

# **Programação de Sistemas Distribuídos - Java para Web**

## **3 - Acesso a base de dados com JDBC**

### **Citeforma**

Jose Aser Lorenzo, Pedro Nunes, Paulo Jorge Martins

[jose.l.aser@sapo.pt](mailto:jose.l.aser@sapo.pt), [pedro.g.nunes@gmail.com](mailto:pedro.g.nunes@gmail.com),  
[paulojsm@gmail.com](mailto:paulojsm@gmail.com)

Dezembro 2013

V 1.4

## Sumário

<b>3- Acesso a base de dados com JDBC .....</b>	<b>3-4</b>
<b>3.1- Objectivos.....</b>	<b>3-5</b>
<b>3.2- Preparar o ambiente de trabalho.....</b>	<b>3-6</b>
<b>3.3- JDBC e tipos de drivers .....</b>	<b>3-13</b>
3.3.1- O que é o JDBC? .....	3-14
3.3.2- Como um cliente liga ao SGBD Oracle? .....	3-16
3.3.2.1- Driver Tipo I: JDBC-ODBC .....	3-17
3.3.2.2- Driver Tipo II: JDBC-OCI .....	3-19
3.3.2.3- Driver Oracle Tipo IV: JDBC Thin.....	3-22
3.3.3- Como um cliente liga ao SGBD MySQL?.....	3-25
3.3.4- Executar SELECT .....	3-27
3.3.5- Exercícios .....	3-29
3.3.5.1- Exercício – Driver Oracle Tipo I: JDBC-ODBC .....	3-29
3.3.5.2- Exercício – Driver Oracle Tipo II: JDBC-OCI .....	3-31
3.3.5.3- Exercício – Driver Oracle Tipo IV: JDBC Thin .....	3-32
3.3.5.4- Exercício – Driver MySQL Tipo IV: JDBC Thin.....	3-34
<b>3.4- Executar DDL e DML .....</b>	<b>3-36</b>
3.4.1- Comandos DDL – Data Definition Language .....	3-37
3.4.2- Comandos DML – Data Manipulation Language.....	3-38
3.4.2.1- Noção de transação .....	3-38
3.4.2.2- Controlo de transações .....	3-39
3.4.3- Exercícios .....	3-41
3.4.3.1- Exercício – Criar Tabela .....	3-41
3.4.3.2- Exercício – INSERT com Commit e Rollback .....	3-43
3.4.3.3- Exercício – UPDATE de várias linhas.....	3-45
3.4.3.4- Exercício – DELETE com Rollback .....	3-46
3.4.3.5- Exercício – Metadata.....	3-48
<b>3.5- PreparedStatement.....</b>	<b>3-50</b>
3.5.1- Porquê usar <i>PreparedStatement</i> ?.....	3-51
3.5.1.1- Vantagens e inconvenientes .....	3-53
3.5.1.2- Select com PreparedStatement .....	3-54
3.5.2- Exercícios .....	3-55
3.5.2.1- Exercício – Sem PreparedStatement .....	3-55
3.5.2.2- Exercício – Com PreparedStatement.....	3-57
3.5.2.3- Verificar a redução de “parsing” .....	3-58
<b>3.6- Executar <i>stored procedure</i> e <i>function</i> .....</b>	<b>3-60</b>
3.6.1- Como invocar uma “ <i>stored procedure</i> ”?.....	3-61
3.6.2- Como invocar uma função? .....	3-62
3.6.3- Exercícios .....	3-63
3.6.3.1- Exercício – Criar uma “ <i>stored procedure</i> ” .....	3-63
3.6.3.2- Exercício – Invocar a “ <i>stored procedure</i> ”.....	3-64
<b>3.7- SQL Injection.....</b>	<b>3-67</b>
3.7.1- O problema .....	3-68
3.7.2- A solução .....	3-70
3.7.3- Exercícios .....	3-71
3.7.3.1- Adicionar a coluna password à tabela Employees .....	3-71
3.7.3.2- Exercício - Criar a classe que autentica o utilizador usando código vulnerável .....	3-72
3.7.3.3- Exercício - Corrigir as vulnerabilidades.....	3-75

<b>3.8- Segurança dos dados – Cifra .....</b>	<b>3-78</b>
3.8.1- Cifra de dados .....	3-79
3.8.2- Exemplo .....	3-80
3.8.3- Exercício – Cifrar uma String .....	3-81



# Programação de Sistemas Distribuídos - Java para Web

---

## Capítulo 3 - Acesso a base de dados com JDBC

José Aser Lorenzo  
Pedro Nunes  
Paulo Jorge Martins



Java Web  
© Citeforma

### 3- Acesso a base de dados com JDBC

JDBC é a abreviatura de Java DataBase Connectivity e é uma das formas possíveis de um programa Java interagir com uma base de dados.

## Objetivos

- Usar o JDBC em programas Java para:
  - Consultar, inserir, alterar e eliminar dados armazenados numa **Base de Dados** – Oracle ou MySQL;
  - Utilizar “*Statements*” e “*Prepared Statements*”;
  - Invocar “*Stored Procedures*” e “*Functions*”.



### 3.1- Objectivos

No fim deste capítulo o formando estará apto a usar o JDBC em programas Java para os objectivos acima propostos.

## Sumário

- Preparar o ambiente de trabalho;
- JDBC e tipos de *drivers*;
- Executar SELECT;
- Executar DDL e DML;
- *PreparedStatement*;
- Executar *stored procedure* e *function*;
- SQL *Injection*
- Segurança dos dados - Cifra



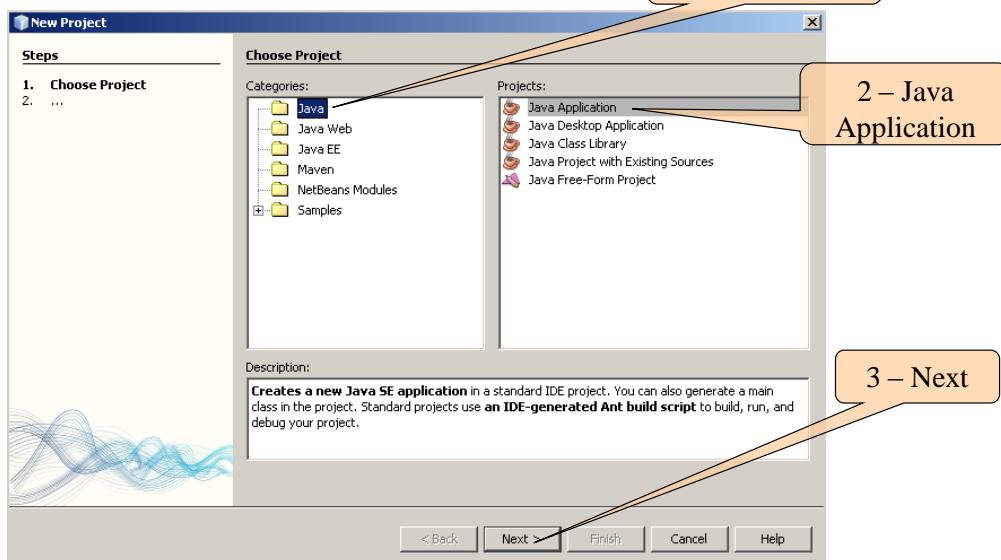
### 3.2- Preparar o ambiente de trabalho

Vamos preparar o ambiente de trabalho para o curso de Programação em Java – Aplicações Web, e em particular para este capítulo de JDBC.

## NetBeans - Criar um projecto

#1/3

File → New Project



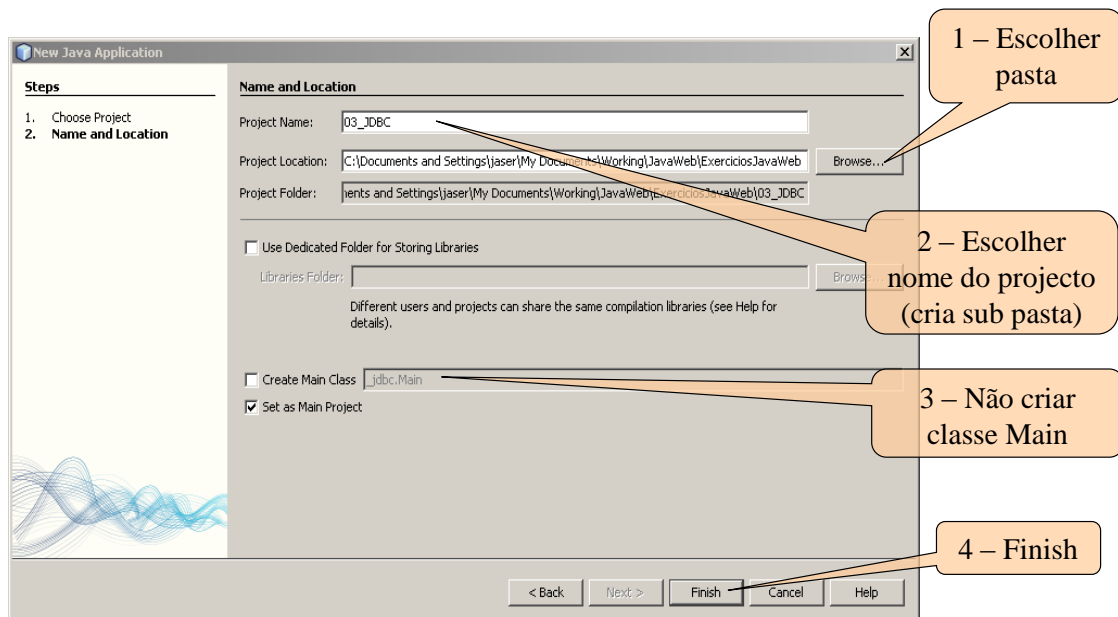
Java Web  
© Citeforma

Capítulo 3 - Acesso a base de dados com JDBC

3

## NetBeans - Criar um projecto

#2/3



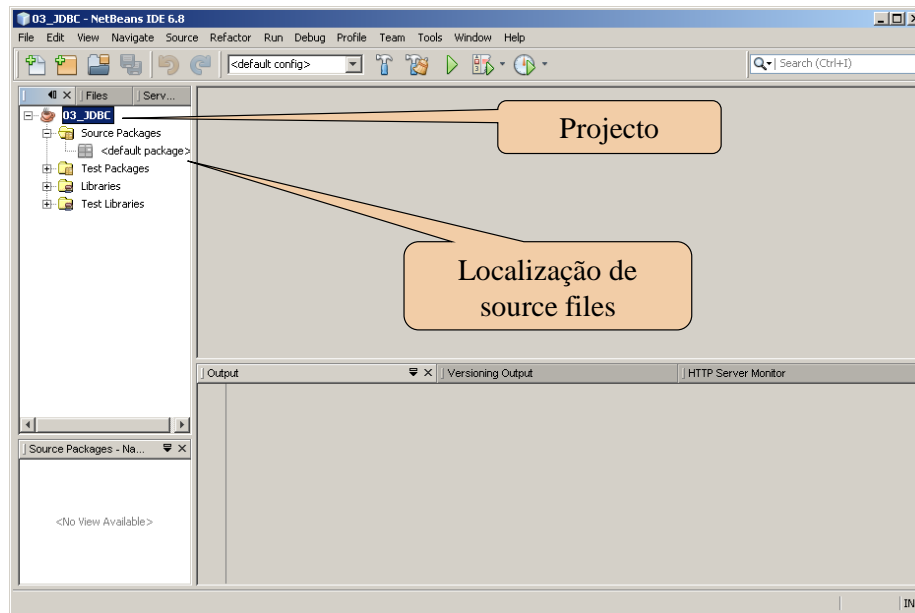
Java Web  
© Citeforma

Capítulo 3 - Acesso a base de dados com JDBC

4

## NetBeans - Criar um projecto

#3/3



## Adicionar biblioteca JDBC

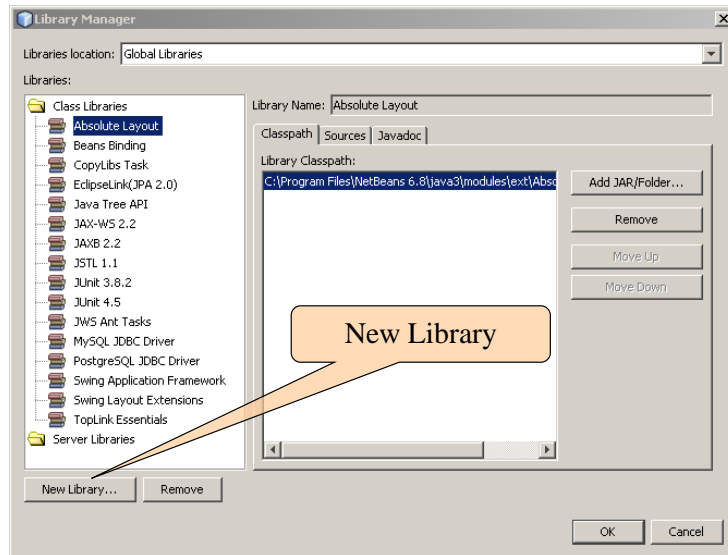
- Para que o programa Java aceda ao SGBD é necessária uma **biblioteca** JDBC;
- Armazenada em:
  - [JavaLib\Oracle](#)
  - [JavaLib\MySQL](#);
- Esta é formada por um ficheiro em formato **JAR**:
  - **Oracle**: classes12.jar, ojdbc14.jar, ojdbc5.jar, ojdbc6.jar;
  - **MySQL**: mysql-connector-java-bin.jar;
- A biblioteca tem que ser adicionada ao projeto...





## NetBeans: adicionar biblioteca JDBC #1/3

### ○ Tools → Libraries

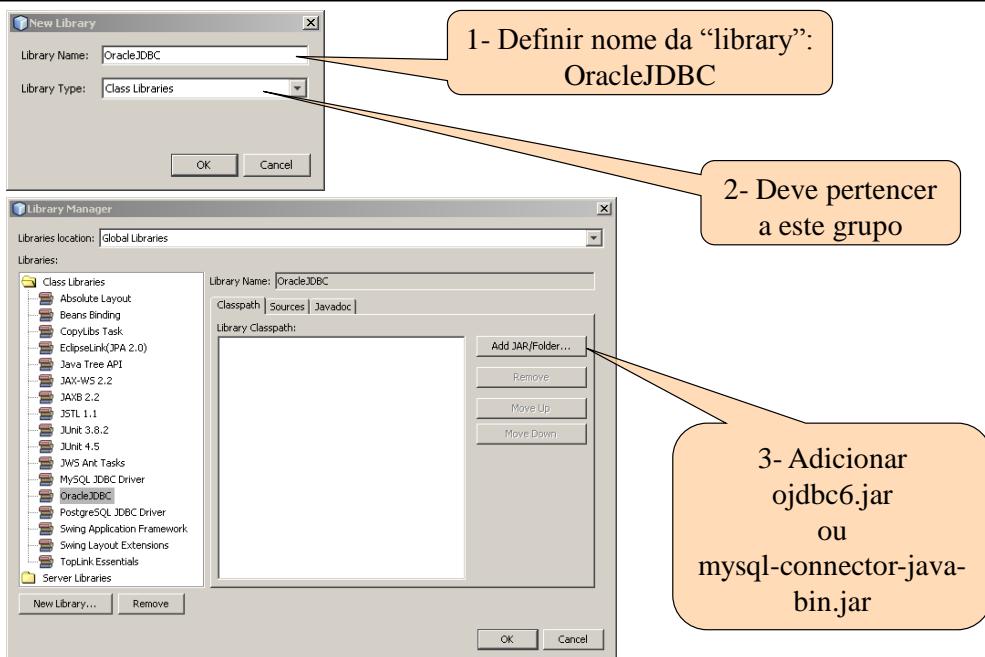


Isto é requerido  
em Oracle.

Para MySQL já  
temos Lib pré  
definida



## NetBeans: adicionar biblioteca JDBC #2/3



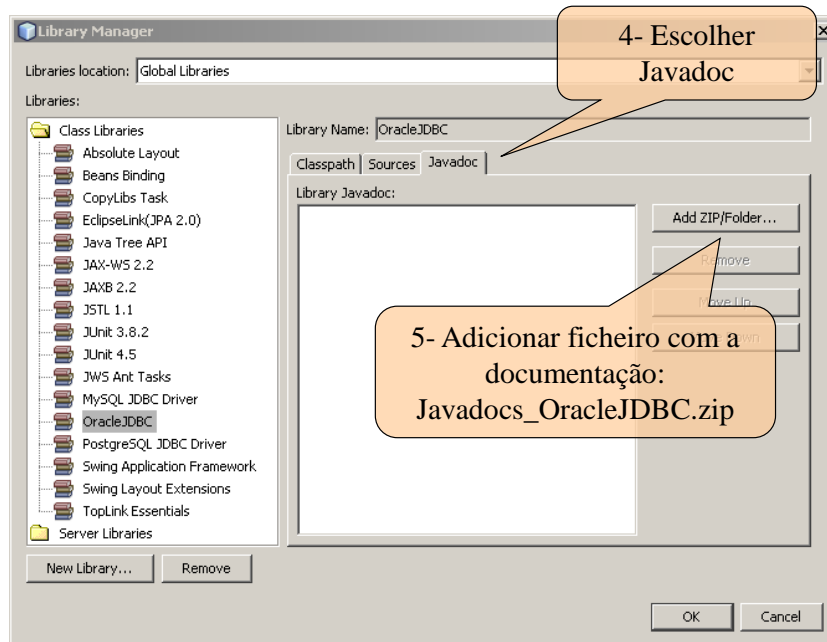
1- Definir nome da "library":  
OracleJDBC

2- Deve pertencer  
a este grupo

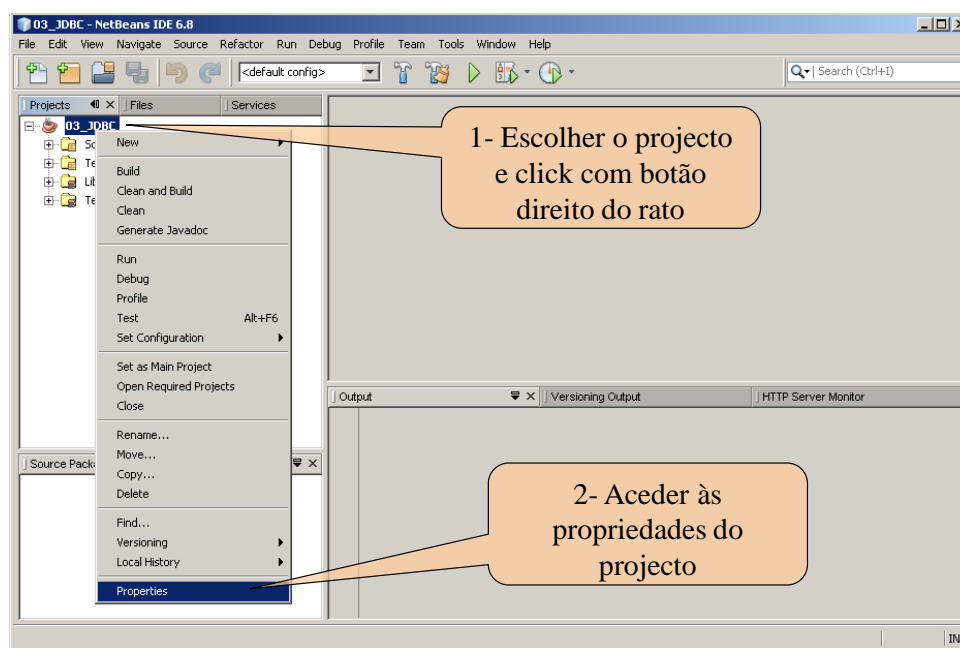
3- Adicionar  
ojdbc6.jar  
ou  
mysql-connector-java-  
bin.jar



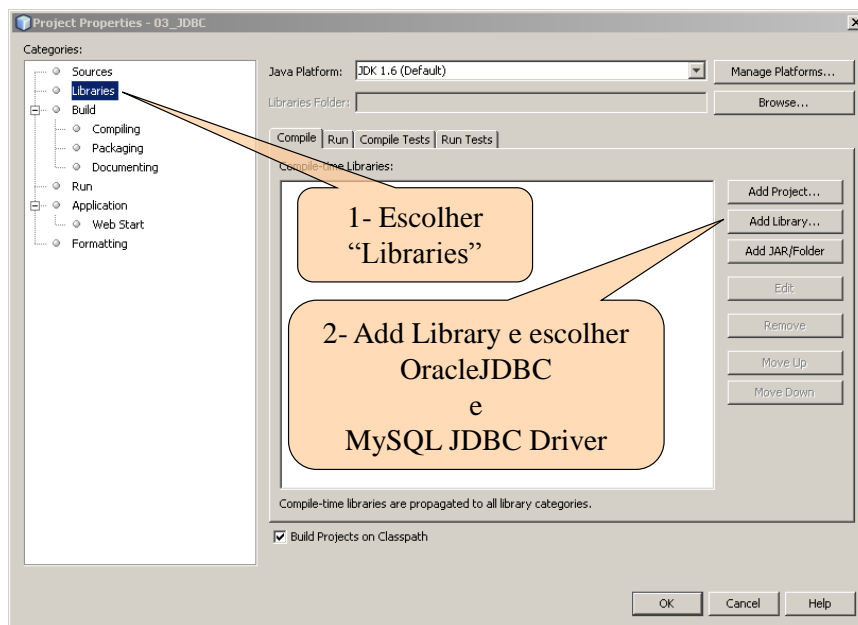
## NetBeans: adicionar biblioteca JDBC #3/3



## Adicionar biblioteca global ao projecto #1/2



## Adicionar biblioteca global ao projecto #2/2



## Instalar o XAMPP Portable

- Fazer unzip para C:\XAMPP ou D:\ XAMPP;
- Arrancar com o MySQL usando o xampp-control.exe;



## Instalar máquina virtual

- Software fornecido pelo formador:
  - Oracle VM VirtualBox (Sun VirtualBox) - runtime da máquina virtual;
  - Disco da máquina virtual – ficheiro vdi;
  - Manual de instalação da máquina virtual;



## Preparar a ligação ao SGBD

- Obter os dados de ligação ao SGBD Oracle:
  - Usar a máquina virtual;
  - Servidor: **IP da MV (localhost?)**;
  - Porto: **1521**;
  - Sid ou Service: **XE** (ou ORCL);
  - User/Password: **hr/xpto1234**;
- Obter os dados de ligação ao SGBD MySQL:
  - Usar o Wamp Portable;
  - Servidor: localhost;
  - Porto: 3306;
  - Schema ou BD: winestore;
  - User/password: winestore/xpto1234;



## Sumário

---

- Preparar o ambiente de trabalho;
- JDBC e tipos de *drivers*;
- Executar SELECT;
- Executar DDL e DML;
- *PreparedStatement*;
- Executar *stored procedure* e *function*;
- *SQL Injection*
- Segurança dos dados - Cifra

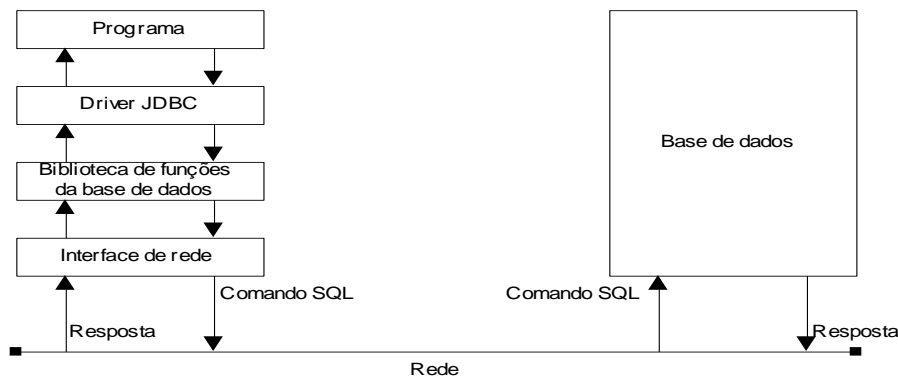


### 3.3- JDBC e tipos de drivers

Vamos ver com mais detalhe o JDBC e quais os tipos de drivers existentes.

## O que é o JDBC?

- **J**a**v**a **D**a**t**a**B**ase **C**o**n**n**e**c**t**i**v**i**t**y é uma API;
- Baseia-se na norma X/OPEN;
- É independente da BD.



### 3.3.1- O que é o JDBC?

As aplicações têm necessidade de guardar os seus dados em suporte não volátil que possibilite o menor tempo possível de escrita e leitura. Nos ambientes empresariais são utilizados **Sistemas de Gestão de Bases de Dados (SGBD)** para gerir esses dados, pois aportam inúmeras vantagens sobre os sistemas de ficheiros, das quais se destacam:-

- **Independência entre dados e aplicações.** A base de dados permite que a consulta e alteração dos dados se tornem independentes dos programas e linguagens em que as aplicações são desenvolvidas;
- Gestão centralizada de segurança no acesso aos dados;
- **Integridade dos dados** no que respeita a relações entre conteúdos;
- **Integridade dos dados** no que respeita a tolerância a falhas de hardware;
- **Regras de negócio associadas aos dados**, sendo partilhadas por todas as aplicações;

A linguagem JAVA define o **JDBC** como uma **API** de programação para permitir a interação dos programas JAVA com **SGBDs**. O acrónimo **JDBC** refere-se a **J**a**v**a **D**a**t**a**B**ase **C**o**n**n**e**c**t**i**v**i**t**y, enquanto a sigla **API** abrevia **A**p**p**l**i****c**a**t**i**o****n** **P**r**o****g**r**a****m**m**i****n****g** **I**n**t**e**r**f**a****c**e. Mais especificamente, a **API JDBC** define como uma aplicação abre uma conexão, comunica com a base de dados, executa um comando **SQL** (**S**tr**u****c**t**u****r****e****d** **Q**u**e****r****y** **L**a**n**g**u****a****g****e**) e manipula o resultado do comando. O termo aplicação refere-se a programas Java que correm isolados, applets, servlets ou Enterprise Java Beans.

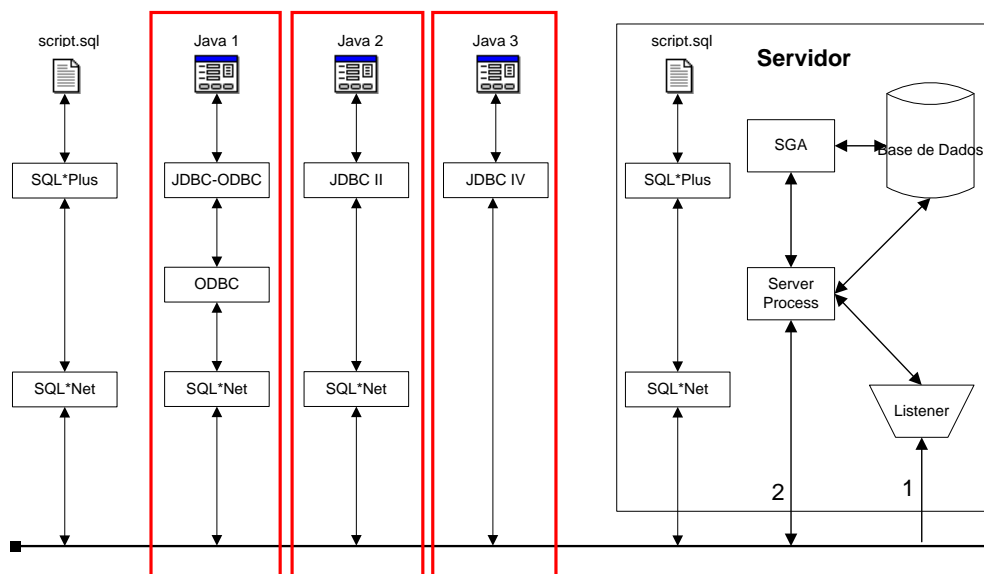
O **JDBC** baseia-se numa interface definida pela norma **X/OPEN** para chamadas a **SQL**. O **JDBC** inspirou-se nos princípios do **ODBC** (**O**p**e****n** **D**a**t**a**B**ase **C**o**n**n**e**c**t**i**v**i**t**y), criado pela Microsoft e com larga aceitação no mercado, mas não é uma “tradução” deste. O **JDBC**

pretende ser uma interface mais simples e compacta, destinada principalmente à execução de comandos **SQL** e ao tratamento dos seus resultados.

A **API JDBC** pretende que os programas JAVA consigam interagir com qualquer base de dados. Para isso é especificada a comunicação entre o programa e o **SGBD**, sendo definida a sintaxe e funcionalidade dos comandos, como devem ser executados e qual o formato dos dados resultantes. Para a sua elaboração foram consultados os principais fabricantes de bases de dados e de servidores aplicativos, entre os quais se destacam, por ordem alfabética: Borland, IBM, Informix, Intersoft, Intersolve, Microsoft, Oracle, Sybase, WebLogic.

A missão do driver **JDBC** é garantir que o programa JAVA tem acesso consistente a uma base de dados. Isto significa que os pedidos feitos pelo programa chegam à base de dados e são por ela compreendidos, assim como os resultados que saem da base de dados chegam ao programa e são por ele interpretados.

## Como um cliente liga ao SGBD Oracle?



### 3.3.2- Como um cliente liga ao SGBD Oracle?

O **SQL\*Net** é um produto desenvolvido pela Oracle para permitir ligações entre clientes e servidores independentes do protocolo de rede. Uma ligação SQL\*Net pode ser efectuada sobre TCP/IP, NetBEUI, SNA ou outro protocolo. Quando um cliente executa um script SQL dentro do SQL\*Plus é estabelecida uma ligação entre o cliente e o servidor, que usa um canal SQL\*Net sobre um protocolo de comunicações.

Para ligação a uma base de dados existem 4 tipos de drivers JDBC. A figura acima ilustra o funcionamento de 3 desses drivers, disponíveis para Oracle, comparados com uma ligação normal SQL\*Plus.



## *Driver Oracle tipo I: JDBC-ODBC* #1/2

- Passos de instalação e uso:
  - Instalar um cliente Oracle com os drivers ODBC:
    - Incluídos na instalação do Oracle XE Server;
  - **Arrancar a Base de Dados e o *listener***;
  - Criar uma “**connect string**” para ligação ao servidor Oracle:
    - Pré-definida com o nome **XE** (*tnsnames.ora*);
    - Testar conectividade usando **SQL\*Plus**;
  - Criar um “**System DSN**” que usa a “**connect string**” anterior:
    - Usar utilitário **odbcad32.exe** e dar o nome **ODBC\_XE**;
    - Testar a conectividade do “**System DSN**”;
  - Criar uma classe Java que usa o “**System DSN**” anterior.



### 3.3.2.1- Driver Tipo I: JDBC-ODBC

O **JDBC** fala com o **ODBC** e este com o **SQL\*Net**, que por sua vez vai estabelecer a ligação com o servidor, usando **TCP/IP** ou **outro protocolo de rede**. Exige a instalação do “Oracle ODBC” e do “Oracle SQL\*Net” no computador cliente.

O slide acima descreve os passos necessários para instalar e usar uma ligação deste tipo.

## Driver Oracle tipo I: JDBC-ODBC #2/2

### ○ Código Java para criar conexão:

```
try {  
    DriverManager.registerDriver(  
        new sun.jdbc.odbc.JdbcOdbcDriver());  
    String systemDSN = "jdbc:odbc:ODBC_XE";  
    String user = "hr";  
    String pwd = "xpto1234";  
    Connection conn = DriverManager.getConnection(  
        systemDSN, user, pwd);  
    ...  
    conn.close();  
} catch (SQLException e) {}
```

Regista o Driver JDBC-ODBC

SystemDSN

Abre a conexão à Base de Dados

Fecha a conexão



As instruções Java do slide acima descrevem como se estabelece uma ligação à base de dados Oracle usando um driver JDBC tipo I: JDBC-ODBC.

## Driver Oracle tipo II: JDBC-OCI

#1/3

### ○ Passos de instalação e uso:

- Instalar um cliente Oracle:
  - Incluído na instalação do Oracle XE;
- **Arrancar a Base de Dados e o *listener*;**
- Criar uma “*connect string*” para ligação ao servidor Oracle:
  - Pré-definida com o nome **XE** (**tnsnames.ora**);
  - Testar conectividade usando **SQL\*Plus**;
- Criar uma classe Java que usa a “*connect string*” anterior.



Java Web  
© Citeforma

Capítulo 3 - Acesso a base de dados com JDBC

20

### 3.3.2.2- Driver Tipo II: JDBC-OCI

Neste tipo de “driver” o **JDBC** fala directamente com o **SQL\*Net**, que por sua vez vai estabelecer a ligação com o servidor Oracle, usando **TCP/IP** ou **outro protocolo de rede**. Este “driver” exige a instalação do “Oracle JDBC OCI” e do “Oracle SQL\*Net” no cliente.

O slide acima descreve os passos necessários para instalar e usar uma ligação deste tipo.

## Driver Oracle tipo II: JDBC-OCI

#2/3

### ○ Código Java para criar conexão:

```
try {  
    String url = "jdbc:oracle:oci:@XE";  
    String user = "hr";  
    String pwd = "xpto1234";  
    OracleDataSource ds = new OracleDataSource();  
    ds.setUser(user);  
    ds.setPassword(pwd);  
    ds.setURL(url);  
    Connection conn = ds.getConnection();  
    ...  
    conn.close();  
} catch (SQLException e) {}
```

Ligação por JDBC-OCI

Cria um DataSource

Define propriedades

Abre a conexão à Base de Dados

Fecha a conexão



As instruções Java do slide acima descrevem como se estabelece uma ligação à base de dados Oracle usando um driver JDBC tipo II: JDBC-OCI.

**XE** representa a “connect string” usada pelo cliente Oracle que está definida no ficheiro **tnsnames.ora** ou noutro mecanismo de resolução de nomes Oracle. Se tiver optado por instalar o Oracle XE na sua máquina local já tem a definição anterior configurada.

## Driver Oracle tipo II: JDBC-OCI

#3/3

- Compatibilidade entre “*libraries*” JAVA e versões do servidor de BD Oracle:

Oracle / Java	1.3 classes12.jar	1.4 ojdbc14.jar	1.5 ojdbc5.jar	1.6 ojdbc6.jar	...
8.0	Sim	Não	Não	Não	
8i	Sim	Não	Não	Não	
9i	Sim	Sim	Sim	Sim	
10g	Sim	Sim	Sim	Sim	
11g	??	??	??	Sim	



Java Web  
© Citeforma

Capítulo 3 - Acesso a base de dados com JDBC

22

A tabela do slide acima mostra a compatibilidade entre versões de base de dados Oracle, versões da JRE (Java Runtime Environment) e versões do “driver” JDBC Oracle. Se usar uma base de dados Oracle 10g então deverá usar um JRE 1.3, 1.4 ou 1.5, com os drivers JDBC classes12.jar para o primeiro JRE e ojdbc14.jar para os segundos.



Nas versões **anteriores à versão 1.4 do JRE** a ligação à base de dados fazia-se utilizando as seguintes instruções:

```
// --> Para versões anteriores a Oracle 9i com classes11.zip ou classes12.jar
// Registrar driver
// DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
// Abrir conexão
// Connection conn = DriverManager.getConnection(url, user, pwd);
```

Para a **versão 1.4 ou outras mais recentes** devem ser utilizadas as instruções apresentadas no exemplo da secção de Exercícios para os drivers de Tipo II e Tipo IV. Estas recorrem à classe **oracle.jdbc.pool.OracleDataSource**, definida no driver de Oracle e que apresenta vários melhoramentos face às versões anteriores, nomeadamente em termos de eficiência na gestão das ligações.

## *Driver Oracle tipo IV: JDBC Thin* #1/3

- A tabela de compatibilidades também se aplica aqui;
- Passos de instalação e uso:
  - Escolher o “**driver**” Oracle adequado à versão de BD;
  - Copiar para uma directoria conhecida;
  - **Arrancar a Base de Dados e o listener;**
  - Configurar a variável **CLASSPATH** do Java para consultar a directoria do ponto anterior;
  - Criar uma classe Java que usa o “**driver thin**”. Necessita estes parâmetros:-
    - URL = **Servidor + Porta + BD\_SID**;
    - **Utilizador** da base de dados;
    - **Password** do Utilizador da **Base de Dados**.



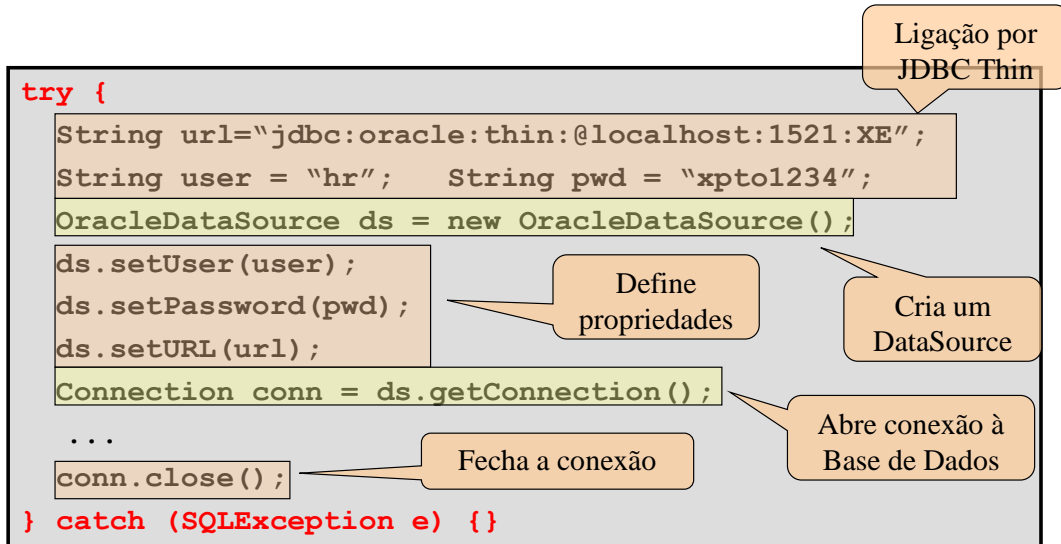
### 3.3.2.3- Driver Oracle Tipo IV: JDBC Thin

O **JDBC** estabelece ligação directa com o **Listener Oracle** e com a **base de dados**, usando **sockets** de comunicações **Java**. Exige que o cliente tenha acesso às classes Java fornecidas pela Oracle, chamadas “**Oracle JDBC Thin Driver**” (**classes11.zip**, **classes12.zip**, **classes12.jar** ou **ojdbc14.jar**).

O slide acima descreve os passos necessários para instalar e usar uma ligação deste tipo.

## Driver Oracle tipo IV: JDBC Thin #2/3

### ○ Código para criar conexão com **BD 10g e 11g**:



As instruções Java do slide acima descrevem como se estabelece uma ligação à base de dados Oracle usando um driver JDBC tipo IV (**thin**) versão ojdbc14.jar e quando o servidor tem versão 10g.

## Driver Oracle tipo IV: JDBC Thin #3/3

- Código para criar conexão com BD anterior a 10g:

```
try {  
    String url = "jdbc:oracle:thin:@localhost:1521:XE";  
    String user = "hr";    String pwd = "xpto1234";  
    DriverManager.registerDriver(  
        new oracle.jdbc.driver.OracleDriver());  
    Connection conn = DriverManager.getConnection(  
        url, user, pwd);  
    ...  
    conn.close();  
} catch (SQLException e) {}
```

Ligação por JDBC Thin

Regista o Driver

Abre a conexão à Base de Dados

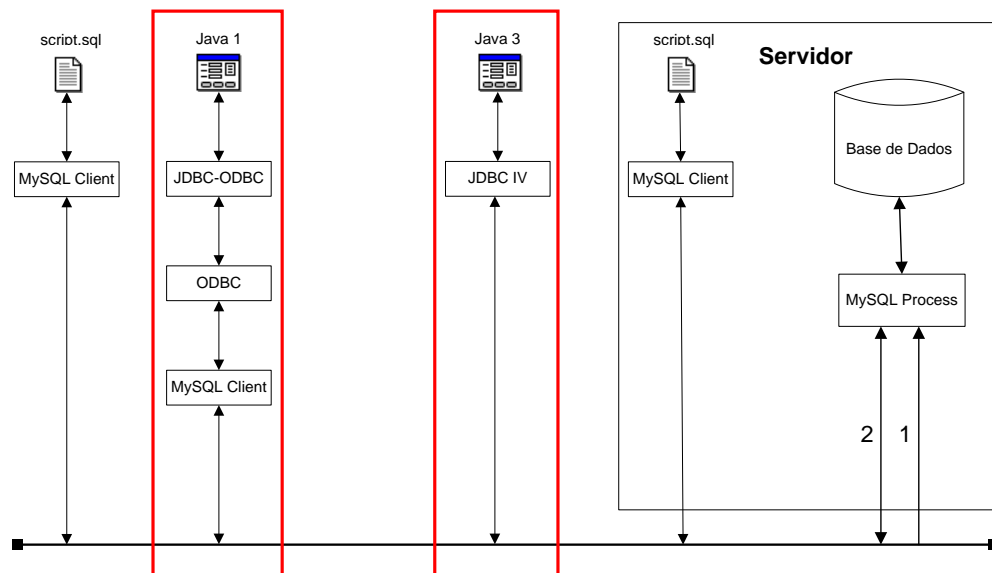
Fecha a conexão



As instruções Java do slide acima descrevem como se estabelece uma ligação à base de dados Oracle usando um driver JDBC tipo IV (**thin de qualquer versão**) e quando o servidor tem versão anterior à 10g.



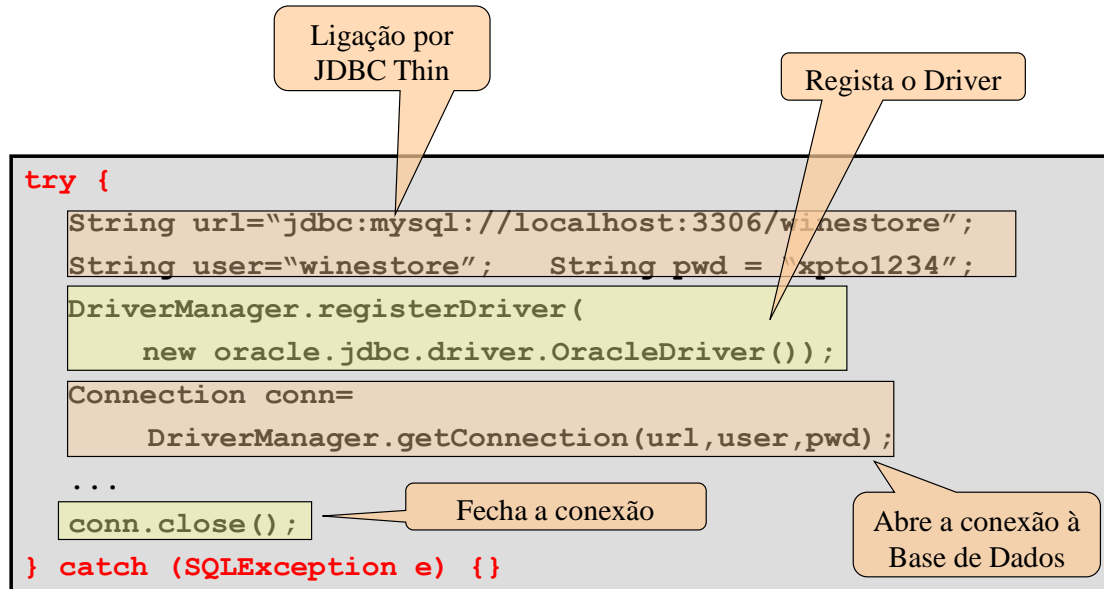
## Como um cliente liga ao SGBD MySQL?



### 3.3.3- Como um cliente liga ao SGBD MySQL?

O Java pode ligar a um servidor MySQL usando um driver do tipo I ou um driver do tipo IV.

## Driver MySQL tipo IV



O código é muito semelhante ao do SGBD Oracle, versão antiga. Em vez do SID ou Service usamos Schema (ou BD). No XAMPP fornecido pelo formador existe uma base de dados (schema) com o nome WINESTORE, o mesmo que o utilizador que será usado para acesso.

## Sumário

---

- Preparar o ambiente de trabalho;
- JDBC e tipos de *drivers*;
- Executar SELECT;
- Executar DDL e DML;
- *PreparedStatement*;
- Executar *stored procedure* e *function*;
- *SQL Injection*
- Segurança dos dados - Cifra



### 3.3.4- Executar SELECT

## SELECT

### ○ Instruções Java para extrair dados com SELECT:

```
try {  
    Statement stmt = conn.createStatement();  
    String sql =  
        "SELECT first_name,last_name FROM Employees";  
    ResultSet rset = stmt.executeQuery(sql);  
    while (rset.next()) {  
        System.out.println(rset.getString(1) +  
            " " + rset.getString(2));  
    }  
    rset.close();  
    stmt.close();  
} catch (SQLException e) { }
```

Cria stmt

Executa stmt

Trata resultados

Fecha rset e stmt



Para executar um SELECT sobre uma base de dados Oracle é necessário:

- Registrar o driver ou criar um DataSource (depende da versão do servidor);
- Abrir a ligação à base de dados;
- Escrever o SELECT statement;
- Executar o SELECT statement;
- Tratar o resultado;
- Fechar a ligação à base de dados.

As instruções Java do slide acima permitem executar um SELECT e mostrar os dados resultantes.

## Exercícios

- Criar uma classe que utiliza um *driver* Tipo I e faz SELECT à tabela *Employees*;
- Criar uma classe que utiliza um *driver* Tipo II e faz SELECT à tabela *Employees*;
- Criar uma classe que utiliza um *driver* Tipo IV e faz SELECT à tabela *Employees*;
- Criar uma classe que utiliza um *driver* Tipo IV para MySQL e faz SELECT à tabela *Customer*;




### 3.3.5- Exercícios

#### 3.3.5.1- Exercício – Driver Oracle Tipo I: JDBC-ODBC

Vamos criar um pequeno programa que nos permite criar uma ligação do **Tipo I**.

1. No servidor **arrancar** com a **base de dados Oracle** e com o respectivo **listener**;
2. No cliente **instalar e configurar o driver ODBC**. Para isso é necessário:
  - 2.1. **Instalar o cliente Oracle com o driver ODBC** (incluído na instalação local do Oracle XE);
  - 2.2. Configurar o **TNSNAMES.ORA** do cliente, definindo uma “*connect string*” para a base de dados, indicando o nome da máquina, SID e porto do listener.  
A instalação do Oracle XE inclui um cliente Oracle com uma ligação pré configurada de nome **XE**.  
Se optar por ligar a outra BD peça ajuda ao administrador da base de dados.
  - 2.3. Configurar o **ODBC**, criando um **System DSN** com a “*connect string*” definida no passo anterior.  
Se optou por usar uma base de dados existente na rede, peça ajuda ao respectivo administrador sobre os passos abaixo;  
Se optou por uma instalação local do **Oracle XE** vai verificar que este instala um driver ODBC que deve ser usado na criação do System DSN:
    - 2.3.1. Start → Control Panel → Administrative Tools → Data Sources (ODBC) ou Start → Run → **odbcad32.exe**;
    - 2.3.2. Separador System DSN → Add;

- 2.3.3. Escolher o driver **Oracle in XE**;
  - 2.3.4. Data Source Name: **ODBC\_XE**;
  - 2.3.5. TNS Service Name: **XE**;
  - 2.3.6. Test Connection: usar o utilizador **HR** com a password **xpto1234**;
3. Criar a classe Java descrita abaixo:-
- 3.1. Seleccionar o projecto **JDBC**;
    - 3.1.1. File → New → Java Class
    - 3.1.2. No wizard de criação da classe dar como nome **JDBC1**, não aceitar “Generate Default Constructor” mas aceitar “Generate Main Method”;
  - 3.2. Introduzir o código abaixo. Para que a compilação tenha sucesso este código necessita a “**library**” **Oracle** que já foi adicionada ao projecto na configuração inicial.



```
package jdbc;

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

public class JDBC1 {
    public static void main(String[] args) throws SQLException {
        // Registrar driver
        DriverManager.registerDriver(new sun.jdbc.odbc.JdbcOdbcDriver());
        String systemDSN = "jdbc:odbc:ODBC_XE";
        String user = "hr";
        String pwd = "xpto1234";
        //Abrir conexão
        Connection conn = DriverManager.getConnection(systemDSN,user,pwd);
        //Definir instrução
        String sql = "SELECT first_name, last_name FROM EMPLOYEES";
        //Executar instrução
        Statement stmt = conn.createStatement();
        ResultSet rset = stmt.executeQuery(sql);
        //Tratar Resultado
        while (rset.next()) {
            System.out.println(rset.getString(1)+" "+rset.getString(2));
        }
        //Fechar
        rset.close();
        stmt.close();
        conn.close();
    }
}
```




Este programa usa a tabela EMPLOYEES do utilizador HR e usa a “connect string” XE que vem pré definida na instalação da base de dados Oracle XE.

### 3.3.5.2- Exercício – Driver Oracle Tipo II: JDBC-OCI

Vamos criar um pequeno programa que nos permite criar uma ligação do **Tipo II**.

1. No servidor **arrancar** com a **base de dados Oracle** e com o respectivo **listener**;
2. Configurar o “**Oracle SQL\*Net**”:-
  - 2.1. A instalação do Oracle XE inclui um cliente Oracle com SQL\*Net configurado para acesso ao próprio servidor. Isto inclui uma “connect string” identificada por **XE**, definida no ficheiro **TNSNAMES.ORA**;
  - 2.2. Testar a ligação à base de dados usando **SQL\*Plus** ou outro cliente Oracle com os seguintes parâmetros:  
Username: **HR**;  
Password: **xpto1234**;  
Connect string: **XE**;
3. Criar a classe Java descrita abaixo:-
  - 3.1. Seleccionar o projecto **01\_JDBC**;
    - 3.1.1. File → New → Java Class
    - 3.1.2. No wizard de criação da classe dar como nome **JDBC2**, não aceitar “Generate Default Constructor” mas aceitar “Generate Main Method”;
  - 3.2. Introduzir o código apresentado abaixo. Para que a compilação tenha sucesso este código necessita a “**library**” **Oracle** que já foi adicionada ao projecto na configuração inicial;



```
package jdbc;

import java.sql.Connection;
//import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import oracle.jdbc.pool.OracleDataSource;

public class JDBC2 {
    public static void main(String[] args) throws SQLException {
        String url = "jdbc:oracle:oci:@XE";
        String user = "hr";
        String pwd = "xpto1234";

        //Para versões anteriores a 9i com classes12.zip ou classes12.jar
        //DriverManager.registerDriver(new
oracle.jdbc.driver.OracleDriver());
        //Connection conn = DriverManager.getConnection(url, user, pwd);
        //Ir para definir instrução

        //Para versões posteriores a 9i
        //Criar um DataSource
        OracleDataSource ds = new OracleDataSource();
        //Definir parametros no datasource
        ds.setUser(user);
        ds.setPassword(pwd);
        ds.setURL(url);
        //Abrir conexão
        Connection conn = ds.getConnection();
```

```
//Definir instrução
String sql = "SELECT first_name,last_name FROM EMPLOYEES";
//Executar instrução
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery(sql);
//Tratar resultado
while (rset.next()) {
    System.out.println(rset.getString(1)+" "+rset.getString(2));
}
//Fechar
rset.close();
stmt.close();
conn.close();
}
```



Este programa usa a tabela EMPLOYEES do utilizador HR e a connect string XE, que foi configurada durante a instalação do Oracle XE.

O driver JDBC que vem com o JDeveloper não funciona com este tipo de ligação, pelo que é necessário instalar o driver que vem com o Oracle XE ou com o cliente Oracle normal, como descrito nas instruções no início da capítulo.

### 3.3.5.3- Exercício – Driver Oracle Tipo IV: JDBC Thin

Vamos escrever um pequeno programa que nos permite criar uma ligação do **Tipo IV - thin**.

1. No servidor **arrancar** com a **base de dados Oracle** e com o respectivo **listener**;
2. Criar a classe Java descrita abaixo:-
  - 2.1. Seleccionar o projecto **JDBC**;
  - 2.2. File → New → Java Class
  - 2.3. No wizard de criação da classe dar como nome **JDBC4New**, não aceitar “Generate Default Constructor” mas aceitar “Generate Main Method”;
  - 2.4. Introduzir o código apresentado abaixo. Para que a compilação tenha sucesso este código necessita a “**library**” **Oracle** que já foi adicionada ao projecto na configuração inicial.
  - 2.5. A ligação à base de dados usará os seguintes parâmetros:  
Host: **localhost** (ou o nome/IP da máquina onde corre o servidor)  
Port: **1521** (ou o porto onde o listener está à escuta no servidor)  
SID: **XE** (ou o valor fornecido pelo administrador da base de dados)  
Username: **HR**;  
Password: **xpto1234**;



```
package jdbc;

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



```
import oracle.jdbc.pool.OracleDataSource;

public class JDBC4New {
    public static void main(String[] args) throws SQLException,
    ClassNotFoundException {
        String host="localhost", port="1521", sid="XE";
        String driver = "jdbc:oracle:thin";
        String url = driver + ":@" + host + ":" + port + ":" + sid;
        String user = "hr", pwd = "xpto1234";
        //Criar um DataSource
        OracleDataSource ds = new OracleDataSource();
        //Definir parametros no datasource
        ds.setUser(user);
        ds.setPassword(pwd);
        ds.setURL(url);
        //Abrir conexao
        Connection conn = ds.getConnection();
        //Definir instrucao
        String sql = "SELECT first_name,last_name FROM EMPLOYEES";
        //Executar instrucao
        Statement stmt = conn.createStatement();
        ResultSet rset = stmt.executeQuery(sql);
        //Tratar resultado
        while (rset.next()) {
            System.out.println(rset.getString(1) + " " +
rset.getString(2));
        }
        //Fechar
        rset.close();
        stmt.close();
        conn.close();
    }
}
```



Este programa usa a tabela EMPLOYEES do utilizador HR.



O driver JDBC incluído no JDeveloper funciona com este tipo de ligação ao servidor. Este programa é igual ao exemplo anterior excepto na forma como se cria o URL.



Os restantes exercícios deste curso irão utilizar o driver tipo IV (thin) para ligar à base de dados, pois não requer a configuração do cliente Oracle nem do JDBC.

O próximo exemplo é idêntico ao anterior mas usa o mecanismo de ligação antigo:

```
package jdbc;

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

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

public class JDBC4Old {
    public static void main(String[] args) throws SQLException,
    ClassNotFoundException {
        String host="localhost", port="1521", sid="XE";
        String driver = "jdbc:oracle:thin";
        String url = driver + ":@" + host + ":" + port + "/" + sid;
        String user = "hr", pwd = "xpto1234";
        //Registrar o driver
        DriverManager.registerDriver(new
oracle.jdbc.driver.OracleDriver());
        //Class.forName("oracle.jdbc.driver.OracleDriver");
        //Abrir conexao
        Connection conn = DriverManager.getConnection(url, user, pwd);
        //Definir instrucao
        String sql = "SELECT first_name,last_name FROM EMPLOYEES";
        //Executar instrucao
        Statement stmt = conn.createStatement();
        ResultSet rset = stmt.executeQuery(sql);
        //Tratar resultado
        while (rset.next()) {
            System.out.println(rset.getString(1) + " " +
rset.getString(2));
        }
        //Fechar
        rset.close();
        stmt.close();
        conn.close();
    }
}
```

#### 3.3.5.4- Exercício – Driver MySQL Tipo IV: JDBC Thin

O próximo exemplo mostra uma ligação ao MySQL com um driver do tipo IV. Use o MySQL disponibilizado pelo formador:

```
package jdbc;

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

public class JDBC4MySQL {

    public static void main(String[] args) throws SQLException,
    ClassNotFoundException {
        String host="localhost", port="3306", dbSchema="winestore";
        String driver="jdbc:mysql";
        String url= driver + "://" + host + ":" + port + "/" + dbSchema;
        String user="winestore", pwd="xpto1234";
        //Registrar o driver
        DriverManager.registerDriver(new com.mysql.jdbc.Driver());
        //Class.forName("com.mysql.jdbc.Driver");
        //url = jdbc:mysql://[host][:port]/[database]
        //"jdbc:mysql://localhost:3306/winestore", "winestore", "xpto1234"
        //Abrir conexao
        Connection conn = DriverManager.getConnection(url, user, pwd);
    }
}
```

```
//Definir instrucao
String sql = "SELECT * from customer";
//Executar instrucao
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery(sql);
//Tratar resultado
while (rset.next()) {
    System.out.println(rset.getString(1) + " " +
rset.getString(2)+ " " + rset.getString(3));
}
//Fechar
rset.close();
stmt.close();
conn.close();
}
}
```

## Sumário

---

- Preparar o ambiente de trabalho;
- JDBC e tipos de *drivers*;
- Executar SELECT;
- Executar DDL e DML;
- *PreparedStatement*;
- Executar *stored procedure* e *function*;
- SQL *Injection*
- Segurança dos dados - Cifra



### 3.4- Executar DDL e DML

Vamos ver como se executam comando DDL e DML.

## Comandos DDL

- São executados pelo método **executeUpdate()** das classes *Statement* ou *PreparedStatement*;
- Podem lançar uma **SQLException**;
- Em Oracle estes comandos terminam a transacção pendente - *commit* implícito;

```
stmt = conn.createStatement();  
String sql = "create table CONTA_BANCARIA (" +  
    " numero number(14) primary key, " +  
    " tipo char(1) not null, " +  
    " saldo number(16,2) not null, " +  
    " ultimo_movimento date )";  
stmt.executeUpdate(sql);
```

Criar stmt

Executar



### 3.4.1- Comandos DDL – Data Definition Language

Os comandos de **DDL** (**Data Definition Language**) são executados pelo método **executeUpdate()** da classe *Statement* ou *PreparedStatement*. Caso ocorra um erro é lançada uma **SQLException**.

Em Oracle os comandos **DDL** finalizam a transacção que está em curso, fazendo um commit implícito.

## Comandos DML

#1/2

- DML = **Select, Insert, Update e Delete**;
- As transacções terminam com **COMMIT**, **ROLLBACK**, um comando **DDL** ou **DCL**;
- No **JDBC** o **AUTOCOMMIT** está activado por omissão;

```
conn.setAutoCommit(false);  
stmt = conn.createStatement();  
String sql = "insert into CONTA_BANCARIA values " +  
    (1, 'O', 2000.00, to_date('2002-04-02', 'yyyy-mm-  
    dd'))";  
int n = stmt.executeUpdate(sql);  
conn.commit();
```

Desactivar autocommit

Criar stmt

Executar

Fazer commit



Java Web  
© Citeforma

Capítulo 3 - Acesso a base de dados com JDBC

33

### 3.4.2- Comandos DML – Data Manipulation Language

#### 3.4.2.1- Noção de transacção

Uma **transacção** é um conjunto de **instruções SQL** que devem funcionar como um todo. Isto significa que uma **transacção** tem sucesso se todas as suas instruções tiverem sucesso e fracassa se uma das suas instruções falhar. Uma **transacção** pode também ser constituída apenas **por um comando SQL**.



**Os comandos dados a uma base de dados Oracle estão sempre incluídos numa transacção.**

Esta afirmação é fácil de compreender quando se trata de comandos como **INSERT**, **UPDATE** e **DELETE**, mas vários comandos **SELECT** podem formar uma transacção desde que se active o nível de isolamento "**SERIALIZABLE**" ou "**REPEATABLE READ**". Na realidade mesmo um único **SELECT** pode ser tratado pela base de dados como se fosse uma transacção.

A execução de um único **SELECT** pode gerar problemas de integridade de leitura e portanto exige do lado da base de dados um tratamento de transacção. Por exemplo vamos considerar que o **SELECT** efectua cálculos estatísticos sobre tabelas de grande dimensão, demorando várias horas a correr. Durante a sua execução podem ocorrer **INSERTS**, **UPDATES** e **DELETES** que alteram as tabelas lidas pelo **SELECT**. Isto põe em perigo a **coerência do resultado** devolvido pelo **SELECT**, pelo que cabe à base de dados garantir a coerência na leitura dos dados. Usando os "**Rollback Segments**" a base de dados vai gerar

a informação auxiliar necessária para guardar as imagens dos dados antes da sua alteração, permitindo ao **SELECT** trabalhar com uma imagem das tabelas congelada no tempo. Quando o **SELECT** terminar a informação auxiliar é removida. Se o volume de alterações for elevado a dimensão dos **Rollback Segments** pode não ser suficiente para manter a fotografia inicial, pelo que o Oracle vai abortar o **SELECT** com uma mensagem de erro: espaço insuficiente no rollback ou “snapshot too old”. O **SELECT** funcionou como uma **transação**.

#### 3.4.2.2- Controlo de transações

As **transações** em Oracle são controladas pelos comandos **ROLLBACK**, **COMMIT** ou por um comando **DDL** (**Data Definition Language**). A **transação** começa após um destes comandos e termina com outro.

Uma conexão **JDBC** ao Oracle é aberta em modo **AUTOCOMMIT**. Isto significa que cada comando **DML** é considerado uma **transação**, sendo feito um **COMMIT** implícito após a sua execução. Para alterar esta definição usamos o método **setAutoCommit()** da classe **Connection**. Depois de desactivar o modo **AUTOCOMMIT** somos obrigados a fazer **COMMIT** cada vez que se pretende que termine a transação e as alterações aos dados fiquem confirmadas.

## Comandos DML

#2/2

```
stmt = conn.createStatement();  
sql = "update CONTA_BANCARIA set saldo = saldo*1.20";  
int n = stmt.executeUpdate(sql);  
System.out.println(n + " linhas alteradas!");  
stmt.close(); conn.close();
```

Executar DML em  
modo autocommit

```
conn.setAutoCommit(false);  
stmt = conn.createStatement();  
String sql =  
    "delete from CONTA_BANCARIA where numero > -1";  
int n = stmt.executeUpdate(sql);  
conn.rollback();  
stmt.close(); conn.close();
```

Executar DML e  
fazer rollback



O slide anterior mostra as instruções Java que executam um UPDATE em modo AUTOCOMMIT e um DELETE com ROLLBACK explícito.

O comando `executeUpdate()` devolve um **int** que representa o número de linhas afetadas pelo comando. Como nos exemplos anteriores se ocorrer um erro é lançada uma **SQLException**.



## Exercícios

- Criar classe que cria a tabela ContaBancaria;
- Criar classe que:
  - Insere uma linha na tabela e faz “rollback”;
  - Insere nova linha e faz “commit”;
- Criar classe que aumenta em 20% o saldo de todas as contas existentes;
- Criar classe que:
  - Apaga todas as linhas com número superior a -1;
  - Mostra as linhas que ficaram na tabela.



### 3.4.3- Exercícios

#### 3.4.3.1- Exercício – Criar Tabela

Vamos criar um pequeno programa que nos permite criar a tabela ContaBancaria:



```
package jdbc;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import oracle.jdbc.pool.OracleDataSource;

public class Table10Create {
    public static void main(String[] args) {
        String url = "jdbc:oracle:thin:@//localhost:1521/XE";
        String user = "HR", pwd = "xpto1234";
        OracleDataSource ds = null;
        Connection conn = null;
        Statement stmt = null;
        try {
            ds = new OracleDataSource();
```

```
ds.setUser(user);
ds.setPassword(pwd);
ds.setURL(url);
conn = ds.getConnection();
stmt = conn.createStatement();
String sql =
    "create table CONTA_BANCARIA (          " +
    "    numero number(14) constraint ContaBancaria_PK primary
key, " +
    "    tipo char(1) not null,            " +
    "    saldo number(16,2) not null,      " +
    "    ultimo_movimento date             " + ")";
stmt.executeUpdate(sql);
System.out.println("Tabela criada!");
} catch (SQLException e1) {
    e1.printStackTrace(System.out);
    //System.out.println(e1.getMessage());
    //System.out.println(e1.getErrorCode());
} finally {
    try {
        if (stmt != null) {
            stmt.close();
            System.out.println("Statement fechado!");
        }
        if (conn != null) {
            conn.close();
            System.out.println("Conexao fechada!");
        }
    } catch (SQLException e2) {
        e2.printStackTrace(System.out);
    }
}
}
```



No fim da execução do programa a conexão à base de dados deve ser fechada. A execução do programa pode terminar normalmente ou pelo tratamento de uma excepção. A clausula “finally” é usada para garantir que o método close() é executado em todos os cenários.

close() é um método de objecto da classe Connection, pelo que só pode ser executado se “conn” apontar para um objecto instanciado. Retirando a instrução if (conn != null) teremos o erro “Null pointer exception” cada vez que a conexão não tenha sido estabelecida. Isto ocorre, por exemplo, quando os parâmetros da conexão foram mal definidos ou o listener da base de dados não está “à escuta”.



Quando é criado um “statement” do lado do cliente a base de dados cria um cursor para o suportar. Quando termina a execução do “statement” devemos fechá-lo, usando o método close(), o que garante que a base de dados fecha o cursor que lhe está associado e liberta os seus recursos.

Quando executamos vários “statements” dentro da mesma ligação, se não os fecharmos, não só não libertamos recursos na base de dados como podemos atingir o número máximo de cursores abertos por sessão, o que provoca um erro Oracle e o fim da nossa transação.



O método `close()` pode gerar uma `Exception`, pelo que tem que ficar dentro de um bloco `try-catch`.



Neste exemplo não nos preocupámos com `commit` ou `rollback`, já que a transação é composta por um único comando `"CREATE TABLE"` que vai provocar `COMMIT`. No exemplo do ponto seguinte vamos ter essa preocupação.



O método `executeUpdate()` devolve um `int`, que será zero caso o comando SQL seja bem sucedido. Se ocorrer um erro na sua execução será lançada uma excepção. Por isto não é relevante tratar o valor devolvido por `executeUpdate()`;

#### 3.4.3.2- Exercício – INSERT com Commit e Rollback

O exemplo abaixo executa duas instruções de `INSERT`. No fim da primeira é feito `ROLLBACK`, enquanto que na segunda é feito `COMMIT`. Os dados do primeiro `INSERT` não ficam armazenados na tabela que foi criada no exercício anterior.



```
package jdbc;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import oracle.jdbc.pool.OracleDataSource;

public class Table20Insert {
    public static void main(String[] args) {
        String url = "jdbc:oracle:thin:@//localhost:1521/XE";
        String user = "HR", pwd = "xpto1234";
        OracleDataSource ds = null;
        Connection conn = null;
        Statement stmt = null;
        try {
            ds = new OracleDataSource();
            ds.setUser(user);
            ds.setPassword(pwd);
            ds.setURL(url);
            conn = ds.getConnection();
            conn.setAutoCommit(false);
            stmt = conn.createStatement();
            String sql = "insert into CONTA_BANCARIA " +
                "values (1, 'O', 2000.00, to_date('2002-04-02', 'yyyy-mm-dd'))";
            int n = stmt.executeUpdate(sql);
            conn.rollback();
            System.out.println(n + " linha inserida, mas a transacao " +
                "foi ROLLBACKed");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

```
        sql = "insert into CONTA_BANCARIA " +  
              "values (2,'O',1000.00,to date('2002-04-03','yyyy-mm-  
dd'))";  
  
        n = stmt.executeUpdate(sql);  
        conn.commit();  
        System.out.println(n + " linha inserida!");  
    } catch (SQLException e1) {  
        e1.printStackTrace(System.out);  
    } finally {  
        try {  
            if (stmt != null) {  
                stmt.close();  
                System.out.println("Statement fechado!");  
            }  
            if (conn != null) {  
                conn.close();  
                System.out.println("Conexao fechada!");  
            }  
        } catch (SQLException e2) {  
            e2.printStackTrace(System.out);  
        }  
    }  
}
```



1 linha inserida, mas a transacao foi ROLLBACKed  
1 linha inserida!  
Statement fechado!  
Conexao fechada!



No modo AUTOCOMMIT cada instrução é uma transação COMMITed após a sua execução. Ao desactivar este modo (setAutoCommit(false)) a execução de cada instrução não é COMMITed de imediato, o que permite que várias instruções sejam agrupadas num conjunto. Todo o conjunto será depois COMMITed ou ROLLBACKed.



Quanto maior for o volume de dados manipulado por uma transação maiores serão os recursos consumidos pela base de dados para garantir:

- Que o respectivo rollback pode ser feito;
- Que as outras transações concorrentes lêem dados coerentes enquanto esta não termina;

Por isto o valor por omissão nas conexões é AUTOCOMMIT activado, já que garante um menor consumo de recursos para a base de dados.



Quando a conexão é fechada é feito um COMMIT implícito, mesmo com `setAutoCommit(false)`. Isso pode ser verificado executando o exemplo acima após comentar a linha `conn.commit()`.



Se o comando INSERT falhar, por exemplo porque a chave primária está duplicada, é gerada uma `SQLException` e não chega a ser mostrada a mensagem de output “1 linha inserida!”;



O comando `executeUpdate()` da classe `Statement` devolve um `int` que contém o número de linhas afectadas pelo comando. Nos comandos INSERT deste exemplo o valor devolvido é sempre 1.

#### 3.4.3.3- Exercício – UPDATE de várias linhas

O exemplo abaixo aumenta o saldo das contas bancárias, actualizando várias linhas numa única operação sobre a tabela usada no exercício anterior:



```
package jdbc;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import oracle.jdbc.pool.OracleDataSource;

public class Table30Update {
    public static void main(String[] args) {
        String url = "jdbc:oracle:thin:@//localhost:1521/XE";
        String user = "HR", pwd = "xpto1234";
        OracleDataSource ds = null;
        Connection conn = null;
        Statement stmt = null;
        try {
            ds = new OracleDataSource();
            ds.setUser(user);
            ds.setPassword(pwd);
            ds.setURL(url);
            conn = ds.getConnection();
            conn.setAutoCommit(false);
            stmt = conn.createStatement();
            String sql = "insert into CONTA_BANCARIA " +
                "values (4, 'O', 1000.00, to_date('2002-04-02', 'yyyy-mm-";
            dd'))";
            int n = stmt.executeUpdate(sql);
            System.out.println(n + " linha inserida!");
            sql = "update CONTA_BANCARIA " + "set saldo = saldo*1.20";
            n = stmt.executeUpdate(sql);
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if (stmt != null) stmt.close();
            if (conn != null) conn.close();
        }
    }
}
```

```
        System.out.println(n + " linhas alteradas!");
        conn.commit();
    } catch (SQLException e1) {
        e1.printStackTrace(System.out);
    } finally {
        try {
            if (stmt != null) {
                stmt.close();
                System.out.println("Statement fechado!");
            }
            if (conn != null) {
                conn.close();
                System.out.println("Conexao fechada!");
            }
        } catch (SQLException e2) {
            e2.printStackTrace(System.out);
        }
    }
}
```



1 linha inserida!  
2 linhas alteradas!  
Statement fechado!  
Conexao fechada!



A transação deste exemplo é composta por um INSERT e um UPDATE;



O comando `executeUpdate()` devolve o número de linhas alteradas pela instrução, o que no primeiro caso é 1 e no segundo são todas as linhas da tabela;

#### 3.4.3.4- Exercício – DELETE com Rollback

O próximo exemplo mostra uma transação composta por um DELETE e um SELECT, seguida de outra transação composta por um SELECT. A primeira termina com ROLLBACK.



```
package jdbc;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import oracle.jdbc.pool.OracleDataSource;

public class Table40Delete {
    public static void main(String[] args) {
        String url = "jdbc:oracle:thin:@//localhost:1521/XE";
```

```
String user = "HR", pwd = "xpto1234";
OracleDataSource ds = null;
Connection conn = null;
Statement stmt = null;
ResultSet rset = null;
try {
    ds = new OracleDataSource();
    ds.setUser(user);
    ds.setPassword(pwd);
    ds.setURL(url);
    conn = ds.getConnection();
    conn.setAutoCommit(false);
    stmt = conn.createStatement();
    String sql = "delete from CONTA_BANCARIA where numero > -1";
    int n = stmt.executeUpdate(sql);
    System.out.println(n + " linhas apagadas!");
    stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                ResultSet.CONCUR_READ_ONLY);

    sql = "select numero, tipo, saldo, ultimo_movimento " +
          "from CONTA_BANCARIA";
    rset = stmt.executeQuery(sql);
    mostraResultados(rset);
    conn.rollback();
    rset = stmt.executeQuery(sql);
    mostraResultados(rset);
} catch (SQLException e1) {
    e1.printStackTrace(System.out);
} finally {
    try {
        if (rset != null) {
            rset.close();
            System.out.println("ResultSet fechado!");
        }
        if (stmt != null) {
            stmt.close();
            System.out.println("Statement fechado!");
        }
        if (conn != null) {
            conn.close();
            System.out.println("Conexao fechada!");
        }
    } catch (SQLException e2) {
        e2.printStackTrace(System.out);
    }
}

public static void mostraResultados(ResultSet rset) {
    try {
        System.out.println("=====");
        System.out.println("Resultado do Select:");
        while (rset.next()) {
            System.out.println(rset.getLong("numero") + " " +
                               rset.getString("tipo") + " " +
                               rset.getFloat("saldo") + " " +
                               rset.getDate("ultimo_movimento"));
        }
        System.out.println("=====");
    } catch (SQLException e) {
        e.printStackTrace(System.out);
    }
}
```

```
}
```



```
2 linhas apagadas!
=====
Resultado do Select:
=====
Resultado do Select:
2  O  1200.0  2002-04-03
3  O  1200.0  2002-04-02
=====
ResultSet fechado!
Statement fechado!
Conexao fechada!
```



Neste exemplo executamos duas instruções SELECT pelo que, além de fechar Connection e Statement, fechamos também o ResultSet, otimizando assim a gestão de recursos de memória na base de dados.



O primeiro SELECT é feito após o DELETE de todas as linhas, pelo que não mostra dados na tabela. O segundo SELECT é feito depois do ROLLBACK que desfaz o DELETE.



As colunas devolvidas pelo ResultSet são referenciadas pelo nome usado no SELECT em vez da posição.

#### 3.4.3.5- Exercício – Metadata

O próximo exercício mostra como podemos obter o nome das colunas da tabela:



```
package jdbc;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;
import oracle.jdbc.pool.OracleDataSource;

public class Metadata01 {

    public static void main(String[] a) {
        String url = "jdbc:oracle:thin:@localhost:1521:XE";
        String user = "HR", pwd = "xptol234";
        OracleDataSource ds;
        Connection conn;
        ResultSetMetaData rsetMetaData;
```



```
Statement st;
ResultSet rset;
try {
    ds = new OracleDataSource();
    ds.setUser(user);
    ds.setPassword(pwd);
    ds.setURL(url);
    conn = ds.getConnection();
    conn.setAutoCommit(false);
    st = conn.createStatement();
    String sql = "Select * from employees";
    rset = st.executeQuery(sql);

    //recolher metadata apos execucao do query
    rsetMetaData = rset.getMetaData();
    //obter o numero de colunas
    int columnCount = rsetMetaData.getColumnCount();
    //obter o nome das colunas
    for (int i = 1; i <= columnCount; i++) {
        System.out.printf("%-25s |", rsetMetaData.getColumnName(i));
    }
    System.out.printf("%n");

    //listar as linhas resultantes do query
    while (rset.next()) {
        for (int i = 1; i <= columnCount; i++) {
            System.out.printf("%-25s |", rset.getString(i));
        }
        System.out.printf("%n");
    }
    rset.close();
    st.close();
    conn.close();
} catch (SQLException e) {
    e.printStackTrace(System.out);
}
}
```

## Sumário

---

- Preparar o ambiente de trabalho;
- JDBC e tipos de *drivers*;
- Executar SELECT;
- Executar DDL e DML;
- *PreparedStatement*;
- Executar *stored procedure* e *function*;
- SQL Injection
- Segurança dos dados - Cifra



### 3.5- PreparedStatement

Vamos ver para que servem os PreparedStatement, como se utilizam e quais as suas vantagens e inconvenientes.

## Porquê usar *PreparedStatement*?

- Instruções SQL que, embora muito parecidas, são todas "*parsed*" pela BD, não havendo reaproveitamento:
  - Select \* from employee where employee\_id=1
  - Select \* from employee where employee\_id=2
  - Select \* from employee where employee\_id=3
- Utilizando o "*PreparedStatement*":
  - Select \* from employee where employee\_id=?
  - Durante a execução ?=1, ?=2, ?=3



### 3.5.1- Porquê usar *PreparedStatement*?

Cada vez que submetemos uma instrução SQL para execução a respetiva String é enviada para a base de dados para a operação de "parsing". Esta operação consiste na validação sintática e semântica, seguida da obtenção do plano de execução por parte da base de dados. O plano de execução é o conjunto de passos que a base de dados vai seguir para executar a instrução. Normalmente as instruções SQL possuem algoritmos alternativos para a sua execução. Cabe às base de dados encontrar as diferentes alternativas e escolher aquela que minimiza o consumo de recursos em termos de CPU, memória e I/O.

Em bases de dados que executam muitas instruções SQL, cada uma consumindo uma pequena quantidade de recursos, o tempo que o servidor gasta em "parsing" não pode ser desprezado. Uma forma de reduzir esse tempo é reaproveitar instruções SQL que já tenham sido traduzidas pelo servidor. Para isto os textos que compõem os comandos SQL são guardadas em memória na base de dados. Cada vez que a base de dados recebe um comando verifica se um texto exatamente igual está em memória. Se estiver a base de dados aproveita o trabalho anteriormente feito, poupando recursos valiosos. Caso contrário avalia o texto do novo comando.

Esta técnica tem um inconveniente: se o texto não for exatamente igual, mesmo que semanticamente equivalente, não há reaproveitamento. As instruções abaixo indicadas são semanticamente idênticas, mas não são consideradas iguais pela base de dados:



```
Select * from employee where employee_id=100;  
SELECT * from employee where employee_id=100;  
Select * from employee where employee_ID=100;  
Select * from employee where employee_id=200;
```

Tendo em conta o acima descrito, sempre que possível, deve-se utilizar comandos com o mesmo texto e assim retirar o melhor desempenho possível do SGBD.

## Vantagens e inconvenientes

### ○ Vantagens:

- A BD faz "*parsing*" da instrução **uma única** vez (poupa CPU);
- A versão compilada é **reutilizada**;
- Evita "**SQL Injection**";

### ○ Inconvenientes:

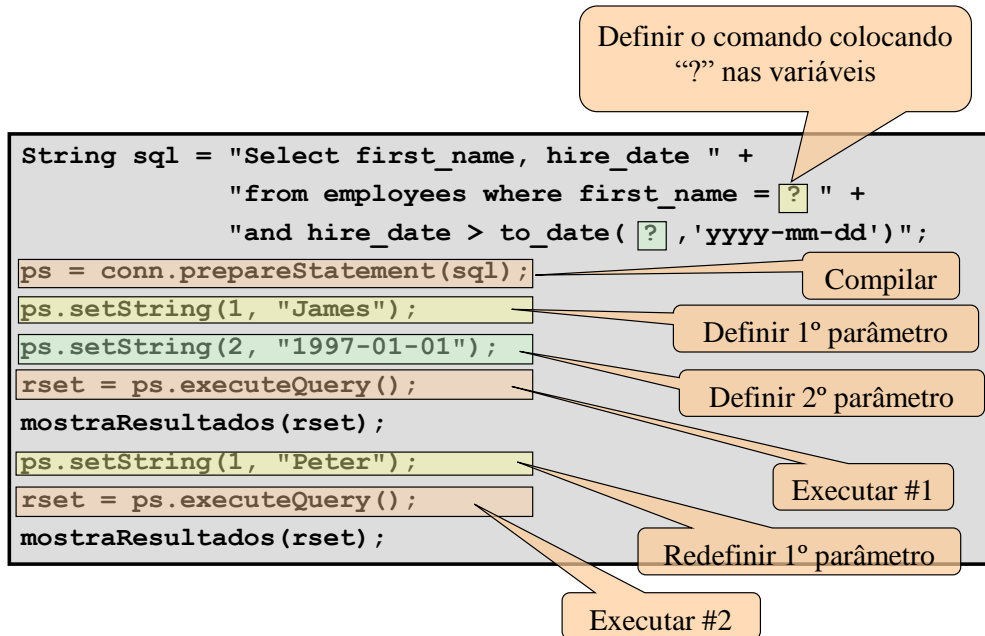
- O plano de execução não aproveita "*histograms*" (versões anteriores a 10g);
- O servidor não mostra o valor real com que a instrução é executada (1, 2 ou 3).
  - Pode ser contornado recorrendo a "trace files";



### 3.5.1.1- Vantagens e inconvenientes

O slide anterior descreve as vantagens e inconvenientes em usar PreparedStatements.

## SELECT com *PreparedStatement*



### 3.5.1.2- Select com PreparedStatement

O slide anterior mostra como executar uma instrução SELECT usando PreparedStatement.

## Exercícios

- Criar classe com SELECT à tabela EMPLOYEES sem *PreparedStatement*;
- Criar classe com SELECT à tabela EMPLOYEES com *PreparedStatement*;
- Verificar na BD o *parsing* dos *statements* anteriores.



### 3.5.2- Exercícios

Vamos criar dois pequenos programas que nos vão permitir verificar e validar o funcionamento do *PreparedStatement*. O primeiro não utiliza o *PreparedStatement* enquanto que o segundo utiliza-o, permitindo-nos verificar a redução de operações de “parsing” no segundo caso.

#### 3.5.2.1- Exercício – Sem *PreparedStatement*

Considere o seguinte programa que não utiliza *PreparedStatement*:



```
package jdbc;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import oracle.jdbc.pool.OracleDataSource;

public class PreparedStmt01 {

    public static void main(String[] args) {
        String url = "jdbc:oracle:thin:@//localhost:1521/XE";
        String user = "HR", pwd = "xpto1234";
        OracleDataSource ds;
```

```
Connection conn;
Statement stmt;
ResultSet rset;
try {
    ds = new OracleDataSource();
    ds.setUser(user);
    ds.setPassword(pwd);
    ds.setURL(url);
    conn = ds.getConnection();
    stmt = conn.createStatement();
    //stmt = conn.createStatement(
                                ResultSet.TYPE_SCROLL_INSENSITIVE,
                                ResultSet.CONCUR_READ_ONLY);
    //
    String sql
        = "Select first_name,last_name,hire_date from employees "
        + "where first_name = 'James' "
        + "and hire_date > to_date('1997-01-01','yyyy-mm-dd')";
    rset = stmt.executeQuery(sql);
    mostraResultados(rset);
    sql = "Select first_name,last_name,hire_date from employees "
        + "where first_name = 'Peter' "
        + "and hire_date > to_date('1997-01-01','yyyy-mm-dd')";
    rset = stmt.executeQuery(sql);
    mostraResultados(rset);
    rset.close();
    stmt.close();
    conn.close();
} catch (SQLException e) {
    e.printStackTrace(System.out);
}

public static void mostraResultados(ResultSet rset) {
    try {
        while (rset.next()) {
            System.out.println(rset.getString("first_name") + " "
                                + rset.getString("last_name") + " "
                                + rset.getDate("hire_date"));
        }
    } catch (SQLException e) {
        e.printStackTrace(System.out);
    }
}
}
```

Na base de dados exemplo que utilizamos neste curso o programa anterior produz o seguinte resultado:



James	Landry	1999-01-14
James	Marlow	1997-02-16
Peter	Vargas	1998-07-09
Peter	Tucker	1997-01-30
Peter	Hall	1997-08-20



As duas instruções SQL têm textos diferentes por causa das palavras “James” e “Peter”, pelo que não há reaproveitamento de “parsing” da primeira para a segunda.



### 3.5.2.2- Exercício – Com PreparedStatement

Vamos reescrever o programa anterior para que utilize PreparedStatement:



```
package jdbc;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import oracle.jdbc.pool.OracleDataSource;

public class PreparedStmt02 {
    public static void main(String[] args) {
        String url = "jdbc:oracle:thin:@//localhost:1521/XE";
        String user = "HR", pwd = "xpto1234";
        OracleDataSource ds;
        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rset = null;
        try {
            ds = new OracleDataSource();
            ds.setUser(user);
            ds.setPassword(pwd);
            ds.setURL(url);
            conn = ds.getConnection();
            String sql =
                "Select first_name,last_name,hire_date from employees " +
                "where first_name = ? " +
                "and hire_date > to_date( ? , 'yyyy-mm-dd')";
            ps = conn.prepareStatement(sql);
            //ps = conn.prepareStatement(sql, ResultSet.TYPE_FORWARD_ONLY,
            //                               ResultSet.CONCUR_READ_ONLY);
            ps.setString(1, "James");
            ps.setString(2, "1997-01-01");
            rset = ps.executeQuery();
            mostraResultados(rset);
            ps.setString(1, "Peter");
            rset = ps.executeQuery();
            mostraResultados(rset);
            rset.close();
            ps.close();
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace(System.out);
        }
    }

    public static void mostraResultados(ResultSet rset) {
        try {
            while (rset.next()) {
                System.out.println(rset.getString("first_name") + " " +
                    rset.getString("last_name") + " " +
                    rset.getDate("hire_date"));
            }
        } catch (SQLException e) {
            e.printStackTrace(System.out);
        }
    }
}
```

```

        e.printStackTrace(System.out);
    }
}

```



Este programa produz o mesmo resultado que o anterior, mas segue uma sequência de execução ligeiramente diferente. O PreparedStatement é um comando SQL que contém variáveis representadas por "?". A string que constitui o comando é enviada para a base de dados com essas variáveis, é compilada, mas não é executada, pois não tem valores concretos para correr (tem "?" em vez de valores).

Antes de executar a instrução é necessário substituir as variáveis por valores. Para isso usamos a instrução setString(arg1,arg2), cujo primeiro parâmetro identifica a variável ("?" ) pela sua posição na string e o segundo parâmetro o valor. Além de setString() ainda temos: setInt(), setLong(), setFloat(), setDate(), setBlob(), setObject().

Neste exemplo usamos o mesmo valor de data para as duas execuções, pelo que só é definida uma vez. A segunda execução aproveita a definição de data da primeira execução.

### 3.5.2.3- Verificar a redução de "parsing"

A redução de parsing pode ser vista executando o comando abaixo na base de dados. Este comando requer que o utilizador que o executa tenha privilégio de select à vista v\$sqlarea. O utilizador HR não tem esse privilégio mas o SYSTEM ou SYS têm-no.



```

select executions,loads, Parsing_schema_name,sql_text
from v$sqlarea
where sql_text like 'Select first_name,last_name,hire_date%';

```

Se as duas classes dos exemplos anteriores forem executadas duas vezes, o comando acima produz o seguinte resultado:

Executions	Loads	P_S_N	SQL_Text
4	1	HR	Select first_name,last_name,hire_date from employees where first_name = :1 and hire_date > to_date( :2,'yyyy-mm-dd')
2	1	HR	Select first_name,last_name,hire_date from employees where first_name = 'James' and hire_date > to_date('1997-01-01','yyyy-mm-dd')
2	1	HR	Select first_name,last_name,hire_date from employees where first_name = 'Peter' and hire_date > to_date('1997-01-01','yyyy-mm-dd')



O quadro anterior mostra que com PreparedStatement temos 100% de reaproveitamento, ou seja, 1 load (parsing) e 4 execuções. Sem prepared statement há reaproveitamento entre diferentes execuções da mesma instrução, mas não há 100% de reaproveitamento.



A instrução é executada quando os parâmetros lhe são enviados mas usa o plano de execução que foi definido na altura do “parsing”. Este plano foi encontrado sem valores concretos nas variáveis e por isso o Oracle não pode tirar partido de “histograms” que estejam definidos sobre os valores da tabela. Para saber mais sobre “histograms” consulte um manual de optimização de SQL em Oracle ou o livro “Oracle Concepts” que faz parte da documentação que acompanha a base de dados;



***Na maior parte das situações compensa usar PreparedStatement em vez de Statement***

## Sumário

---

- Preparar o ambiente de trabalho;
- JDBC e tipos de *drivers*;
- Executar SELECT;
- Executar DDL e DML;
- *PreparedStatement*;
- Executar *stored procedure* e *function*;
- SQL *Injection*
- Segurança dos dados - Cifra



### 3.6- Executar *stored procedure* e *function*

Vamos ver como executar uma “stored procedure” e uma “function” numa base de dados Oracle.

## Como invocar uma “*stored procedure*”?

- Definir instrução com ? nos parâmetros;
- Definir os parâmetros de entrada, saída, e entrada e saída;
- Executar a instrução;
- Apanhar os parâmetros de saída.

```
cs = conn.prepareCall("begin get_max_sal_jobid(?,?,?);end;");
cs.setString(1, "SA_MAN");
cs.registerOutParameter(2, java.sql.Types.DOUBLE);
cs.registerOutParameter(3, java.sql.Types.DOUBLE);
cs.executeUpdate();
System.out.println("Max Salary=" + cs.getDouble(2));
System.out.println("Max Commission=" + cs.getDouble(3));
```



### 3.6.1- Como invocar uma “*stored procedure*”?

Uma “*stored procedure*” é código PL/SQL ou Java que está guardado dentro da base de dados. As instruções do slide mostram como despoletar a execução da “*stored procedure*”, como passar-lhe parâmetros e como capturar os seus resultados.

Esta *stored procedure* recebe o primeiro parâmetro e devolve os dois seguintes. Estes têm que ser registados no Java como “Output Parameter”.

## Como invocar uma “function”?

```
cs = conn.prepareCall(  
    "begin; ?:= sql_capture_create_session(?,?,?); end;");
```

- O primeiro "?" deve ser registado como "*output parameter*";
- Os outros são idênticos a “*stored procedure*”.



### 3.6.2- Como invocar uma função?

Uma função é código PL/SQL ou Java que está guardado dentro da base de dados. Assim como as stored procedures, as funções podem ter parâmetros de entrada de dados, de saída ou de entrada e saída. Ao contrário da stored procedure, que não devolve valor, a função devolve sempre um valor, que tem que ser registado como “output parameter”.

A maneira de invocar uma função é semelhante à de uma “stored procedure”.

## Exercícios

- Criar uma “*stored procedure*”;
- Criar uma classe Java que invoca essa “*stored procedure*”.



### 3.6.3- Exercícios

#### 3.6.3.1- Exercício – Criar uma “stored procedure”

Neste exercício vamos criar uma “stored procedure” na base de dados, escrita em PL/SQL. Esta recebe um valor de JOB\_ID e determina o salário mais alto e a comissão mais alta de todos os empregados dessa categoria (JOB\_ID). Este procedimento tem um parâmetro de entrada e dois de saída.


Vamos usar o cliente Oracle que está incluído no IDE para ligar à base de dados e executar o código PL/SQL:-

1. Abrir o separador **Services**;
2. Escolher **Databases → Drivers**;
3. Adicionar um driver seguindo as indicações dadas no início do capítulo;
4. Para criar uma ligação à base de dados:
  - 4.1. Escolher o driver criado no passo 3 e abrir o menu sensível ao contexto (botão do lado direito do rato) e escolher **Connect Using**;
  - 4.2. No “wizard” de criação de ligação introduzir os seguintes dados:
    - 4.2.1. Connection Name: **LocalHostXE**;
    - 4.2.2. Connection type: **Oracle (JDBC)**;
    - 4.2.3. User name: **HR**;

- 4.2.4. Password: **xpto1234**;
- 4.2.5. Driver: **thin**;
- 4.2.6. Host Name: **localhost**;
- 4.2.7. JDBC Port: **1521**;
- 4.2.8. SID: **XE**.

5. Para executar o código PL/SQL e criar a “stored procedure”:

- 5.1. Seleccionar a ligação que acabou de ser criada (**LocalHostXE**);
- 5.2. Abrir o menu sensível ao contexto com o botão do lado direito do rato e escolher **Execute command**;
- 5.3. Introduzir o código apresentado abaixo:




```
create or replace procedure get_max_sal_jobid (  
    p_job_id in employees.job_id%type,  
    p_max_sal out employees.salary%type,  
    p_max_com out employees.commission_pct%type)  
is  
begin  
    select max(salary) ,nvl(max(commission_pct),0)  
    into p_max_sal, p_max_com  
    from employees  
    where job_id like p_job_id;  
end;
```

- 6. Executar;
- 7. Para confirmar a criação do procedimento vamos navegar nos objetos do utilizador HR:
  - 7.1. No separador Connections escolher a ligação e abrir;
  - 7.2. Ver os objetos do tipo Procedure.

### 3.6.3.2- Exercício – Invocar a “stored procedure”

Depois de criada a “stored procedure” no exercício anterior, vamos forçar a sua execução a partir de um programa Java, que irá enviar um parâmetro e receber duas respostas.

A classe apresentada abaixo permite invocar este procedimento, passando-lhe um JOB\_ID e recebendo os seguintes valores: maior salário e maior valor de comissão.



```
package jdbc;  
  
import java.sql.CallableStatement;  
import java.sql.Connection;  
import java.sql.SQLException;  
  
import oracle.jdbc.pool.OracleDataSource;  
  
public class StoredProcedure {  
    public static void main(String[] args) {  
        String url = "jdbc:oracle:thin:@//localhost:1521/XE";  
        String user = "HR", pwd = "xpto1234";  
        OracleDataSource ds;  
        Connection conn = null;  
        CallableStatement cs = null;
```



```
try {
    ds = new OracleDataSource();
    ds.setUser(user);
    ds.setPassword(pwd);
    ds.setURL(url);
    conn = ds.getConnection();
    conn.setAutoCommit(false);
    cs = conn.prepareCall("begin get_max_sal_jobid(?,?,?); end;");
    cs.setString(1, "SA_MAN");
    cs.registerOutParameter(2, java.sql.Types.DOUBLE);
    cs.registerOutParameter(3, java.sql.Types.DOUBLE);
    cs.executeUpdate();
    System.out.println("Max Salary=" + cs.getDouble(2));
    System.out.println("Max Commission=" + cs.getDouble(3));
} catch (SQLException e) {
    e.printStackTrace(System.out);
} finally {
    try {
        if (cs != null) {
            cs.close();
        }
        if (conn != null) {
            conn.close();
        }
    } catch (SQLException e2) {
        e2.printStackTrace(System.out);
    }
}
```



A estrutura da classe é semelhante ao exemplo de PreparedStatement, mas agora usamos a classe CallableStatement.



A instrução SQL é introduzida como um bloco anónimo PL/SQL. Os pontos de interrogação são colocados nas posições que irão receber parâmetros e são numerados da esquerda para a direita. Os parâmetros podem ser de entrada, de saída ou de entrada/saída.



Para definir os parâmetros que o Java vai enviar (e portanto o PL/SQL vai receber) usamos o método `cs.setString()`, sendo o primeiro argumento a posição do ponto de interrogação que nos interessa e o segundo argumento o valor que queremos passar para a “stored procedure”. Existem outros métodos como `setInt()` ou `setDouble()`.



Para definir os parâmetros que o Java vai receber (e portanto o PL/SQL vai enviar) usamos o método `cs.registerOutParameter()`. O primeiro argumento indica a posição do ponto de

interrogação que nos interessa, enquanto que o segundo argumento indica o tipo de dados Java para onde se vai converter o valor recebido da base de dados.

## Sumário

---

- Preparar o ambiente de trabalho;
- JDBC e tipos de *drivers*;
- Executar SELECT;
- Executar DDL e DML;
- *PreparedStatement*;
- Executar *stored procedures* e *function*;
- **SQL Injection**
- Segurança dos dados - Cifra



### 3.7- SQL Injection

## SQL Injection – O problema

- É uma **vulnerabilidade** das aplicações;
- Permite que um utilizador mal intencionado **injecte** SQL no código executado pela aplicação;
- O **SQL injectado** permite:
  - **Superar** os mecanismos de segurança da aplicação;
  - Consultar dados **confidenciais**;
  - **Alterar** o fluxo de execução da aplicação;
  - **Aceder** a áreas restritas do código da aplicação;



### 3.7.1- O problema

O “SQL Injection” é uma vulnerabilidade que permite a um utilizador mal intencionado executar instruções SQL para além daquelas que foram programadas numa aplicação, obtendo dados confidenciais, alterando o fluxo normal de execução do código da aplicação e superando os mecanismos de segurança. Todas as aplicações são potencialmente vulneráveis a “SQL Injection”, especialmente as desenvolvidas para a WEB.

As aplicações informáticas que lidam com diferentes utilizadores têm necessidade de autenticar a pessoa. O mecanismo mais usual de autenticação é *login/password* (ou *username/password*). O *login* é o nome pelo qual a pessoa é conhecida na aplicação. A *password* é algo que só a própria pessoa conhece e que portanto permite a sua autenticação. Utilizando “SQL Injection” uma pessoa mal intencionada pode contornar o mecanismo de autenticação de uma aplicação e assumir a identidade de um utilizador válido.

## SQL Injection – Exemplo # 1/2

- Exemplo de código para validar utilizador:

```
String query = "SELECT 1 FROM employees WHERE email = '"  
              + login + "' and password = '" + password + "'";  
Statement st = conn.createStatement();  
rset = st.executeQuery(query);  
if (rset.next()) {  
    return true;  
} else {  
    return false;  
}
```

O query é construído  
usando os conteúdos das  
variáveis de entrada

Se a linha existir na tabela,  
então o utilizador está  
validado



## SQL Injection – Exemplo #2/2

- Suponha que um utilizador mal intencionado introduz a seguinte password:

```
xpto' or '1'='1'
```

- O query gerado dinamicamente fica assim:

```
SELECT 1 FROM employees  
WHERE email = 'XPTO' and password = 'xpto' or '1'='1'
```

- **O utilizador injectou SQL;**
- **Com isto conseguiu a autenticação, mesmo sem saber a password!**



## SQL Injection – A solução

- Limitar a contribuição do utilizador na construção do SQL;
- Controlar essa contribuição fazendo validações;
- Usar PreparedStatement em vez de Statement;

```
String query = "SELECT password FROM employees WHERE email = ? ";
PreparedStatement ps = conn.prepareStatement(query);
ps.setString(1, login);
this.rset = ps.executeQuery();
if (rset.next()) {
    if (password.equals(rset.getString("password"))) {
        return true;
    } else {
        return false;
    }
}
```



### 3.7.2- A solução

O query utilizado no exemplo anterior para autenticar o utilizador é construído dinamicamente, sendo uma parte dele formada por inputs pedidos ao utilizador. Isto dá ao utilizador o poder de injetar SQL. Para reduzir a vulnerabilidade da nossa aplicação ao “SQL Injection” devemos seguir os princípios abaixo:

- Limitar ao máximo a contribuição do utilizador na construção dinâmica do SQL;
- Controlar essa contribuição fazendo validações;

Estes princípios traduzem-se em medidas concretas:

- Usar PreparedStatement em vez de Statement, o que limita o âmbito das variáveis e portanto dificulta a injeção de SQL;
- Reescrever o query para que devolva a password de um determinado login, sendo feita em Java a comparação das passwords;

## Exercícios

- Adicionar a coluna password na tabela Employees;
- Criar a classe `EmployeeDataHandler` que liga à BD e autentica o utilizador:
  - Usando código vulnerável;
  - Corrigindo as vulnerabilidades;



### 3.7.3- Exercícios

Neste exercício adicionar à tabela EMPLOYEES a coluna PASSWORD e em seguida vamos desenvolver a classe **EmployeeDataHandler** que valida o utilizador da aplicação.

#### 3.7.3.1- Adicionar a coluna password à tabela Employees

##### 1. Adicionar a coluna PASSWORD à tabela EMPLOYEES:

- 1.1. Usando a ligação à base de dados definida nos exercícios anteriores deste capítulo, abrir uma “SQL Worksheet” e executar o comando abaixo:



```
alter table employees add(password varchar2(100));
```

- 1.2. Alterar todas as linhas da tabela EMPLOYEES de forma que a coluna password receba a concatenação das colunas FIRSTNAME com LASTNAME, o que é feito com o seguinte comando SQL sobre a “SQL Worksheet”:



```
update employees set password = first_name || ' ' || last_name;  
commit;
```

- 1.3. Alterar a coluna password no sentido de a tornar NOT NULL:



```
alter table employees modify (password not null);
```

### 3.7.3.2- Exercício - Criar a classe que autentica o utilizador usando código vulnerável

2. Seleccionar o projecto **JDBC**;
3. Criar uma nova Java Class
4. No *wizard* de criação da classe dar como nome **EmployeeDataHandler**, não aceitar "Generate Default Constructor" mas aceitar "Generate Main Method";
5. A ligação à base de dados usará os seguintes parâmetros:  
Host: **localhost** (ou o nome/ip da máquina onde corre o servidor)  
Port: **1521** (ou o porto onde o listener está à escuta)  
SID: **XE** (ou o valor fornecido pelo administrador da base de dados)  
Username: **HR**;  
Password: **xpto1234** (definida durante a instalação do Oracle XE);
6. Introduzir o código apresentado abaixo;



```
package jdbc;

import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;
import oracle.jdbc.pool.OracleDataSource;

public class SQLInjection01 {
    private String erro = null;
    private Connection conn = null;;
    private ResultSet rset = null;

    public SQLInjection01(String jdbcUrl, String userid, String password) {
        //A criação deste objeto só é valida se conn for instanciada sem
        problemas
        try {
            OracleDataSource ds;
            ds = new OracleDataSource();
            ds.setURL(jdbcUrl);
            this.conn = ds.getConnection(userid, password);
            this.erro = null;
        } catch (SQLException e) {
            this.conn = null;
            this.erro = e.toString();
        }
    }

    public boolean authenticateLogin (String login, String password) {
        //versão vulneravel a SQL Injection
        try {
            String query = "SELECT 1 FROM employees WHERE email = '" +
login +
                        "' and password = '" + password + "'";
            Statement st =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
            System.out.println("Executando o query: " + query);
            rset = st.executeQuery(query);
            if (rset.next()) {
                //encontrou o email e password introduzidos pelo utilizador
```



```
        this.erro = null;
        return true;
    } else {
        //não encontrou o email e password introduzidos pelo
utilizador
        this.erro = "Login inválido ou password errada.";
        return false;
    }
} catch (SQLException e) {
    this.erro = e.toString();
    return false;
}
}

public String getErro() {
    return this.erro;
}

public Connection getConn() {
    return conn;
}

public static void main(String[] args) throws Exception {
    SQLInjection01 dh = new SQLInjection01(
        "jdbc:oracle:thin:@localhost:1521:XE", "HR", "xpto1234");
    if (dh.getConn() == null) {
        //a ligação à base de dados não foi criada
        System.out.println(dh.getErro());
        System.exit(0);
    }
    String login="DLEE", password="David Lee";
    //String login="UmQualquer", password = "xpto' or '1'='1";
    //String login="DLEE", password = "Serra D'Aires";

    if (dh.authenticateLogin(login,password)) {
        System.out.println("Login válido");
    } else {
        System.out.println(dh.getErro());
    }
    System.out.println("Login=" + login);
    System.out.println("Password=" + password);
}
}
```



O construtor desta classe inicializa a conexão à base de dados. Se ocorrer um erro na abertura da ligação, por exemplo porque a *password* de HR está errada ou porque o porto do listener está errado, o objecto Connection recebe null, sendo actualizada a mensagem de erro. A classe que cria um objecto de DataHandler tem que validar se a criação da conexão teve sucesso.



O método que faz a autenticação usa um query que devolve 1 se encontrar linhas que satisfaçam a condição de pesquisa.



O método main permite testar o funcionamento da classe. Começa por verificar se a ligação à base de dados teve sucesso, para depois fazer a validação de um login/password, que são variáveis do tipo String. Este algoritmo simula uma aplicação, que usaria um TextFied para pedir ao utilizador estes dados.



Se executarmos a classe acima obtemos o seguinte resultado:



```
Executando o query: SELECT 1 FROM employees WHERE email = 'DLEE' and password =  
'David Lee'  
Login válido  
Login=DLEE  
Password=David Lee
```

Agora suponha que um utilizador mal intencionado introduz na password a String apresentada a seguir. Tenha atenção à posição das plicas, pois é muito importante para que o exemplo funcione correctamente:

```
xpto' or '1'='1'
```

Isto cria uma condição sempre verdadeira, o que vai validar qualquer login que ele escolha. Esta situação pode ser simulada alterando as duas linhas abaixo no programa inicial:



```
. . .  
    //String login = "DLEE";  
    //String password = "David Lee";  
    String login = "XPTO";  
    String password = "xpto' or '1'='1';  
. . .
```



A execução do programa produz o seguinte resultado:



```
Executando o query: SELECT 1 FROM employees WHERE email = 'XPTO' and password =  
'xpto' or '1'='1'  
Login válido  
Login=XPTO  
Password=xpto' or '1'='1'
```



O utilizador acabou de injetar no SQL uma condição sempre verdadeira, fazendo com que a aplicação o autenticasse com o login XPTO, que não existe na base de dados. Pior seria se ele usasse um Login que existe na base de dados, por exemplo DLEE. Não é difícil para um “hacker” encontrar logins válidos.



***A String xpto' or '1'='1 é um exemplo de SQL Injection***



A solução anterior apresenta ainda outro problema não relacionado com segurança mas com impacto na “qualidade do serviço”: suponha que a password do utilizador é a String: **Serra D'Aires**. Para fazer o teste introduzimos a seguinte instrução:



```
. . .
    //String login = "DLEE";
    //String password = "David Lee";
    String login = "XPTO";
    //String password = "xpto' or '1'='1";
    String password = "Serra D'Aires";
. . .
```



A execução do programa produz o seguinte resultado:



```
Executando o query: SELECT 1 FROM employees WHERE email = 'DLEE' and password =
'Serra D'Aires'
java.sql.SQLException: ORA-00933: SQL command not properly ended

Login=DLEE
Password=Serra D'Aires
```



A plica contida na password invalida a instrução SQL e provoca o erro. Para evitar este problema temos que percorrer a string e, sempre que encontrar-mos uma plica, substituí-la por duas plicas consecutivas, para que a instrução seja aceite pelo interpretador de comandos SQL.

### 3.7.3.3- Exercício - Corrigir as vulnerabilidades

O query utilizado no exemplo anterior para autenticar o utilizador é construído dinamicamente, pois uma parte dele é formada por inputs pedidos ao utilizador. Isto dá ao


utilizador o poder de injetar SQL. Para reduzir a vulnerabilidade da nossa aplicação ao “SQL Injection” devemos seguir os princípios abaixo:

- Limitar ao máximo a contribuição do utilizador na construção dinâmica do SQL;
- Controlar essa contribuição fazendo validações;

Estes princípios traduzem-se em medidas concretas:

- Usar PreparedStatement em vez de Statement, o que limita o âmbito das variáveis e portanto dificulta a injeção de SQL;
- Reescrever o query para que devolva a password de um determinado login, sendo feita em Java a comparação das passwords;

No exemplo abaixo a função authenticateLogin() vai ser reescrita seguindo as recomendações anteriores. Isto obriga a alterar os imports por causa de PreparedStatement:



```
...
//import java.sql.Statement;
import java.sql.PreparedStatement;
...

...
public boolean authenticateLogin (String login, String password) {
    //versão mais resistente a SQL Injection
    try {
        String query = "SELECT password FROM employees WHERE email =
?";
        PreparedStatement ps = conn.prepareStatement(query,
ResultSet.TYPE_FORWARD_ONLY,ResultSet.CONCUR_READ_ONLY);
        ps.setString(1,login);
        System.out.println("\nExecuting query: " + query);
        this.rset = ps.executeQuery();
        if (rset.next()) {
            //encontrou email
            //System.out.println(rset.getString("password"));
            if (password.equals(rset.getString("password"))) {
                this.erro = null; //a password é igual
                return true;
            } else {
                this.erro = "Login inválido ou password errada.";
                return false; //a password é diferente
            }
        } else {
            //não encontrou o email introduzido pelo utilizador
            this.erro = "Login inválido ou password errada.";
            return false;
        }
    } catch (SQLException e) {
        this.erro = e.toString();
        return false;
    }
}
...
```



A execução desta nova versão resiste a esta tentativa de SQL Injection:



```
Login inválido ou password errada.  
Login=XPTO  
Password=xpto' or '1'='1
```



A alteração do query reduz as possibilidades de escrita de SQL injetado, pois diminui os graus de liberdade;



A utilização de PreparedStatement ajuda nas validações de input do utilizador, pois a instrução SQL é compilada com um ponto de interrogação na variável, sendo esta depois substituída pela String introduzida pelo utilizador. Qualquer texto que este introduza em excesso é interpretado como um nome de Login e não como cláusula adicional da instrução SQL ou uma nova instrução SQL.



Esta solução ultrapassa também o problema da plica, pois a instrução SQL já foi interpretada e portanto a plica não tem influência no “*parsing*” do query.

## Sumário

---

- Preparar o ambiente de trabalho;
- JDBC e tipos de *drivers*;
- Executar SELECT;
- Executar DDL e DML;
- *PreparedStatement*;
- Executar *stored procedures* e *function*;
- *SQL Injection*
- Segurança dos dados - Cifra



### 3.8- Segurança dos dados – Cifra

## Cifra de dados

- A cifra torna os dados **ilegíveis**;
- Todos os dados confidenciais devem ser cifrados:
  - Dados pessoais de clientes;
  - Salários;
  - Vendas / Compras;
- Normalmente não é necessário cifrar todo o conteúdo, mas apenas a **referência** que permite identificar a quem pertence o dado;
- A cifra pode ser feita pelo Java ou pelo SGBD;



### 3.8.1- Cifra de dados

Para aumentar os níveis de segurança das aplicações é frequente cifrar os dados armazenados nas tabelas que contêm informação sensível. Além de ser uma boa prática, é uma recomendação Sarbanes-Oxley, que todas as empresas cotadas na bolsa de Nova York têm que seguir.

Isto obriga a que:

- Os dados sejam cifrados antes de serem armazenados na tabela;
- Os dados sejam decifrados antes de serem mostrados ao utilizador;

A cifra torna os dados ilegíveis, estando disponíveis na NET, em regime de open source, vários algoritmos de cifra. O algoritmo de cifra pode ser feito pelo Java ou pela base de dados, sendo mais frequente a segunda solução, já que os dados ficam disponíveis de forma independente do programa.

O trabalho com passwords cifradas tem dificuldade adicional. Para simplificar o programa devemos fazer comparações com valores cifrados, o que cria as situações abaixo:

- Se o algoritmo de cifra estiver no programa Java, então ciframos a password introduzida pelo utilizador, recolhemos a password que está na base de dados e comparamos as versões cifradas no Java;
- Se o algoritmo de cifra estiver na base de dados, então escrevemos um query que recolha duas colunas: uma é a password que está na base de dados, a outra é uma invocação da função de cifra que recebe como argumento a password introduzida pelo utilizador. Desta forma recebemos em Java os dois valores cifrados, o que nos permite fazer a comparação;

## Cifra de dados - Exemplo

```
import java.security.MessageDigest;

public static String codificaPassword(String password) {
    MessageDigest digest = null;
    try {
        digest = MessageDigest.getInstance("sha");
    } catch (Exception e) {
        e.printStackTrace();
    }
    byte[] digestedPassword = password.getBytes();
    digest.update(digestedPassword);
    digestedPassword = digest.digest();
    return new String(digestedPassword);
}
```

Chave de cifra



Java Web  
© Citeforma

Capítulo 3 - Acesso a base de dados com JDBC

53

### 3.8.2- Exemplo

O algoritmo do slide permite cifrar uma String, usando uma chave de cifra.



## Exercício

- Escrever uma classe que faça a cifra de uma String;



### 3.8.3- Exercício – Cifrar uma String

Escrever uma classe que faça a cifra de uma String:

```
package jdbc;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class CodificadorPassword {

    public static String codificaPassword(String password) {
        MessageDigest digest = null;
        try {
            digest = MessageDigest.getInstance("sha");
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace(System.out);
            return null;
        }
        byte[] digestedPassword = password.getBytes();
        digest.update(digestedPassword);
        digestedPassword = digest.digest();
        return new String(digestedPassword);
    }

    public static void main(String[] args) {
        String myPass = "jaser";
        String myPass2 = "m";
        System.out.println("Password: " + myPass);
    }
}
```

```
        System.out.println("COD: " +  
CodificadorPassword.codificaPassword(myPass));  
        System.out.println("-----");  
        System.out.println("Password2: " + myPass2);  
        System.out.println("COD: " +  
CodificadorPassword.codificaPassword(myPass2));  
    }  
}
```



A execução desta classe produz o seguinte resultado:



```
Password: jaser  
COD: èâÆ•_ââ[ÿBWF^F;^4'  
-----  
Password2: m  
COD: k1ÀÕc"0$ÚEi,,d:ÇŒ-è
```

## Sumário

---

- Preparar o ambiente de trabalho;
- JDBC e tipos de *drivers*;
- Executar SELECT;
- Executar DDL e DML;
- *PreparedStatement*;
- Executar *stored procedures* e *function*;
- *SQL Injection*
- Segurança dos dados - Cifra

