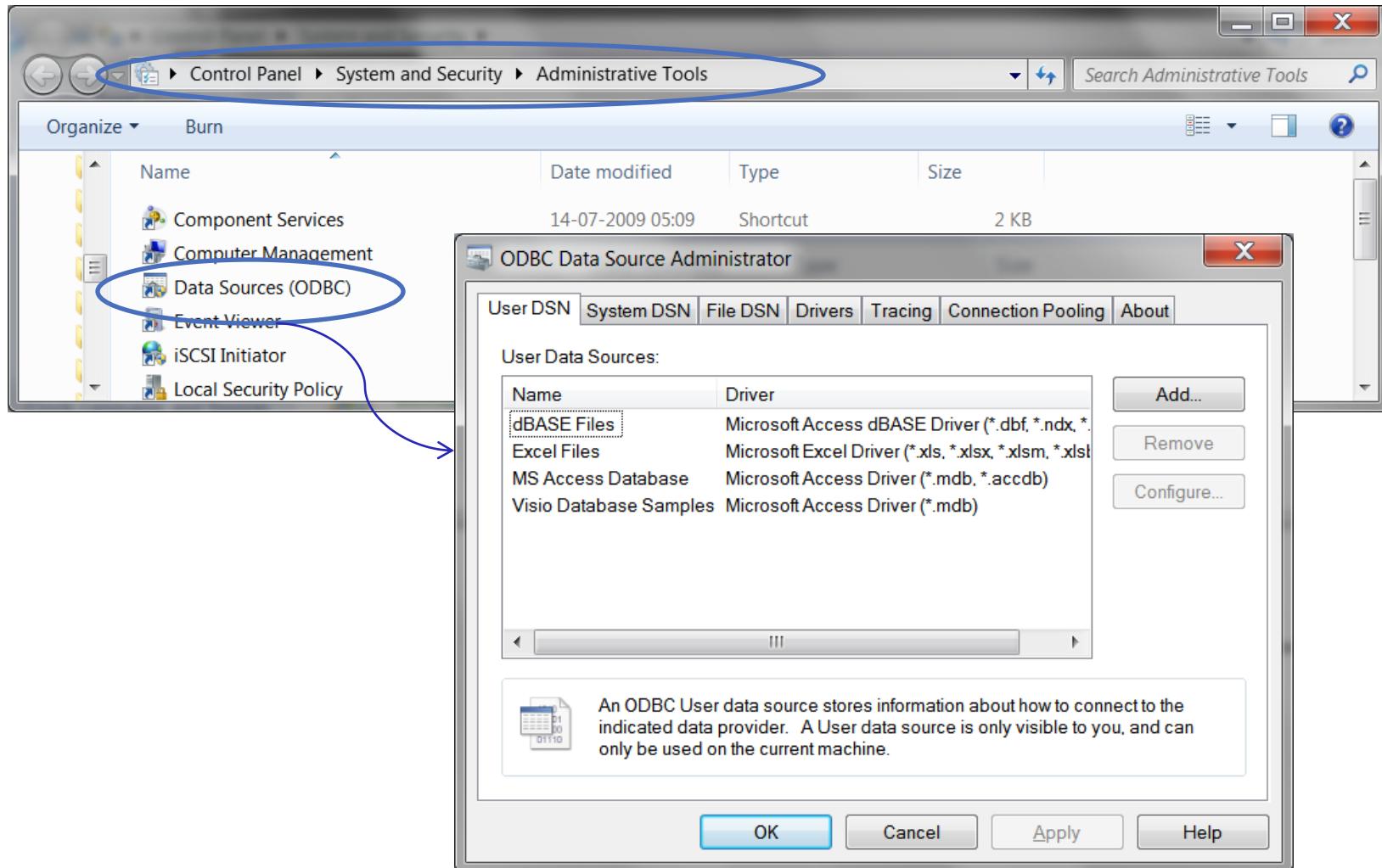
- O ODBC é um standard de acesso a Bases de dados Heterogéneas
- Através de drivers ODBC pode aceder-se a diversos repositórios, tais como SQL Server, DB2, Oracle, MySql, Excel, texto, etc.
- A linguagem de manipulação de dados usada é sempre o SQL
- Para aceder a um determinada BD é necessário criar uma “fonte de dados ODBC”, que se designa de *DataSource ODBC*, na máquina onde se quer aceder ao repositório de dados

## *ODBC (cont.)*



# ODBC (cont.)

- Nas máquinas Windows, o acesso ao gestor de drivers é feito através do Painel de controlo

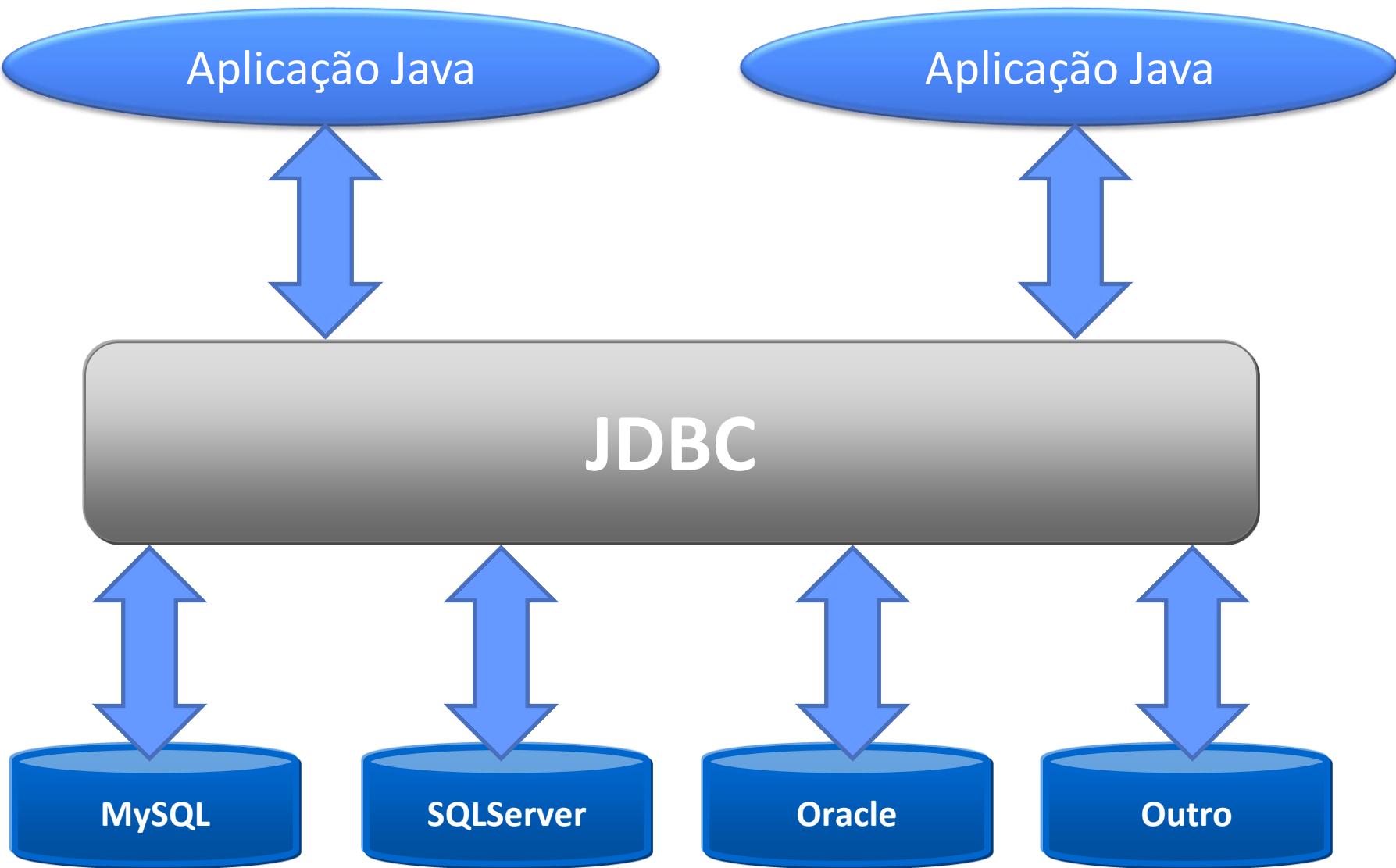


# *O que é o JDBC?*

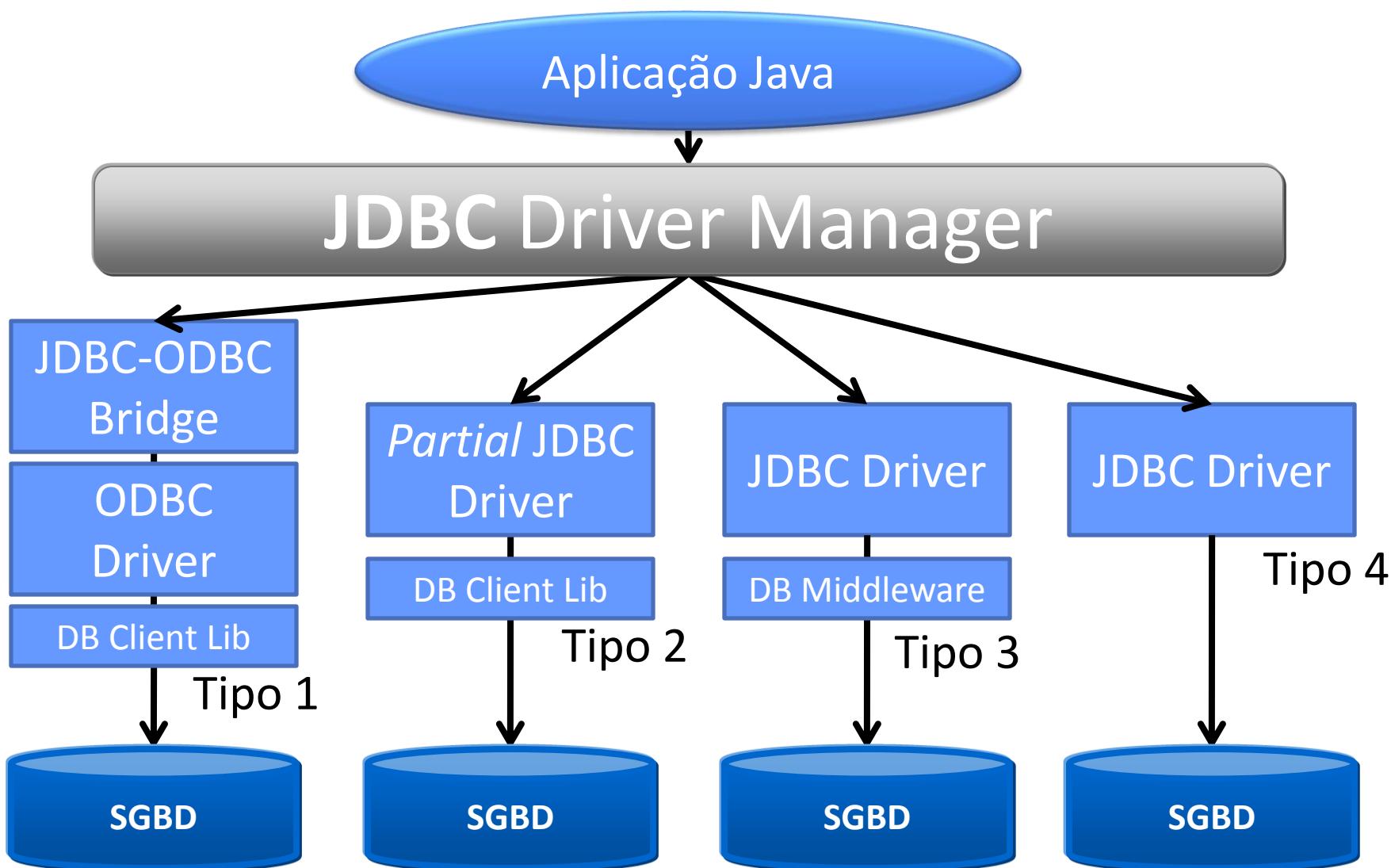
---

- JDBC é um API baseada no *X/Open SQL CLI*, que permite a execução de comandos SQL. Possibilita:
  - Estabelecer uma ligação com a base de dados
  - Utilizar essa ligação para executar comandos SQL
  - Processar os resultados devolvidos pela base de dados
- Foi desenhada tendo em conta:
  - Ser suportada nas versões Java EE e Java SE
  - Ser consistente com a norma SQL2003
  - Oferecer acesso a a das independente do fabricante do SGBD
  - Manter o enfoque em dados tabulares
  - Ser simples

# *Arquitectura JDBC*



## Drivers JDBC



## JDBC 4.0

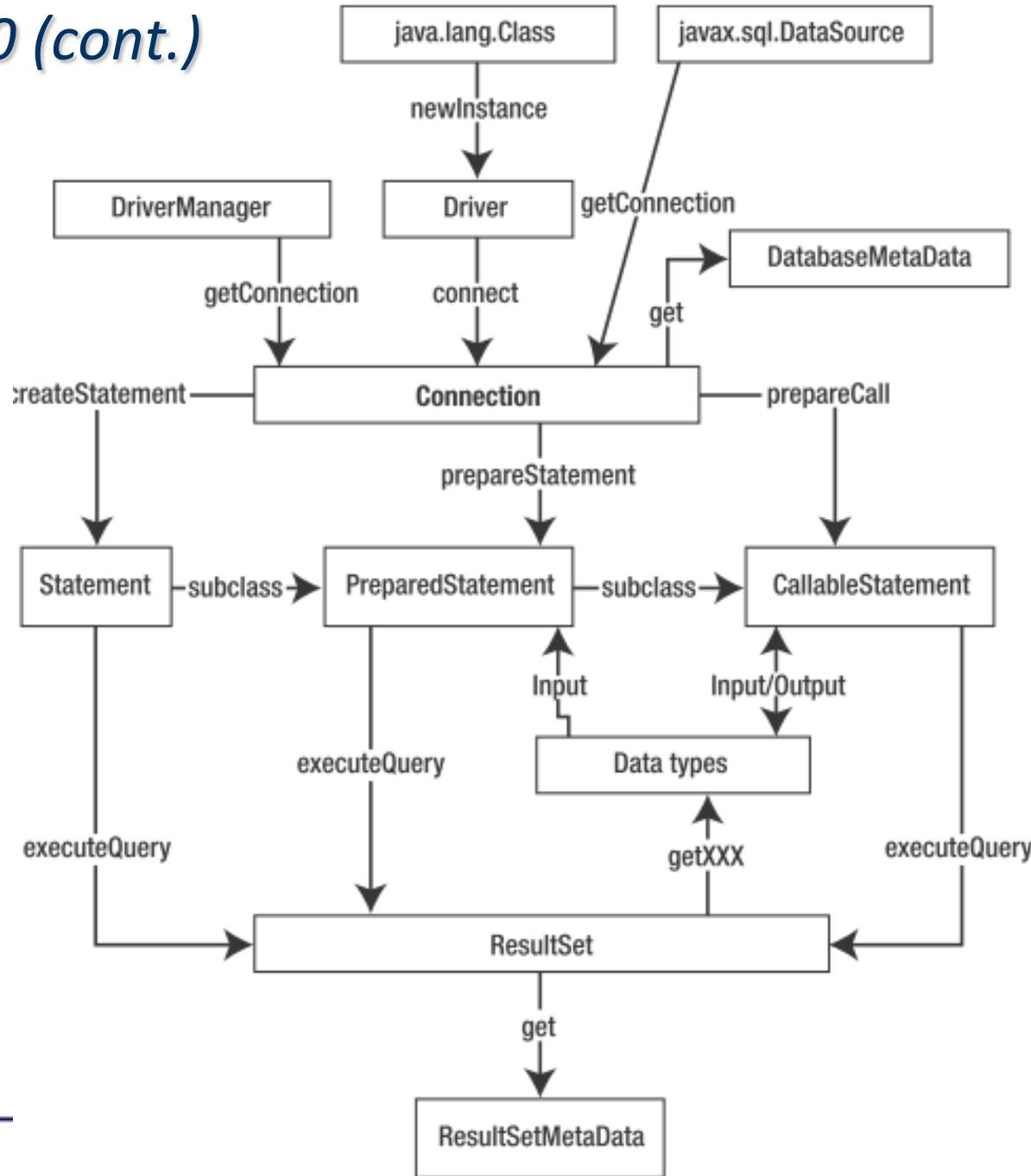
- É parte integrante do Java SE 6.0 e inclui dois pacotes:
  - **JDBC 4.0 Core API**, disponível através do pacote `java.sql`
  - **JDBC 4.0 Standard Extension API**, através do pacote `javax.sql`. É necessária para aplicações que utilizem *connection pooling*, transacções distribuídas , entre outros
- As classes principais do `java.sql` são:
  - **DriverManager**, responsável por carregar os drivers JDBC drivers em memória. Também disponibiliza métodos para criar ligações para fontes de dados
  - **Connection**, interface que disponibiliza os métodos necessário para realizar uma ligação à fonte de dados, permitindo criar um conjunto de objectos necessários à execução de comandos

## *JDBC 4.0 (cont.)*

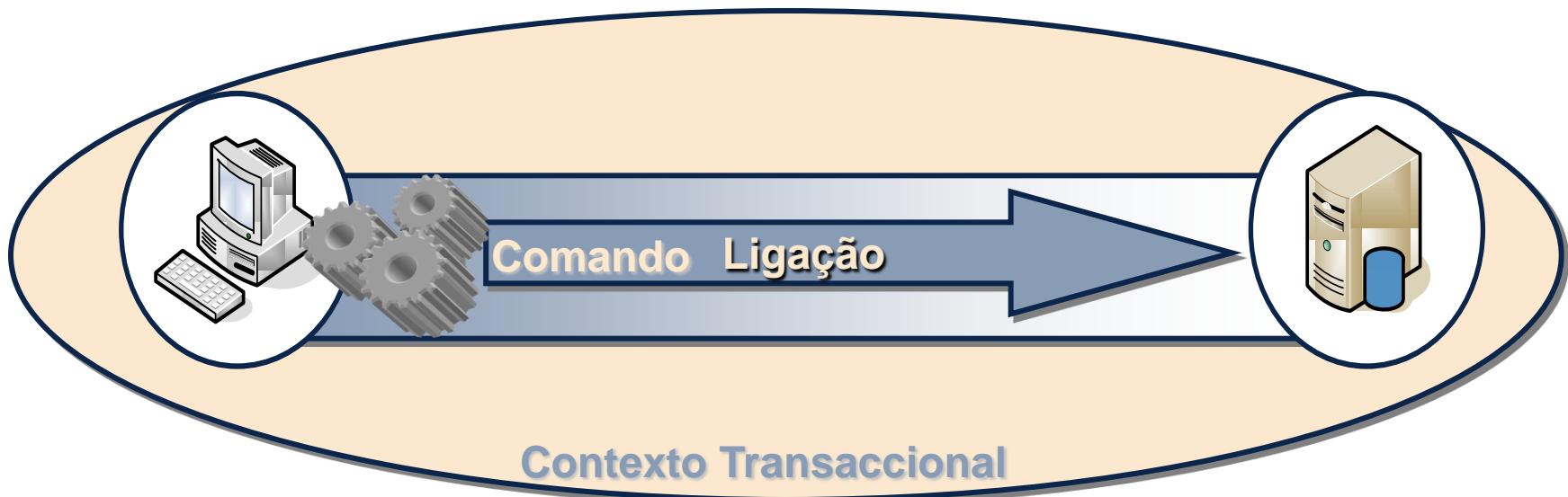
---

- As classes principais do `java.sql` (cont.):
  - **Statement**, interface que permite a execução de comandos SQL estáticos
  - **PreparedStatement**, interface que permite a manipulação de comandos SQL pré compilados
  - **CallableStatement**, é uma interface que representa um procedimento armazenado
  - **ResultSet**, é uma interface que representa o resultado gerado pelo SGBD em resposta a um comando SELECT
  - **SQLException**: é uma classe que permite o acesso aos erros gerados durante a execução de um comando SQL

## JDBC 4.0 (cont.)



## *Sequência típica de acções*



1. Estabelecer uma ligação à fonte de dados
2. Especificar um comando a ser efectuado
  - Opcionalmente pode ser necessário utilizar parâmetros
3. Se necessário manipular transacções
4. Executar o comando
5. Caso o comando produza um conjunto de dados, manipular o resultado
6. Se existir um contexto transaccional activo, indicar o sucesso da acção
7. Fechar a ligação

# *Primeiro passo!*

- Antes de iniciar qualquer execução de comandos SQL é necessário carregar o driver JDBC que iremos utilizar

```
//Driver JDBC-ODBC Bridge  
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

```
//Driver Tipo 4 para SQLServer  
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

- Por omissão apenas são distribuídos drivers JDBC–ODBC Bridge
- Para utilizar outros drivers será necessário obter junto dos fabricantes do SGBD, uma implementação de um driver específico
  - Microsoft SQL server (<http://msdn.microsoft.com/en-us/data/aa937724.aspx>)
  - MySql (<http://www.mysql.com/products/connector/>)

# Hello World

```
...
//Driver ODBC genérico
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

//Estabelecer a ligação
String url = "jdbc:odbc:jdbcdemo1";
Connection con = DriverManager.getConnection(url);

//Executar o comando
Statement stmt = con.createStatement();
ResultSet rs=stmt.executeQuery("select value from JDBCDEMO");

//Iterar no resultado
while(rs.next())
    System.out.print(rs.getString("value"));
System.out.println();

//Libertar recursos do ResultSet
rs.close();
//Libertar recursos do Statement
stmt.close();
//Fechar a ligação
con.close();

...
```

## *Definição de uma data source ODBC*

---

- No código anterior, "jdbc:odbc:jdbcdemo1" inclui uma parte que é estática e específica do driver JDBC para OBDC, "jdbc:odbc: ", e uma parte variável
  - Indica o nome de uma fonte de dados ODBC
- Uma fonte de dados ODBC contém informação sobre o repositório que se quer aceder, nomeadamente:
  - O seu nome
  - Credenciais de acesso
  - Base de dados por omissão
- Note-se que a fonte de dados ODBC tem de existir no momento em que o programa é executado.

## *Criação de Statement*

- A execução de comandos é feita utilizando implementação da interface **Statement**
  - **Statement**, para execução de instruções DDL/DML
  - **PreparedStatement**, para execução de comandos preparados
  - **CallableStatement**, para procedimentos armazenados
    - não é tratado no âmbito da UC de SI1
- Essa implementação é obtida através de uma ligação (objecto **Connection**), que na sua forma mais simples fica:

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

- Obtemos um **Statement** que é capaz de executar comandos de DDL (**create**, **alter** e **drop**) e DML (**select**, **insert**, **update** e **delete**). No entanto, os resultados produzidos pelas instruções **select**, apenas podem ser percorridos uma única vez

## *Execução de comandos*

- A execução de comandos é feita através de um conjunto de métodos disponíveis em **Statement**, que devem ser escolhidos de acordo com a instrução a executar

`ResultSet executeQuery(String sql)`

- Permite a execução de uma única instrução de SELECT

`int executeUpdate(String sql)`

- Permite executar instruções de DDL e `insert`, `update` e `delete`. No primeiro caso é devolvido -1, nos restantes é devolvido o número de tuplos afectados

`boolean execute(String sql)`

- Permite a execução da(s) instrução(ões) SQL passadas no parâmetro. É possível serem retornados múltiplos **ResultSet**. Devolve `true` se o primeiro objecto for um **ResultSet**, `false` caso contrário

## *Execução de comandos - executeQuery*

- A execução de uma interrogação através do método `executeQuery` produz um `ResultSet` que necessita de ter uma ligação aberta para aceder aos resultados

```
...
Statement stmt = con.createStatement();
//executar o comando
ResultSet rs=stmt.executeQuery("select value from JDBCDEMO");
...
```

- É lançada uma excepção do tipo `SQLException` sempre que existir um erro no acesso à base de dados

## Execução de comandos - executeUpdate

```
...
final static String cmdUpt =
"update JDBCDEMO set value='Teste do executeUpdate';";
final static String cmdIns =
"insert into JDBCDEMO select max(id)+1, 'Teste Insert' from JDBCDEMO;";
final static String cmdDDL =
"create table T(i int); drop table T;";

...
//executar o comando de update
int nTuplos = stmt.executeUpdate(cmdUpt);
System.out.println(nTuplos); // numero de tuplos da tabela

//executar o comando de inserção
nTuplos = stmt.executeUpdate(cmdIns);
System.out.println(nTuplos); // 1

//executar o comando de DDL
nTuplos = stmt.executeUpdate(cmdDDL);
System.out.println(nTuplos); // -1.

...
...
```

## Execução de comandos - execute

```
...
final static String cmdUpt =
"update JDBCDEMO set value='Teste do execute ';select * from JDBCDEMO;";
final static String cmdIns =
"insert into JDBCDEMO select max(id)+1,'Teste Insert' from JDBCDEMO;";
final static String cmdSel =
"SELECT id from JDBCDEMO;select value from JDBCDEMO";
...
//executar o comando de select
boolean res = stmt.execute(cmdSel);
System.out.println(res); // true

//executar o comando de insert
res = stmt.execute(cmdIns);
System.out.println(res); // false

//executar o comando de Update seguido de um select
res = stmt.execute(cmdUpt);
System.out.println(res); // false
...
```

# *Execução de comandos parametrizados*

- Quando os comandos a executar contém informação que é recolhida dos utilizadores, é essencial utilizar comandos parametrizados

Por questões de robustez e segurança



- Um comando parametrizado:
  - Tem uma instrução SQL independente do valor específico para as partes variáveis
  - Obriga a que sejam criados parâmetros, indicando os seus tipos e valores
- Em JDBC, a execução de comandos parametrizados é feito recorrendo ao tipo PreparedStatement

## Execução de comandos parametrizados (cont.)

- Em primeiro lugar é necessário definir uma instrução SQL que suporte parâmetros

```
String cmdIns = "insert into JDBCDEMO values(?,?)";
```

- No local de cada valor é colocado o carácter ‘?’
- Seguidamente, obtém-se o comando preparado para a instrução pretendida

```
PreparedStatement pstmt = con.prepareStatement(cmdIns);
```

- Posteriormente é necessário especificar o tipo e o valor para cada um dos parâmetros, indicando o seu número de ordem
  - Esta acção é efectuada utilizando um conjunto de métodos, da forma *SetTipo*, que recebem o número do parâmetro e o valor
    - Onde Tipo são todos os tipos suportados pelo JDBC

## *Execução de comandos parametrizados (cont.)*

- Para o exemplo, temos 2 parâmetros
  - Um inteiro
  - Uma cadeia de caracteres variável

```
"insert into JDBCDEMO values( ? , ? );"
```

- Ou seja,

```
stmt.setInt(1, 10);  
stmt.setString(2,"teste com parâmetros");
```

- Finalmente executasse o comando

```
stmt.executeUpdate();
```

```
insert into JDBCDEMO values( 10 , 'teste com parâmetros' )
```

## *Considerações sobre execução de comandos*

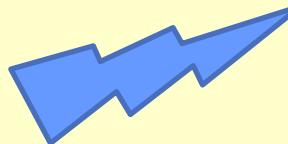
- Devem ser utilizados os métodos adequados ao tipo de instrução SQL a executar por questão de desempenho
- Um objecto do tipo Statement apenas pode ter um ResultSet “aberto” de cada vez

```
//executar o comando de select  
ResultSet rs = stmt.executeQuery(cmdSel);
```

```
//executar o comando de insert  
stmt.executeUpdate(cmdIns);
```

```
//acceder ao ResultSet  
rs.next(); //gera exceção
```

ResultSet is closed



- Depois de utilizado, o objecto Statement deve ser fechado para libertar os recursos por ele detidos
- Sempre que um comando inclui informação inserida pelo utilizador, utilizar sempre um comando parametrizado

## *Manipulação de ResultSets*

---

- A execução de comandos de SELECT produzem dados cuja manipulação é feita através de um objecto ResultSet
- Um RecordSet não guarda, geralmente, os resultados da interrogação em memória, mantendo um cursor cujas características foram definidas pelo Statement
- Na maioria das implementações, a execução da interrogação é feita apenas quando os resultados são necessário, e.g., quando o método next( ) é executado pela primeira vez
- E de cada chamada ao next( ), é obtido um tuplo
  - Directamente do SGDB, ou
  - Se o driver suportar, de uma cache local, que foi populada com um conjunto de tuplos

## *Manipulação de ResultSets (cont.)*

- A iteração sobre os resultados é feita, para os RecordSets FORWARD\_ONLY, através do método `next()`, que coloca disponível o próximo tuplo
- Antes do primeiro acesso aos dados é sempre necessário evocar o `next()`

```
...
//iterar no resultado
while(rs.next())
{...}
...
```

- O acesso a cada coluna do tuplo corrente é feita através de um conjunto de métodos `getTipo`, onde tipo depende do *tipo* de cada coluna retornada

## Manipulação de ResultSets (cont.)

```
...
ResultSet rs=stmt.executeQuery("select id,value from JDBCDEMO");
//iterar no resultado
while(rs.next())
    System.out.format("%d \t %s\n",
                      rs.getInt(1),
                      rs.getString("value"));
...
...
```

Os índices iniciam-se em 1!



- O acesso a cada coluna pode ser feito pelo seu número
  - `rs.getInt(1)`
- Ou pelo seu nome (*case insensitive*)
  - `rs.getString("value")`
- Em qualquer dos casos é necessário utilizar o método `get` de acordo com o tipo da coluna a aceder. Por questões de portabilidade, os valores devem ser lidos por ordem e apenas uma única vez

## *Manipulação de ResultSets (cont.)*

- Quando a interrogação produz um resultado com colunas com o mesmo nome, o acesso através do nome devolve o valor da primeira coluna apenas

A utilização do índice em vez do nome é mais eficiente e mais geral



- Quando a método get utilizado não é o adequado, tendo em conta o tipo da coluna, é feita uma tentativa de coerção
  - Embora possível, o desempenho é menor e deve sempre ser utilizado o método mais adequado
- De seguida é apresentada uma tabela onde nas linhas temos os métodos get disponíveis e nas colunas os tipos SQL suportados:
  - x – indica que a conversão é possível
  - X – indica que deve ser utilizado o método get para obter valores desse tipo

Fonte:<http://java.sun.com/javase/6/docs/technologies/guides/jdbc/getstart/resultset.html#1012368>

|                    | TINYINT | SMALLINT | INTEGER | BIGINT | REAL | FLOAT | DOUBLE | DECIMAL | NUMERIC | BIT | CHAR | VARCHAR | LONGVARCHAR | BINARY | VARBINARY | LONGVARBINARY | DATE | TIME | TIMESTAMP | CLOB | BLOB | ARRAY | REF | STRUCT | JAVA_OBJECT |  |  |
|--------------------|---------|----------|---------|--------|------|-------|--------|---------|---------|-----|------|---------|-------------|--------|-----------|---------------|------|------|-----------|------|------|-------|-----|--------|-------------|--|--|
| getByte            | X       | x        | x       | x      | x    | x     | x      | x       | x       | x   | x    | x       | x           | x      | x         | x             | x    | x    | x         | x    | x    | x     | x   | x      |             |  |  |
| getShort           | x       | X        | x       | x      | x    | x     | x      | x       | x       | x   | x    | x       | x           | x      | x         | x             | x    | x    | x         | x    | x    | x     | x   | x      |             |  |  |
| getInt             | x       | x        | X       | x      | x    | x     | x      | x       | x       | x   | x    | x       | x           | x      | x         | x             | x    | x    | x         | x    | x    | x     | x   | x      |             |  |  |
| getLong            | x       | x        | x       | X      | x    | x     | x      | x       | x       | x   | x    | x       | x           | x      | x         | x             | x    | x    | x         | x    | x    | x     | x   | x      |             |  |  |
| getFloat           | x       | x        | x       | x      | X    | x     | x      | x       | x       | x   | x    | x       | x           | x      | x         | x             | x    | x    | x         | x    | x    | x     | x   | x      |             |  |  |
| getDouble          | x       | x        | x       | x      | x    | X     | X      | x       | x       | x   | x    | x       | x           | x      | x         | x             | x    | x    | x         | x    | x    | x     | x   | x      |             |  |  |
| getBigDecimal      | x       | x        | x       | x      | x    | x     | X      | X       | x       | x   | x    | x       | x           | x      | x         | x             | x    | x    | x         | x    | x    | x     | x   | x      |             |  |  |
| getBoolean         | x       | x        | x       | x      | x    | x     | x      | x       | x       | X   | x    | x       | x           | x      | x         | x             | x    | x    | x         | x    | x    | x     | x   | x      |             |  |  |
| getString          | x       | x        | x       | x      | x    | x     | x      | x       | x       | x   | X    | X       | x           | x      | x         | x             | x    | x    | x         | x    | x    | x     | x   | x      |             |  |  |
| getBytes           |         |          |         |        |      |       |        |         |         |     |      |         | X           | X      | x         |               |      |      |           |      |      |       |     |        |             |  |  |
| getDate            |         |          |         |        |      |       |        |         |         |     |      |         | x           | x      | x         |               |      | X    | x         |      |      |       |     |        |             |  |  |
| getTime            |         |          |         |        |      |       |        |         |         |     |      |         | x           | x      | x         |               |      |      | X         | x    |      |       |     |        |             |  |  |
| getTimestamp       |         |          |         |        |      |       |        |         |         |     |      |         | x           | x      | x         |               |      | x    | x         | X    |      |       |     |        |             |  |  |
| getAsciiStream     |         |          |         |        |      |       |        |         |         |     |      |         | x           | x      | X         | x             | x    | x    |           |      |      |       |     |        |             |  |  |
| getUnicodeStream   |         |          |         |        |      |       |        |         |         |     |      |         | x           | x      | X         | x             | x    | x    |           |      |      |       |     |        |             |  |  |
| getBinaryStream    |         |          |         |        |      |       |        |         |         |     |      |         | x           | x      | X         |               |      |      |           |      |      |       |     |        |             |  |  |
| getClob            |         |          |         |        |      |       |        |         |         |     |      |         |             |        |           |               |      |      |           | X    |      |       |     |        |             |  |  |
| getBlob            |         |          |         |        |      |       |        |         |         |     |      |         |             |        |           |               |      |      |           |      | X    |       |     |        |             |  |  |
| getArray           |         |          |         |        |      |       |        |         |         |     |      |         |             |        |           |               |      |      |           |      |      | X     |     |        |             |  |  |
| getRef             |         |          |         |        |      |       |        |         |         |     |      |         |             |        |           |               |      |      |           |      |      |       | X   |        |             |  |  |
| getCharacterStream |         |          |         |        |      |       |        |         |         |     |      |         |             |        |           | x             | x    | X    | x         | x    | x    |       |     |        |             |  |  |
| getObject          | x       | x        | x       | x      | x    | x     | x      | x       | x       | x   | x    | x       | x           | x      | x         | x             | x    | x    | x         | x    | x    | x     | x   | x      | X           |  |  |

## *Manipulação de ResultSets (cont.)*

- Por omissão, os ResultSets obtidos de um Statement são *FORWARD ONLY* e *READ ONLY*
- Podemos indicar que pretendemos um Statement que produza resultSets com outras características

```
Statement createStatement( int resultSetType,  
                           int resultSetConcurrency)
```

- Onde: resultSetType pode ser:
  - `ResultSet.TYPE_FORWARD_ONLY`, permite a iteração nos resultados devolvidos pela execução de um comando SQL
  - `ResultSet.TYPE_SCROLL_INSENSITIVE`, permite a navegação nos resultados, insensível a alterações na Base de dados
  - `ResultSet.TYPE_SCROLL_SENSITIVE`, torna visíveis as alterações efectuadas na Base de dados enquanto o RecordSet estiver aberto

## *Manipulação de ResultSets (cont.)*

```
Statement createStatement( int resultSetType,  
                           int resultSetConcurrency)
```

- Onde: resultSetConcurrency pode ser:
  - ResultSet.CONCUR\_READ\_ONLY, os dados obtidos apenas podem ser lidos
  - ResultSet.CONCUR\_UPDATABLE, os dados obtidos podem ser modificados
- No âmbito da UC de SI1, o valor deste parâmetro deve ser ResultSet.CONCUR\_READ\_ONLY

## *Manipulação de valores NULL*

- Em certas situações, ás interrogações contém colunas com valores a null
  - Se os campos forem de um tipo nativo, o null é convertido para o valor por omissão desse tipo. Por exemplo 0 para os casos dos inteiros.
  - Se for um tipo referência, irá fica com o valor null
    - Note-se que este comportamento depende da implementação do driver.
- Para garantir que os dados apresentados são consistentes com o valor existente no SGBD, devem ser testados os valores que podem ser null, e tratados convenientemente

```
int id = rs.getInt(1);
boolean res = rs.wasNull(); //determina se o valor
                           anteriormente devolvido é null
```

# *Obter informação sobre as colunas de um ResultSet*

- Um ResultSet disponibiliza um conjunto de informação através do tipo ResultSetMetaData, nomeadamente
  - O número de colunas

```
ResultSetMetaData meta = rs.getMetaData();
int columnsCount = meta.getColumnCount();
```

- O seu nome ( no formato sugerido para apresentação)

```
String name = meta.getColumnLabel(index);
```

- O seu tipo

```
String name = getColumnTypeName(index) //e.g. VARCHAR
//ou
int type = meta.getColumnType(index)
```

Ver `java.sql.Types`

## *Controlo transaccional*

- Por omissão, cada execução de um comando está em modo *auto commit*, ou seja, se não acontecer nenhum erro durante a execução, os resultados tornam-se persistentes no SGBD no final da execução
- Para garantirmos que a execução de um conjunto de comandos é atómica, com a semântica de tudo ou nada, é necessário desligar o modo de *auto commit*
- Após o conjunto de comandos terem sido executados, é necessário indicar o termo da transacção, com *commit* ou *rollback*

Com o *auto commit* inibido, se não for explicitamente evocado o *commit*, os dados nunca ficam persistentes não SGBD



- É através do tipo **Connection** que todo este controlo é feito

## Controlo transaccional (cont.)

```
...
con.setAutoCommit(false);

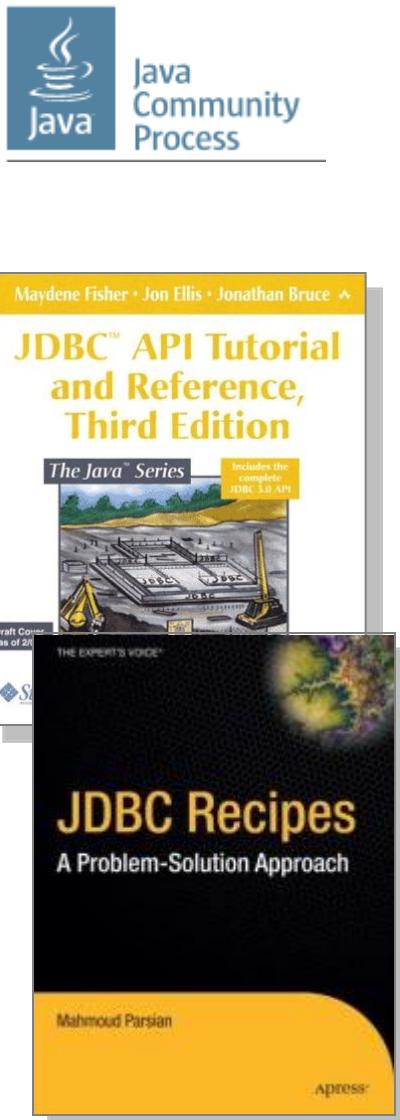
// várias inserções
for (int i = 0; i < 4; i++)
    stmt.executeUpdate("insert into JDBCDEMO
                        select max(id)+1,'teste'
                        from JDBCDEMO;");

//terminar a transacção corrente
con.commit();

//voltar a ligar o auto commit
con.setAutoCommit(true);
...
```

Contexto  
transaccional

# Bibliografia



**SR 221: JDBC™ 4.0 API Specification,**  
<http://jcp.org/en/jsr/detail?id=221>, Sun Microsystem

**JDBC(TM) API Tutorial and Reference, (3rd Edition)**  
*Maydene Fisher, Jon Ellis, Jonathan Bruce*  
*Prentice Hall, 2003*

**JDBC Recipes: A Problem-Solution Approach**  
*Mahmoud Parsian*  
*Apress, 2005*