estgoh
Politécnico de Coimbra

Programação Aplicada

Acesso a Bases de Dados relacionais através de JDBC

Marco Veloso
marco.veloso@estgoh.ipc.pt

Agenda

Acesso a Bases de Dados relacionais através de ODBC

***Packages*, *Code Conventions*, Formatações e Administração**

Debugging & Logging

Acesso a Bases de Dados relacionais através de JDBC

Sistemas Concorrentes (*Threads*)

Comunicação em Rede (*Sockets*)

Interfaces Gráficos

Agenda

Acesso a Bases de Dados relacionais através de ODBC

Java DataBase Connectivity (JDBC)

Acesso a bases de dados relacionais

Prepared Statements

SQLJ

JDBC

Em alternativa aos ficheiros para armazenamento de dados de forma persistente existem as **bases de dados relacionais**, geridas por um **Sistema Gestor de Bases de Dados** (SGBD, *e.g.* MySQL, Oracle, SQL Server, PostgreSQL, *etc.*)

Até ao momento, o **acesso a uma base de dados foi sempre realizado através de uma aplicação intermediária** (*e.g.* HeidiSQL)

Este **acesso é transparente** para o utilizador, apenas necessitando de especificar o **IP** da máquina e **porto** da máquina que disponibiliza o serviço, o respectivo ***login*** e ***password*** para a aplicação estabelecer a ligação

Porém, por vezes é necessário, no desenvolvimento das nossas próprias aplicações, estabelecer ligações directas a bases de dados

O acesso a partir de aplicação em *Java* a uma base de dados pode ser realizado através de um **JDBC** (***Java DataBase Connectivity***).

Trata-se de uma **interface SQL para acesso a diversas bases de dados do tipo relacional**, permitindo uma maior liberdade no desenvolvimento de aplicações

Este tipo de interfaces designa-se genericamente por **ODBC** (***Open DataBase Connectivity***), permitindo que qualquer aplicação aceda a uma base de dados específica (no caso concreto, MySQL).

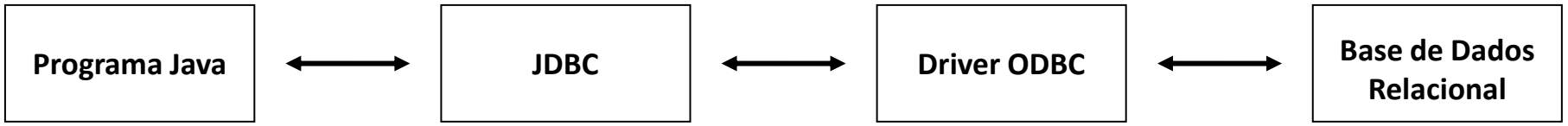
O termo **JDBC é uma especialização do ODBC** genérico, sendo um ODBC específico para aplicações desenvolvidas sobre linguagem Java.

Para efectuar a ligação entre uma aplicação *Java* e uma determinada base de dados é necessário existir um ***driver*** especifico que faça a **ligação entre o JDBC e a base de dados** em causa

No entanto, e dado o crescimento da plataforma *Java*, muitos fabricantes das principais bases de dados disponibilizam ***drivers* JDBC** para as bases de dados

A *Sun* / Oracle criaram o "**JDBC API**", uma especificação com um conjunto de classes, interfaces e métodos, com o objectivo de ter uma interface única e independente da base de dados para as suas aplicações *Java*

A ideia é que o **código de ligação à base de dados é sempre o mesmo**, mas **cada base de dados terá um *driver* específico** (disponibilizado pelo fabricante) que fará a ligação



No entanto, e dada a existência de uma grande quantidade de aplicações que utilizam *drivers* ODBC para ligação a bases de dados (Oracle, Access, SQL Server, etc.), é também disponibilizado no SDK um produto chamado "***Bridge***", um *driver* JDBC que transforma as chamadas JDBC em O**JDBC-ODBC** DBC.

Desta forma, os programas *Java* que implementem acesso SQL a bases de dados podem aceder de forma transparente a bases de dados

Acesso a bases de dados relacionais

Acesso a bases de dados relacionais: Visão global

Acesso a Bases de Dados relacionais através de ODBC

```
Connection conn = null;
Statement st = null;
ResultSet rs = null;

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

     conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/bd", "root", "root");
     st = conn.createStatement();

     rs = st.executeQuery(" SELECT * FROM java_test ");

      if (rs == null || !rs.next()) {
         System.out.println("!! No Record on table !!");
      } else
         while (rs.next()) {
            int    column1 = rs.getInt("column1");
            String column2 = rs.getString("column2");
            System.out.println("Column1 = "+ column1 + " ; Column2 = "+column2);
         }
} catch (SQLException e) {
    System.out.println("!! SQL Exception !!\n"+e);
    e.printStackTrace();
} catch (ClassNotFoundException e) {
        System.out.println("!! Class Not Found. Unable to load Database Drive !!\n"+e);
} catch (IllegalAccessException e) {
        System.out.println("!! Illegal Access !!\n"+e);
} catch (InstantiationException e) {
        System.out.println("!! Class Not Instanciaded !!\n"+e);
} finally {
     if (st != null) {
         try {
            st.close();
         } catch (Exception e) {
            System.out.println("!! Exception returning statement !!\n"+e);
         }
     }
     if (conn != null) {
         try {
            conn.close();
         } catch (Exception e) {
            System.out.println("!! Exception closing DB connection !!\n"+e);
         }
     }
} // end of finally
```

Etapa 1: Carregar driver

Etapa 2: Estabelecer ligação

Etapa 3: Inicializar o objecto `Statement`

Etapa 4: Execução da *query* SQL

Etapa 5: Iteração sobre os registos da tabela (`ResultSet`)

Tratamento de excepções

Etapa 6: Encerrar e libertar os recursos (`Statement` e `Connection`)

Classes para manipulação de bases de dados

Acesso a Bases de Dados relacionais através de ODBC

As classes necessárias para manipulação de bases de dados encontram-se na *package* `java.sql.*`, sendo necessário proceder à sua importação:

```
import java.sql.*;
```

Para a manipulação de bases de dados relacionais é necessário recorrer às seguintes classes, em **6 etapas**:

`DriverManager`	responsável por **inicializar o *driver***;
`Connection`	responsável por **estabelecer a ligação à base de dados** e remeter comandos de administração;
`Statement / PreparedStatement`	responsável por **execução dos comandos (*statements*) SQL**;
`ResultSet`	responsável pelo **armazenamento da informação devolvida pela base de dados**;

Estabelecer ligação a uma base de dados relacional

Acesso a Bases de Dados relacionais através de ODBC

Etapa 1: O **driver para ligação** é carregado através da instrução:

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

Nas recentes versões da biblioteca Java não é necessário carregar explicitamente o driver.

Etapa 2: A **ligação é estabelecida** através da do método `getConnection` que devolve um objecto `Connection`:

```
Connection conn =
 DriverManager.getConnection("jdbc:mysql://localhost/db");
```

Como argumento do método `getConnection`, a **sintaxe de ligação** da base de dados é indicada pela instrução "`jdbc:mysql://localhost/db`", ou seja:

localização do servidor (*e.g.* `localhost` ou `127.0.0.1`), e

nome da base de dados (*e.g.* `db`).

Estabelecer ligação a uma base de dados relacional

O driver ODBC é disponibilizado pelo próprio fabricante do Sistema Gestor de Base de Dados. No caso de um SGBD MySQL, **o driver ODBC (JDBC) deve corresponder à versão do servidor MySQL** e pode ser obtido de:

http://www.mysql.com/products/connector/j/

https://dev.mysql.com/downloads/connector/j/

Caso se use um **driver JDBC inferior à versão 8.x** (e.g. 5.1.47), O **driver para ligação** é carregado através da instrução indicada na página anterior:

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

No entanto, se o **driver JDBC for superior à versão 8.x** (e.g. 8.0.22), O **driver para ligação** é carregado através da seguinte instrução:

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

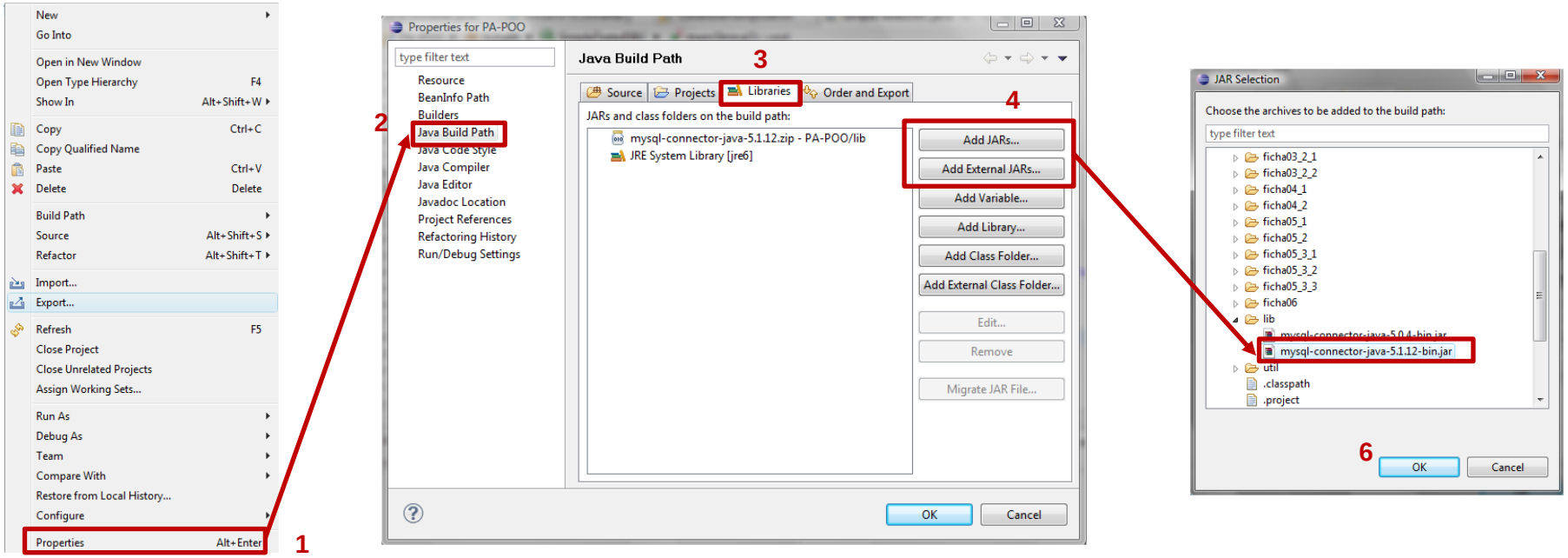
Ou seja, o driver deixa de ser "`com.mysql.jdbc.Driver`" e passa a ser "`com.mysql.cj.jdbc.Driver`" e não é necessário criar uma nova instância (`.newInstance()`)

Estabelecer ligação a uma base de dados relacional

Acesso a Bases de Dados relacionais através de ODBC

No **IDE Eclipse**, para o compilador carregar o driver JDBC é **necessário indicar onde se encontra fisicamente o respectivo JAR**.

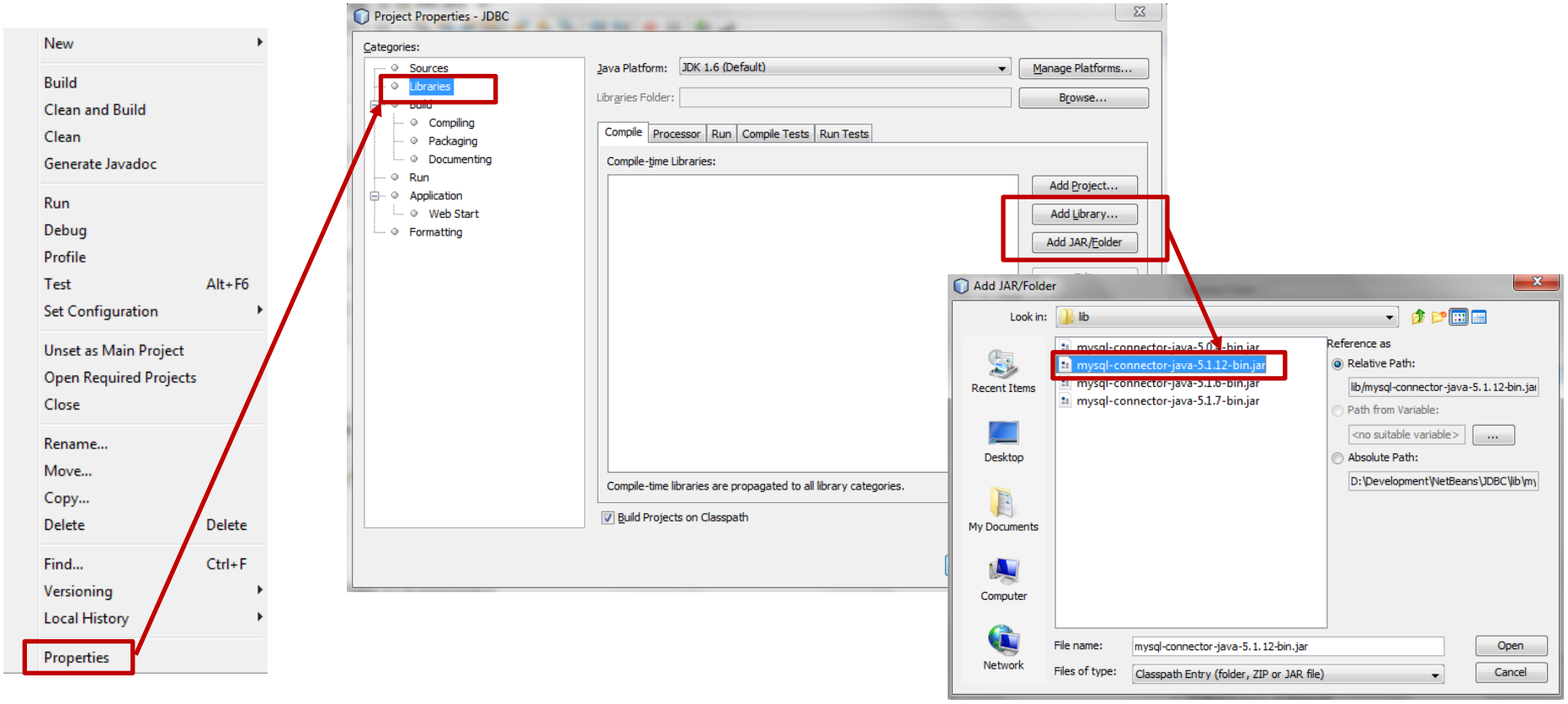
Para o efeito deve-se aceder às **propriedades do projecto** (clicar com o botão direito sobre o projecto e seleccionar **`Properties`** no meu) e na secção **`Java Build Path`**, selecionar o botão **`Add JARs`** ou **`Add External JARs`** (consoante o ficheiro se encontre numa directoria interna ou externa à estrutura do projecto) das bibliotecas (**`Libraries`**) e seleccionar o ficheiro JAR correspondente:



Estabelecer ligação a uma base de dados relacional

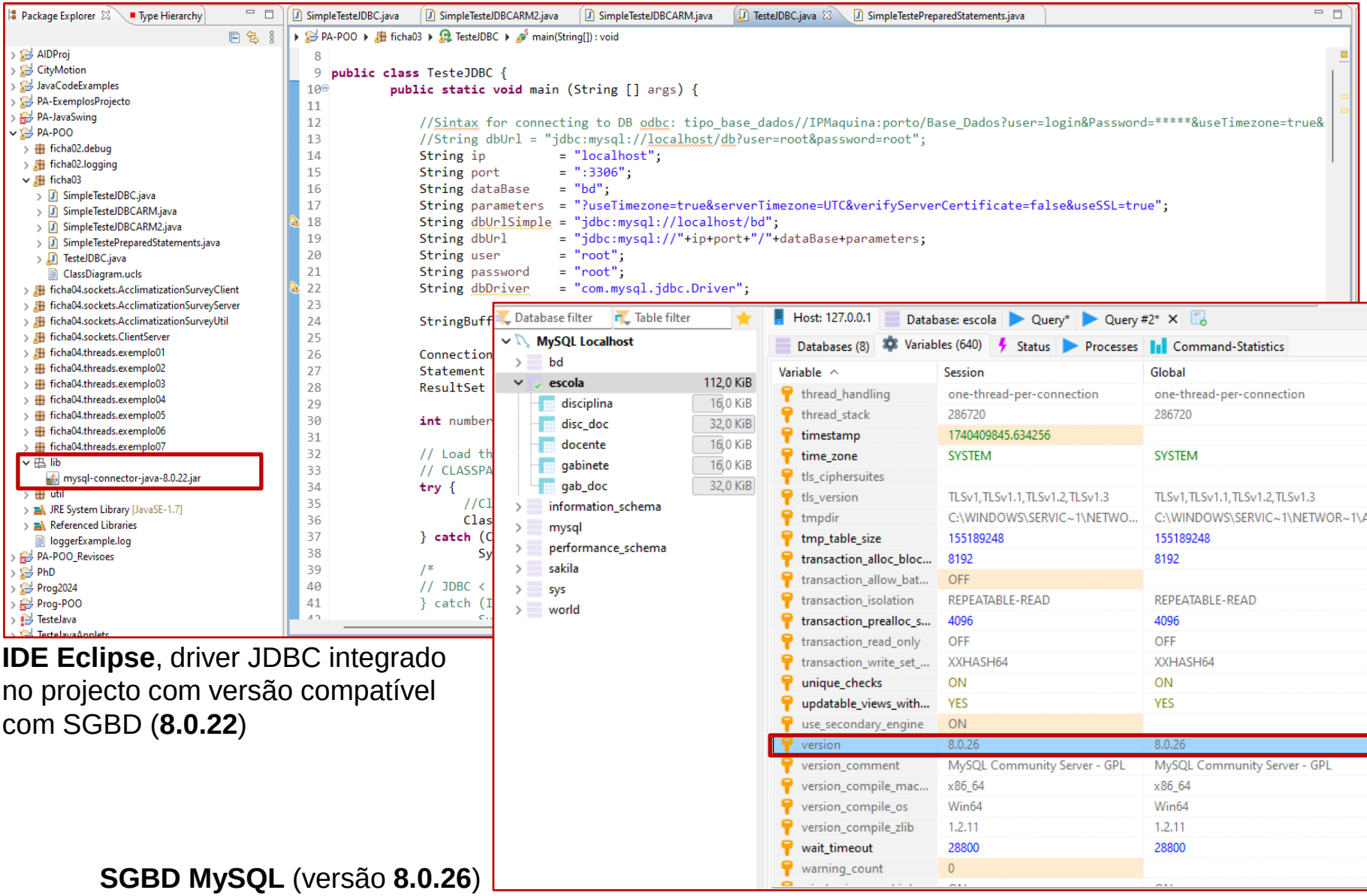
Acesso a Bases de Dados relacionais através de ODBC

O procedimento é semelhante noutros IDE, como no **IDE NetBeans**: aceder às propriedades do projecto (clicar com o botão direito sobre o projecto e seleccionar **`Properties`**) e na secção **`Libraries`**, selecionar o botão **`Add JAR Folder`** ou **`Add Library`** e seleccionar o ficheiro JAR correspondente:



Estabelecer ligação a uma base de dados relacional

Acesso a Bases de Dados relacionais através de ODBC



IDE Eclipse, driver JDBC integrado no projecto com versão compatível com SGBD (**8.0.22**)

SGBD MySQL (versão **8.0.26**)

Estabelecer ligação a uma base de dados relacional

O método **`getConnection`** recebe os parâmetros de ligação:

```
getConnection("jdbc:mysql://localhost/db");
```

Caso a seja necessário proceder a **autenticação para acesso ao servidor de base de dados** a sintaxe apresenta a seguinte estrutura, com a introdução das variáveis **`user`** e **`password`** para introdução, respectivamente, do ***login*** e ***password***

```
jdbc:mysql://localhost/db?user=xxxx&password=yyyy
```

Alternativamente poderemos recorrer a uma variação do método **`getConnection`** que recebe três argumentos: uma **localização do servidor**, o ***login*** e a ***password***, ou seja:

```
getConnection(String url, String userName, String password)
```

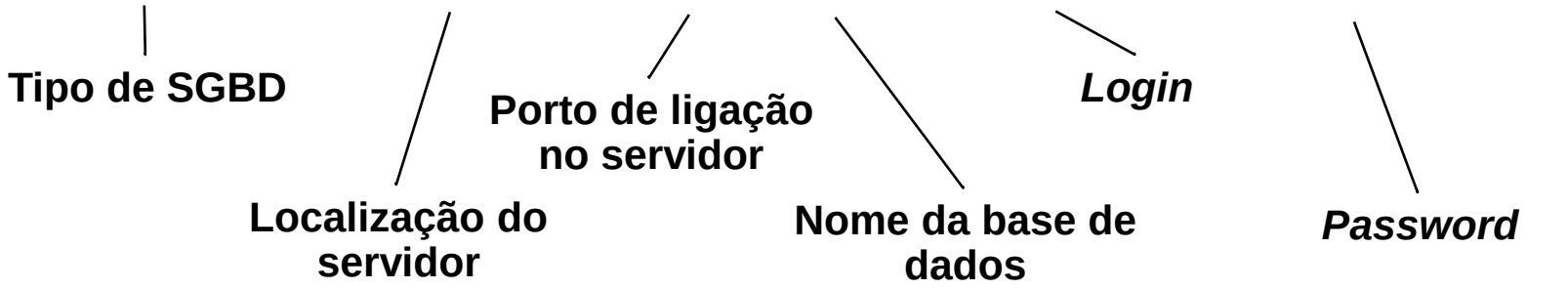
Como exemplo:

```
Connection conn = DriverManager.getConnection
                ("jdbc:mysql://localhost/db", "root", "1234");
```

Usualmente a **ligação a um servidor de base de dados é limitado a um porto**. A sintaxe anterior não especifica esse porto, sendo assim usado o **porto por defeito** (**3306**, no caso do SGBD MySQL)

Caso se pretenda **incluir a porta do servidor** MySQL (que não seja a porta por defeito 3306), basta incluir, após a designação da máquina, a referência da porta, por exemplo:

```
jdbc:mysql://localhost:3306/db?user=xxxx&password=yyyy
```



Estabelecer ligação a uma base de dados relacional (MySQL)

A **instanciação do driver ODBC** é conseguida através da instrução

```
Class.forName(String aDriver).newInstance();
```

recebendo uma *string* como parâmetro de entrada que indica a localização dos drivers específicos para a ligação à base de dados.

Por exemplo:

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

inicializa o driver JDBC **`com.mysql.jdbc.Driver`** para acesso a uma **base de dados MySQL**.

Notar que se o **driver JDBC for superior à versão 8.x ou superior**, o driver para ligação é carregado através da seguinte instrução:

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

Estabelecer ligação a uma base de dados relacional (Oracle)

Caso se pretendesse aceder a uma **base de dados Oracle** o processo descrito neste manual é idêntico bastando alterar o driver JDBC **`oracle.jdbc.driver.OracleDriver`**

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

Ou em alternativa recorrendo ao **`DriverManager`**:

```
DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
```

Por fim estabelecendo a ligação

```
String url = "jdbc:oracle:thin:@//server.local:1521/prod";
// sintaxe:   jdbc:oracle:thin:@//host:port/service
Connection conn = DriverManager.getConnection(url, "root", "1234");
```

Caso se pretendesse aceder a uma **base de dados PostgreSQL** o processo descrito neste manual é idêntico bastando alterar o driver JDBC **`org.postgresql.Driver`**

```
Class.forName("org.postgresql.Driver");
```

Por fim estabelecendo a ligação

```
String url = "jdbc:postgresql://localhost:5432/db;
// sintaxe:   jdbc:postgresql://host:port/dbname
Connection conn = DriverManager.getConnection(url, "root", "1234");
```

Caso se pretendesse aceder a uma **base de dados MS SQL Server** o processo descrito neste manual é idêntico bastando alterar o driver JDBC **com.microsoft.sqlserver.jdbc.SQLServerDriver**

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

Por fim estabelecendo a ligação

```
String url =
 "jdbc:sqlserver://localhost:1433;database=db; integratedSecurity=true;"
// sintaxe: jdbc:sqlserver://host:port;database=dbname
Connection conn = DriverManager.getConnection(url,"root", "1234");
```

Etapa 3: Após estabelecer a ligação à base de dados **inicializa-se o objecto** `Statement` aplicado à ligação:

```
Connection conn = DriverManager.getConnection
                  ("jdbc:mysql://localhost/db", "root", "root");
Statement st = conn.createStatement();
```

Etapa 4: Após a inicialização do `Statement` é possível **executar comandos SQL** através do método `executeQuery(String query)` aplicado sobre o objecto `Statement` :

```
st.executeQuery(" SELECT * FROM java_test ");
```

Existem **quatro operações DML** possíveis: **`SELECT`**, **`INSERT`**, **`UPDATE`** e **`DELETE`**

De acordo com o comando SQL, recorre-se ao método:

- **`execute()`** para **comando SQL que não devolvem resultados** (**`INSERT`** e **`DELETE`**) e os métodos, **devolvendo um *boolean*** se o primeiro resultado for um **`ResultSet`**;

  NOTA: Também é possível executar **comandos DDL**, como sejam os comandos `drop` e `create`, recorrendo ao método **`execute()`** para a respectiva execução, uma vez que estes comandos SQL não devolvem registos

- **`executeUpdate()`** para **comandos SQL que devolvem resultados na forma de um valor inteiro**, e.g. número de registos afectados (**`UPDATE`**).

  NOTA: O método **`executeUpdate()`** devolve um **valor inteiro** que corresponde ao **número de linhas actualizadas** com sucesso

- **`executeQuery()`** para **comandos SQL que devolvem resultados na forma de tabela** (**`ResultSet`**), e.g. registos (**`SELECT`**) ;

Da execução de comandos SQL (*e.g.* **`SELECT`**) podem ser devolvidos vários registos. Na prática um **SGBD responde sempre com uma tabela**, **que será transformada pelo JDBC no formato de um objecto do tipo `ResultSet`** (**Etapa 5**):

```
ResultSet rs = st.executeQuery(" SELECT * FROM java_test ");
```

É assim necessário **iterar pelos respectivos elementos** (registos)

A instrução **`rs.next()`** retorna ***true*** se **existirem elementos** no **`ResultSet`**, ou ***false*** caso **não existam elementos** disponíveis no **`ResultSet`**

O objecto **`ResultSet`** apenas será **nulo** (**`rs == null`**) **se a pesquisa não poder ser executada** (e.g. a tabela sobre a qual incide a pesquisa não existir), no entanto tal situação também invocará uma **`SQLException`**

Caso a pesquisa não devolva registos, **o SGBD retorna sempre uma tabela vazia**. Desta forma, caso o comando SQL seja válido, o JDBC retorna sempre um objecto **`ResultSet`** não nulo

Etapa 5: Após a obtenção dos dados, armazenados no objecto **`ResultSet`**, podemos proceder à obtenção dos diversos registos, **iterando pela estrutura através da condição `rs.next()`**:

```
while (rs.next()) {
        int c1    = rs.getInt("column1");
        String c2 = rs.getString("column2");
        System.out.println("Column1 = "+ c1 + "; Column2 = "+c2);
}
```

Para se **aceder a um campo de um registo** podemos usar

- o **respectivo nome do atributo**, como em **`rs.getInt("column1")`**,
- ou podemos invocar a sua **posição na tabela**, no exemplo anterior seria **`rs.getInt(1)`**

Para obter o respectivo valor de cada campo é **necessário invocar um método `get` desse tipo**, ou seja:

- caso um campo seja do tipo ***int*** necessita de recorrer ao método **`getInt()`**;
- caso o campo seja do tipo ***string*** necessita de usar o método **`getString()`**;
- caso fosse uma data (***date***) invocaríamos o método **`getDate()`**;

e assim sucessivamente

O método `next()` permite-nos **obter a próxima linha lógica** na *query* de resposta.

No entanto, também nos podemos movimentar em outros sentidos na estrutura da *query* de resposta, nomeadamente:

`previous()`	move para trás uma linha;
`absolute(int num)`	move para a linha com o número especificado;
`relative(int num)`	avança para a frente ou para trás (caso o num seja negativo) relativamente à posição actual. `relative(-1)` tem o mesmo efeito de `previous()`;
`first()`	move para a primeira linha;
`last()`	move para a última linha;

A **execução de comandos SQL** (e.g. `st.executeQuery()`) pode lançar uma **excepção** do tipo `SQLException`

Enquanto o processo de **estabelecimento da ligação** (instrução `Class.forName(dbDriver).newInstance()`) pode lançar 3 tipos de excepções:

- `ClassNotFoundException`,
- `IllegalAccessException` e `// JDBC < 8.x (.newInstance())`
- `InstantiationException` `// JDBC < 8.x (.newInstance())`

que devem ser tratadas dentro de um bloco `try (...) catch`

De notar que as excepções `IllegalAccessException` e `InstantiationException` só se aplicam quando usamos o **driver JDBC inferior à versão 8.x**, ou **JSE igual ou inferior à versão 15**, uma vez que acima desta versão não é invocada a instrução `.newInstance()`

Etapa 6: No final da execução da aplicação é necessário

- **encerrar o *statement*** (`st.close()`)
- e **por fim a ligação à base de dados** (`conn.close()`), libertando os respectivos recursos

Estas instruções podem ser inseridas numa **cláusula *finally*** para que independentemente da execução correcta da aplicação, ou da existência de excepções, a ligação seja sempre terminada

Esta finalização pode, por sua, vez lançar excepções na sua execução, pelo que também deverá ser tratada por um bloco **`try`** (…) **`catch`**

Tratamento de excepções e encerramento de recursos

Acesso a Bases de Dados relacionais através de ODBC

```
try {

      (...) // código de acesso à base de dados

} catch (SQLException e) {
      System.err.println("!! SQL Exception !!\n"+e);
      e.printStackTrace();
} catch (ClassNotFoundException e) {
      System.err.println("!! Unable to load Database Drive !!\n"+e);
} catch (IllegalAccessException e) {   // JDBC < 8.x (.newInstance())
      System.err.println("!! Illegal Access !!\n"+e);
} catch (InstantiationException e) {   // JDBC < 8.x (.newInstance())
      System.err.println("!! Class Not Instantiated !!\n"+e);

} finally {
      if (st != null) {    // encerra statement
         try { st.close(); } catch (Exception e) {
             System.err.println("!! Exception returning statement !!\n"+e);
         }
      }
      if (conn != null) { // encerra ligação ao SGBD
          try { conn.close(); } catch (Exception e) {
             System.err.println("!! Exception closing DB connection !!\n"+e);
          }
      }
} // end of finally
```

Exemplo completo

Acesso a Bases de Dados relacionais através de ODBC

```
Connection conn = null;
Statement st = null;
ResultSet rs = null;

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

    conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/bd", "root", "root");
    st = conn.createStatement();

    rs = st.executeQuery(" SELECT * FROM java_test ");

    if (rs == null || !rs.next()) {
       System.out.println("!! No Record on table !!");
    } else
       while (rs.next()) {
          int    column1 = rs.getInt("column1");
          String column2 = rs.getString("column2");
          System.out.println("Column1 = "+ column1 + " ; Column2 = "+column2);
       }
} catch (SQLException e) {
    System.out.println("!! SQL Exception !!\n"+e);
    e.printStackTrace();
} catch (ClassNotFoundException e) {
       System.out.println("!! Class Not Found. Unable to load Database Drive !!\n"+e);
} catch (IllegalAccessException e) {
       System.out.println("!! Illegal Access !!\n"+e);
} catch (InstantiationException e) {
       System.out.println("!! Class Not Instanciaded !!\n"+e);
} finally {
    if (st != null) {
       try {
          st.close();
       } catch (Exception e) {
          System.out.println("!! Exception returning statement !!\n"+e);
       }
    }
    if (conn != null) {
       try {
          conn.close();
       } catch (Exception e) {
          System.out.println("!! Exception closing DB connection !!\n"+e);
       }
    }
} // end of finally
```

Etapa 1: Carregar driver

Etapa 2: Estabelecer ligação

Etapa 3: Inicializar o objecto `Statement`

Etapa 4: Execução da *query* SQL

Etapa 5: Iteração sobre os registos da tabela (`ResultSet`)

Tratamento de excepções

Etapa 6: Encerrar e libertar os recursos (`Statement` e `Connection`)

Tratamento de excepções e encerramento de recursos

Tal como na manipulação de ficheiros, caso se pretenda **abrir, usar e fechar os recursos de acesso à base de dados na mesma chamada**, pode-se recorrer ao ***Automatic Resource Management*** **(ARM)** usando o ***try-with-resources***, evitando a necessidade da clausula **`finally`** para encerrar os recursos, uma vez que implementa **`java.lang.AutoCloseable`** (**`java.io.Closeable`**).

```
import java.sql.*;
//(...)
try ( Connection conn = DriverManager.getConnection( connectionParameters ) ;
      Statement st = conn.createStatement() )
{
      ResultSet rs = st.executeQuery(" SELECT * FROM java_test");
      while (rs.next()) {
            int    column1 = rs.getInt("column1");
            String column2 = rs.getString("column2");
            System.out.println("Column1 = "+ column1 + " ; Column2 = "+column2);
      }
}
// recursos Connection e Statement encerrados automaticamente
catch (SQLException sqle)
{
      System.err.println("Statement Exception"+sqle);
      sqle.printStackTrace();
}
```

Exemplo completo (*Automatic Resource Management*)

Acesso a Bases de Dados relacionais através de ODBC

```
String connectionParameters = "jdbc:mysql://localhost:3306/bd?user=root&password=root";

try(  Connection conn = DriverManager.getConnection(connectionParameters) ;
      Statement st = conn.createStatement() )
{

      ResultSet rs = st.executeQuery(" SELECT * FROM java_test");

      while ( rs.next() ) {
              int column1    = rs.getInt("column1");
              String column2  = rs.getString("column2");
              System.out.println("Column1 = "+ column1 + " ; Column2 = "+column2);
      }

      // recursos Connection e Statement encerrados automaticamente
} catch (SQLException sqle) {
      System.err.println("Statement Exception"+sqle);
      sqle.printStackTrace();
}
```

Exemplo completo (*Automatic Resource Management*)

Acesso a Bases de Dados relacionais através de ODBC

```
String connectionParameters = "jdbc:mysql://localhost:3306/bd?user=root&password=root";

try {
   Class.forName("com.mysql.cj.jdbc.Driver");
} catch (ClassNotFoundException cnfe) {
   System.err.println("Class Not Found. Unable to load Database Drive"+cnfe);
}

try(Connection conn = DriverManager.getConnection(connectionParameters)  ) {
     conn.setAutoCommit(false);

     try( Statement st = conn.createStatement() ) {

               ResultSet rs = st.executeQuery(" SELECT * FROM java_test");

               while ( rs.next() ) {
                        int column1 = rs.getInt("column1");
                        String column2 = rs.getString("column2");
                        System.out.println("Column1 = "+ column1 + " ; Column2 = "+column2);
               }
               conn.commit();

     // recurso Statement encerrado automaticamente
     } catch (SQLException sqle) {
               System.err.println("Statement Exception"+sqle);
               sqle.printStackTrace();
               conn.rollback();
     }

// recurso Connection encerrado automaticamente
} catch (SQLException sqle) {
     System.err.println("Connection Exception"+sqle);
     sqle.printStackTrace();
}
```

Uso de *Statement* em múltiplos acessos

A execução de um comando SQL através de um **`Statement`** **automaticamente elimina os recursos da execução anterior**

Assim, se um **`Statement`** devolve vários registos que têm que ser iterados por um **`ResultSet`**, se **durante o ciclo** o **`Statament`** é novamente usado para outro acesso ao SGBD, **o `ResultSet` é automaticamente encerrado e qualquer acesso ao `ResulSet` gera uma `SQLException`** com a descrição "*`Operation not allowed after ResultSet closed`*"

Desta forma será sempre necessário **criar um novo `Statement`** (a partir da **`Connection`** original)

Uso de *Statement* em múltiplos acessos - erro

Acesso a Bases de Dados relacionais através de ODBC

```
try {
	Class.forName("com.mysql.jdbc.Driver").newInstance();
	Connection conn =
		DriverManager.getConnection("jdbc:mysql://localhost/bd", "root", "root");
	Statement st = conn.createStatement();
	ResultSet rs1 = st.executeQuery(" SELECT * FROM emp ");

	while (rs1.next()) {
		int column1 = rs1.getInt("nemp");
		System.out.println("Column1 = "+ column1);

		// este acesso encerra o ResultSet rs1 e todos os recursos associados
		ResultSet rs2 = st.executeQuery(" SELECT * FROM dep ");
		while (rs2.next()) {
			String column2 = rs2.getString("nome");
			System.out.println("Column2 = "+ column2);
		}

		// o acesso ao ResultSet rs1 gera uma excepção, uma vez que rs1 está encerrado
		Date data = rs1.getTimestamp("Data_Entrada");
		System.out.println("Date = " +data);
	}
} catch (SQLException e) {
	System.out.println("!! SQL Exception !!\n"+e);
	e.printStackTrace();
}
```

Uso de *Statement* em múltiplos acessos - sem erro

Acesso a Bases de Dados relacionais através de ODBC

```
try {
	Class.forName("com.mysql.jdbc.Driver").newInstance();
	Connection conn =
		DriverManager.getConnection("jdbc:mysql://localhost/bd", "root", "1234");
	Statement st1 = conn.createStatement();
	ResultSet rs1 = st1.executeQuery(" SELECT * FROM emp ");

	while (rs1.next()) {
		int column1 = rs1.getInt("nemp");
		System.out.println("Column1 = "+ column1);

		// necessário criar um novo Statement para não encerrar o ResultSet rs1
		Statement st2 = conn.createStatement();
		ResultSet rs2 = st2.executeQuery(" SELECT * FROM dep ");
		while (rs2.next()) {
			String column2 = rs2.getString("nome");
			System.out.println("Column2 = "+ column2);
		}

		// o ResultSet rs1 continua activo e já não gera uma excepção
		Date data = rs1.getTimestamp("Data_Entrada");
		System.out.println("Date = " +data);
	}
} catch (SQLException e) {
	System.out.println("!! SQL Exception !!\n"+e);
	e.printStackTrace();
}
```

Após estabelecer a ligação ao SGBD (**`Connection`**) **é possível criar um `Statement`**, através do método **`createStatment()`**, para o envio de comandos SQL

Adicionalmente, é possível **definir vários parâmetros da ligação sobre o objecto `Connection`**. Nomeadamente, o **controlo de transacções** é relevante num sistema concorrente. Assim, é possível definir se todas as operações SQL vão automaticamente confirmar a transacção ou a criação de *SavePoints*

A propriedade **`AutoCommit`** é controlada pelo método

```
setAutoCommit(boolean);      // default: true;
```

sobre o objecto **`Connection`** e define se após cada comando o SGBD deve **realizar um *commit*** automaticamente.

Para realizar um *commit* manual recorre-se ao método

```
commit();
```

A **criação de um `SavePoint`** é realizada através do método

```
setSavePoint(boolean);
```

É possível desfazer as alterações de uma transacção até ao último *commit* através do método **`rollback()`**, ou até ao último *SavePoint* através do método

```
rollback(Savepoint);
```

A criação de uma **transacção só de leitura** é obtida através do método

```
setReadOnly(boolean);
```

Controlo de transacções

Acesso a Bases de Dados relacionais através de ODBC

```
//STEP 1. Import required packages
import java.sql.*;
Connection conn = null;
Statement stmt  = null;
String DB_URL   = "jdbc:mysql://localhost/db";
String USER     = "username";
String PASS     = "password";
try{
   //STEP 2: Register JDBC driver
   Class.forName("com.mysql.jdbc.Driver");

   //STEP 3: Open a connection
   conn = DriverManager.getConnection(DB_URL, USER, PASS);

   //STEP 4: Set auto commit as false.
   conn.setAutoCommit(false);

   //STEP 5: Execute a query to create statment with
   // required arguments for RS example.
   stmt = conn.createStatement(
                        ResultSet.TYPE_SCROLL_INSENSITIVE,
                        ResultSet.CONCUR_UPDATABLE);

   //STEP 6: INSERT a row into Employees table
   String SQL = "INSERT INTO emp " +
                "VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);

   //STEP 7: INSERT one more row into Employees table
   SQL = "INSERT INTO emp " +
                "VALUES (107, 22, 'Sita', 'Singh')";
   stmt.executeUpdate(SQL);

   //STEP 8: Commit data here.
   conn.commit();

   //STEP 9: Now list all the available records.
   String sql = "SELECT nemp, name FROM emp";
   ResultSet rs = stmt.executeQuery(sql);
   //Ensure we start with first row
   rs.beforeFirst();
   while(rs.next()){
      //Retrieve by column name
      int id       = rs.getInt("nemp");
      String name  = rs.getString("name");

      //Display values
      System.out.print("ID: " + id + "Name: " + name);
   }
   //STEP 10: Clean-up environment
   rs.close();
   stmt.close();
   conn.close();

}catch(SQLException se){
   //Handle errors for JDBC
   se.printStackTrace();
   // If there is an error then rollback the changes.
   try{
      if(conn!=null)
         conn.rollback();
   }catch(SQLException se2){
      se2.printStackTrace();
   }//end try

}catch(Exception e){
   //Handle errors for Class.forName
   e.printStackTrace();
}finally{
   //finally block used to close resources
   try{
      if(stmt!=null)
         stmt.close();
   }catch(SQLException se){
      // nothing we can do
   }
   try{
      if(conn!=null)
         conn.close();
   }catch(SQLException se){
      se.printStackTrace();
   }//end finally try
}//end try
```

Uma data pode ser manipulada e armazenada como uma variável *String* (*"2019-03-19"*) ou um valor inteiro (*20190319*). Porém, este procedimento impede o uso dos vários métodos úteis de manipulação e formatação de datas, disponibilizados tanto pela API Java como SQL.

Assim, **é aconselhável que numa base de dados as datas sejam armazenadas como tipo `Date`** (se só contiverem os parâmetros ano, mês e dia) ou, se pretendermos um instante no tempo incluindo a hora, minuto e segundo, **como tipo `DateTime` ou `Timestamp`**

Em **Java**, **uma data é representada pela classe `java.util.Date`**.

Um **SGBD utiliza o seu próprio formato de data,** que em Java **é representado pela classe `java.sql.Date`**.

Quando **se lê** ou **se escreve uma data** (métodos **java.sql.ResultSet.getDate()** e **java.sql.ResultSet.setDate()**) numa base de dados é necessário **realizar uma conversão entre os dois formatos** (**java.util.Date** e **java.sql.Date**), **através do método getTime()**

Por exemplo, **converter um java.util.Date em java.sql.Date para escrita** no SGBD, através do método **getTime()** (da classe **java.util.Date**):

```
java.util.Date utilDate = new java.util.Date();
java.sql.Date  sqlDate  = new java.sql.Date( utilDate.getTime() );
java.sql.ResultSet.setDate(1, sqlDate);
```

Em sentido inverso, **ler uma data do SGBD (java.sql.Date) e converter em java.util.Date**, também se recorre ao método **getTime()** (neste caso, da classe **java.sql.Date**):

```
java.util.Date utilDate =
        new Date( java.sql.ResultSet.getDate("atributo").getTime() );
```

Em termos genéricos, quando se lê um **campo `Date`** de um SGBD, é possível **armazena-lo directamente numa variável do tipo `java.util.Date`** como numa **variável do tipo `java.util.Date`**

```
java.util.Date data = java.sql.ResultSet.getDate("Data_Entrada");
```

ou

```
java.sql.Date  data = java.sql.ResultSet.getDate("Data_Entrada");
```

Adicionalmente, ao obter-se um **campo `Date`** de um SGBD, podemos imediatamente **converte-lo numa `String`** e armazena-lo numa variável **`String`**, embora desta forma deixa de ser possível recorrer ao vários métodos de formatação e manipulação de datas disponíveis na classe **`Date`**

```
String data = java.sql.ResultSet.getDate("Data_Entrada").toString();
```

Ou de forma simplificada, através de uma **concatenação com uma *string* vazia**:

```
String data = java.sql.ResultSet.getDate("Data_Entrada")+"";
```

Porém, o procedimento anterior (**getDate()**) **apenas utiliza os campos ano, mês e dia**, não manipulando hora, minuto e segundo.

Para **manipular simultaneamente campos de data e hora**, está disponível a classe **java.sql.Timestamp**.

```
java.util.Date utilDate =
                    java.sql.ResultSet.getTimestamp(“atributo”);
```

Recorrendo ao método **getTimestamp()** ao invés do método **getDate()**, **obtemos todos os parâmetros de uma *timestamp* (Data + Hora)**. Caso o campo **Date** da SGBD não possua informação sobre hora, minuto e segundo, estes parâmetros são colocados a zero:

```
java.util.Date data =
                    java.sql.ResultSet.getTimestamp("Data_Entrada");
System.out.println(data);
```

Resultado: *1996-02-07 00:00:00.0*

Manipulação de datas: Fuso horário

Devido a conflitos de fusos horários entre a máquina cliente e o servidor SGBD é possível surgirem conflitos, sendo lançada a seguinte excepção:

"*java.sql.SQLException: The server time zone value is unrecognized or represents more than one time zone. **You must configure either the server or JDBC driver (via the 'serverTimezone' configuration property) to use a more specific time zone value** if you want to utilize time zone support*"

Para corrigir esta ambiguidade e definir um fuso horário especifico ou sincronizar o fuso horário entre cliente e servidor deve-se definir as propriedades **useTimezone** e **ServerTimezone**

```
jdbc:mysql://localhost:3306/bd?useTimezone=true&serverTimezone=UTC
```

Desta forma estamos a indicar que será seguido o fuso horário do cliente, tendo por base o **UTC** (***Coordinated Universal Time***, anteriormente conhecido por GMT ***Greenwich Mean Time***)

Lisboa segue o UTC no Inverno e o UTC+1 no Verão

Ao usar uma versão MySQL superior a 5.5, é possível obter o seguinte aviso, ao estabelecer a ligação ao SGBD:

"*WARN: Establishing SSL connection without server's identity verification is not recommended (…) requirements SSL connection must be established by default (…) **You need either to explicitly disable SSL by setting useSSL=false, or set useSSL=true and provide truststore for server certificate verification**.*"

Como o aviso refere, o SGBD espera o uso de uma **ligação segura (SSL *Secure Socket Layer*)**, validada por certificado, não estando a aplicação a disponibilizar um certificado válido

Caso não se pretenda utilizar uma ligação SSL, pode-se **instruir o servidor para ignorar essa opção**, **alterando o valor do parâmetro `useSSL` para `false`** no momento de estabelecimento da ligação:

```
Connection conn = DriverManager.getConnection
    ("jdbc:mysql://localhost/db?useSSL=false", "root", "1234");
```

De recordar que ao invocar o método **`getConnection()`**, após o URL de localização do servidor SGBD, caso se pretenda incluir parâmetros, deve-se introduzir o caracter ***?***

Caso existam vários parâmetros recorre-se ao caracter ***&*** para a sua separação e identificação:

```
jdbc:mysql://localhost:3306/bd?user=root&password=1234&useSSL=false
```

Se pretender usar uma **ligação com SSL, mas não possuir certificado**, é possível activar a ligação SSL (**`useSSL=true`**) e desactivar a verificação de certificados (**`verifyServerCertificate=false`**)

```
jdbc:mysql://localhost:3306/bd?verifyServerCertificate=false&useSSL=true
```

Parâmetros para configuração do acesso à base de dados

Parâmetros comuns para configuração da ligação à base de dados:

```
getConnection("jdbc:mysql://localhost:3386/db_name?var1=valor1&var2=valor2");
    user=root
    password=PA239$fg%
    useSSL=false
    verifyServerCertificate=false
    allowPublicKeyRetrieval=true
    useTimezone=true
    serverTimezone=UTC
    autoReconnect=true
```

O processo também pode ser realizado recorrendo a **`properties`**:

```
Properties properties = new Properties();

properties.setProperty("user",           "root");
properties.setProperty("password",       "1234");
properties.setProperty("useSSL",         "false");
properties.setProperty("autoReconnect", "true");

String connURL = "jdbc:mysql://localhost:3306/bd";

try {
        Connection conn = DriverManager.getConnection(connURL, properties);
        //...
} catch (SQLException e) {
        //...
}
```

Alternativamente, os parâmetros de ligação podem ser armazenados num ficheiro e acedidos através de **`properties`**:

```
Properties prop = new Properties();

try {
     prop.load( new FileInputStream("config_database.ini") );

     String dbName  = prop.getProperty("dataBase") );
     String dbUser  = prop.getProperty("user") );
     String dbPass  = prop.getProperty("password") );

     Connection conn = DriverManager.getConnection(
                  "jdbc:mysql://localhost/"+dbName+"?
                                   user="+dbUser+"&password"+dbPass);
} catch (IOException ioe) {
     ioe.printStackTrace();
} catch (SQLException sqle) {
     sqle.printStackTrace();
}
```

config_database.ini

```
1  ip                       = 192.168.5.25
2  port                     = 3306
3  dataBase                 = db_project1
4  user                     = root
5  password                 = PR0Jmain!25$
6  dbDriver                 = com.mysql.jdbc.Driver
7  useTimezone              = true
8  serverTimezone           = UTC
9  useSSL                   = true
10 verifyServerCertificate  = false
11 autoReconnect            = true
12 autoCommit               = false
```

Prepared Statements

SQL Injection

Acesso a Bases de Dados relacionais através de ODBC

Uma alternativa à classe `Statement`, é a classe `PreparedStatement` como um procedimento para prevenir situações de *SQL Injection*

Esta nova classe (tendo como base a classe `Statement`) permite-nos **gerar dinamicamente declarações SQL pré-compiladas** que podem ser **utilizadas inúmeras vezes**

Estas **declarações podem apresentar argumentos**, no entanto a sua estrutura é fixada quanto o objecto do tipo `PreparedStatement` (representando a declaração SQL) é criado

Relativamente ao método anterior, **todas as etapas – exceptuando a 3 – seguem o mesmo procedimento**

Assim, o processo para **carregar o driver** e **estabelecer ligação ao SGBD** (**Etapas 1** e **2**) segue os mesmos passos do método anterior (tenha em atenção se está a usar um JDBC superior a 8.x ou JSE superior a 15):

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
Connection conn =
        DriverManager.getConnection("jdbc:mysql://localhost/db");
```

Porém, na **Etapa 3**, ao invés de se criar um `Statement`, cria-se um `PreparedStatement`. Este objecto vai **validar a estrutura do comando SQL**:

```
PreparedStatement ps = conn.prepareStatement(
                    "SELECT Column1 from java_test WHERE id = ?");
```

De notar que o **comando SQL não possuir variáveis**. Estas são substituídas pelo caracter '**?**', uma indicação ao SGBD que os valores serão integrados posteriormente

O objecto **`PreparedStatement`** (que substitui o **`Statement`**) permite **validar a estrutura (ou sintaxe) do comando SQL antes da sua execução**

Os **valores das variáveis serão integrados posteriormente**, e nesses valores **não será permitida a inclusão de comandos SQL**, impedindo desta forma *SQL Injection*

Para limpar valores anteriores das variáveis recorre-se à instrução

```
ps.clearParameters();
```

Para introduzir os valores das variáveis recorre-se aos métodos **`setInt()`**, **`setFloat()`**, **`setLong()`**, **`setDouble()`**, **`setString()`**, **`setDate()`**, **`setCharacter()`**, *etc.*, consoante o tipo de valor que se pretenda introduzir

Estes métodos recebem dois argumentos: a posição da variável que vai receber o valor e o próprio valor:

```
ps.setInt(1, 1000);
```

Introduz na **primeira variável** (**1**) o **valor inteiro 1000**

Prepared Statements

Acesso a Bases de Dados relacionais através de ODBC

```
//(…)
Class.forName("com.mysql.jdbc.Driver").newInstance();                    //1
Connection conn =
           DriverManager.getConnection("jdbc:mysql://localhost/db");//2
//(…)
StringBuffer sqlQuery = new StringBuffer();
sqlQuery.append(" INSERT INTO java_test (column1, column2) ");
sqlQuery.append(" VALUES (?, ?) ");
try {
     PreparedStatement ps =
                        conn.prepareStatement(sqlQuery.toString()); //3a

     ps.clearParameters();
     ps.setInt(1, 1000);                                                 //3b
     ps.setString(2, "Insert with prepared statement");                  //3b

     int count = ps.executeUpdate();                                     //4
     //(…)
} catch (SQLException e) { /* (…) */ }
//(…)
     ps.close();                                                         //6
//(…)
```

No exemplo anterior, a *query* SQL **especifica a existência de um campo do tipo *String* e do tipo *int***, no entanto **usa ao caracter '`?`' para os valores dos parâmetros**, que são **definidos posteriormente pelos métodos** `setString()` e `setInt()`

Os **caracteres '`?`' podem ser utilizados em qualquer posição na declaração SQL onde possam ser substituídos por um valor**.

Por exemplo, podem ser utilizados na cláusula `WHERE` (*e.g.* '`WHERE column2 = ?`'), ou nas declarações SQL de inserção (`INSERT`) e actualização (`UPDATE`)

O método `setString()` é uma forma de definir o valor do parâmetro, existindo métodos análogos para dados do tipo *int*, *float* ou *date*.

É uma boa regra de programação começar esta acção invocando o método `clearParameters()`, por forma a **remover os dados anteriores** que possam existir nos parâmetros

Prepared Statements com variáveis

Ao atribuir valores a um **`PreparedStatement`** podemos indicar um **valor fixo** (e.g. **`1000`** ou **“`Hello World!`”**) ou uma **variável**:

```
String pesquisa = "i";     // pesquisar todos os nomes com caracter ‘i’
StringBuffer sqlQuery = new StringBuffer();
sqlQuery.append(" SELECT * FROM java_test ");
sqlQuery.append(" WHERE column2 LIKE ? ");     // objectivo: pesquisar ‘%i%’

PreparedStatement ps = conn.prepareStatement(sqlQuery.toString());
ps.clearParameters();
ps.setString(1,”%”+pesquisa+”%”); // equivalente a: ps.setString(1,"%i%");
// de notar que a pesquisa ps.setString(1, pesquisa); devolve 0 resultados
// em alternativa poder-se-ia definir a variável pesquisa = “%i%” e executar o
// o set parameter sem concatenação: ps.setString(1, pesquisa);

ResultSet rs = ps.executeQuery();

if (rs == null) {
        System.out.println("!! No Record on table !!");
} else
        while (rs.next()) {
                int column1 = rs.getInt(1);
                System.out.println("Column1 = "+ column1);
        }
```

De realçar que em SQL a pesquisa com **a cláusula `LIKE` compara um valor entre plicas** (‘), não sendo necessário introduzir quando se recorre a **`Prepared Statements`**

Exemplo completo

Acesso a Bases de Dados relacionais através de ODBC

```
try {
    // load driver; establish connection
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/bd", "root", "root");

    // prepared statement for updating
    StringBuffer sqlQuery = new StringBuffer();
    sqlQuery.append(" UPDATE java_test ");
    sqlQuery.append(" SET column3 = ? ");
    sqlQuery.append(" WHERE column1 = ? ");

    PreparedStatement ps = conn.prepareStatement(sqlQuery.toString());
    ps.clearParameters();
    ps.setBoolean(1,  true);
    ps.setInt(2, 1);

    int count = ps.executeUpdate();
    System.out.println("Rows updated = "+count);
    ps.close();

    // prepared statement for select
    String pesquisa = "i";
    sqlQuery = new StringBuffer();
    sqlQuery.append(" SELECT * FROM java_test ");
    sqlQuery.append(" WHERE column2 like ? ");

    ps = conn.prepareStatement(sqlQuery.toString());
    ps.clearParameters();
    ps.setString(1,'%'+pesquisa+'%');

    rs = ps.executeQuery();

    if (rs == null) {
          System.out.println("!! No Record on table !!");
    } else
        while (rs.next()) {
            int   column1 = rs.getInt(3);
            System.out.println("C Column3 = "+ column1);
        }
    ps.close();
} catch (SQLException e) {
    System.out.println("!! SQL Exception !!\n"+e);
    e.printStackTrace();
}
```

Etapa 1: Carregar driver

Etapa 2: Estabelecer ligação

Etapa 3a: Inicializar o objecto `PreparedStatemente` validar a sintaxe da *query* SQL

Etapa 3b: Introduzir os valores das variáveis

Etapa 4: Execução da *query* SQL

Etapa 5: Iteração sobre os registos da tabela (`ResultSet`)

Etapa 6: Encerrar e libertar os recursos (`PreparedStatement`)

Tratamento de excepções

Tratamento de excepções em *prepared statements*

Acesso a Bases de Dados relacionais através de ODBC

```
try {

      (...) // código de acesso à base de dados

} catch (SQLException e) {
  System.err.println("!! SQL Exception !!\n"+e);
  e.printStackTrace();
} catch (ClassNotFoundException e) {
   System.err.println("!! Unable to load Database Drive !!\n"+e);
} catch (IllegalAccessException e) {         // JDBC < 8.x (.newInstance())
   System.err.println("!! Illegal Access !!\n"+e);
} catch (InstantiationException e) {         // JDBC < 8.x (.newInstance())
   System.err.println("!! Class Not Instanciaded !!\n"+e);

} finally {
   if (st != null) {          // statement
      try { st.close(); } catch (Exception e) {
         System.err.println("!! Exception returning statement !!\n"+e);
      }
   }
   if (ps != null) {          // prepared statement
      try { ps.close(); } catch (Exception e) {
         System.err.println("!! Exception returning statement !!\n"+e);
      }
   }
   if (conn != null) {        // connection
     try { conn.close(); } catch (Exception e) {
         System.err.println("!! Exception closing DB connection !!\n"+e);
      }
   }
} // end of finally
```

Exemplo completo (*Automatic Resource Management*)

Acesso a Bases de Dados relacionais através de ODBC

```
String connectionParameters = "jdbc:mysql://localhost:3306/bd?useTimezone=true";

try ( Connection conn = DriverManager.getConnection(connectionParameters, "root", "root") ;
      PreparedStatement ps = conn.prepareStatement(    " UPDATE java_test " +
                                                       " SET column3 = ?  " +
                                                       " WHERE column1 = ? ")
 ){

    ps.clearParameters();
    ps.setBoolean(1,  true);
    ps.setInt(2, 1);

    int count = ps.executeUpdate();
    System.out.println("Rows updated = "+count);

// recursos Connection e PreparedStatement encerrados automaticamente
} catch (SQLException e) {
    System.err.println("SQL Exception\n"+e);
    e.printStackTrace();
}
```

Exemplo completo (*Automatic Resource Management*)

Acesso a Bases de Dados relacionais através de ODBC

```
String connectionParameters = "jdbc:mysql://localhost:3306/bd?useTimezone=true";

try (Connection conn = DriverManager.getConnection(connectionParameters, "root", "root") ){

  StringBuffer sqlQuery = new StringBuffer();
  sqlQuery.append(" UPDATE java_test ");
  sqlQuery.append(" SET column3 = ? ");
  sqlQuery.append(" WHERE column1 = ? ");

  try (PreparedStatement ps = conn.prepareStatement( sqlQuery.toString() ) ) {

            ps.clearParameters();
            ps.setBoolean(1,  true);
            ps.setInt(2, 1);

            int count = ps.executeUpdate();
            System.out.println("Rows updated = "+count);

  // recurso PreparedStatement encerrado automaticamente
  } catch (SQLException sqlep) {
            System.err.println("!! PreparedStatement Exception !!\n"+ sqlep);
            sqlep.printStackTrace();
  }

// recurso Connection encerrado automaticamente
} catch (SQLException sqlec) {
    System.err.println("Connection Exception\n"+ sqlec);
    sqlec.printStackTrace();
}
```

É possível **obter informação sobre o sistema, bem como o catálogo da base de dados**. Para o efeito recorremos ao objecto **`DatabaseMetaData`**

Por exemplo, o seguinte fragmento demonstra como podemos obter o nome, e a versão do *driver* JDBC

```
DatabaseMetaData dmd = conn.getMetaData();
    System.out.println("System information:");
    System.out.println(" Name: " +      dmd.getDriverName()
                       +" Version: " +  dmd.getDriverVersion());
```

O objecto `DatabaseMetaData` disponibiliza um diversificado número de métodos, sendo aqui apresentados alguns desses métodos mais relevantes:

- **`public ResultSet getCatalog() throws SQLException`**. Esta função devolve um `ResultSet` que pode ser usado para iteração sobre todas as relações do catálogo. As funções **`getIndexOf()`** e **`getTables()`** funcionam de forma análoga;
- **`public int getMaxConnections() throws SQLException`**. Esta função devolve o número máximo de ligações possíveis.

SQLJ

SQLJ (diminutivo para '***SQL-Java***') foi desenvolvido pelo *SQL Group* para **complementar a forma dinâmica de criação de *querys* em JDBC com um modelo estático**, muito próximo do SQL "embebido"

Ao contrário do JDBC, a existência de *querys* SQL semi-estáticas permite ao compilador realizar **verificações de sintaxe, verificação do tipo de dados entre atributos e valores, e a consistência entre a *query* e o esquema de bases de dados no momento de compilação**

Por exemplo, tanto em SQLJ como no SQL "embebido", as variáveis na linguagem de programação são associadas aos tipos de forma estática, enquanto no JDBC necessitamos de separar as declarações para atribuir cada variável ao argumento e obter o resultado

A declaração SQLJ seguinte atribui o tipo de variáveis da linguagem de programação `title`, `price` e `author` aos valores de retorno do cursor `books`:

```
#sql books = {
   SELECT title, price INTO :title, :price
   FROM books WHERE author = :author
};
```

Em JDBC, podemos **decidir dinamicamente quais as variáveis da linguagem de programação que vão suportar o resultado da *query***

No exemplo seguinte, obtemos o valor do título do livro para a variável `ftitle` caso o livro tenha sito escrito por um autor denominado "*Feynman*" e para a variável `otitle` caso contrário:

```
// (...)
author = rs.getString(3);

if (author == "Feynman") {
      ftitle = rs.getString(2);
} else {
      otitle = rs.getString(2);
}
```

O fragmento de código de SQLJ seguinte pretende representar o processo de obter os registos da tabela `Books` que sejam semelhantes a um autor específico

```
String title, author;
Float price;
#sql iterator Books(String title, Float price);
Books books;

// a aplicação define o autor
// executa a query e abre o cursor
#sql books = {
   SELECT title, price INTO :title, :price
   FROM books WHERE author = :author
};

// obtem os resultados
while(books.next()) {
   System.out.println(books.title() +","+ books.price());
}
books.close();
```

O correspondente fragmento de código JDBC á apresentado de seguida (assumindo a declaração de `price`, `name` e `author`):

```
PreparedStatement stmt = connection.prepareStatement
          ("SELECT title, price FROM books WHERE author = ?");

// define o parâmetro na query e executa
stmt.setString(1, author);

ResultSet rs = smt.executeQuery();

// obtem os resultados
while(rs.next()) {
   System.out.println(rs.getString(1) + "," + rs.getFloat(2));
}
```

Todas as declarações SQLJ têm o prefixo **`#sql`**. Em SQLJ obtemos os resultados com objectos do tipo `iterator`, que são não sua essência cursores. Um `interator` é uma instancia de uma classe `iterator`. Assim, o uso de SQLJ decorre em 5 passos:

- Declarar a classe `Iterator`: no código anterior isto acontece recorrendo a uma declaração: `#SQL iterator Books(String title, Float price);`
- Esta declaração cria uma nova classe Java que pode ser usada para instanciar objectos.
- Instanciar um objecto `iterador` da classe `Iterator`;
- Inicializar o `iterator` usando a declaração SQL. No nosso exemplo isto ocorre na declaração `#sql books = … ;`
- De forma iterativa, proceder à leitura das linhas a partir do objecto `iterator`: passo semelhante à leitura de linhas recorrendo ao `ResultSet` em JDBC;
- Fechar o objecto `iterator`.

Referências

"***Java Enterprise in a Nutshell***“, Capítulo 2 “*JDBC*”
David Flanagan, Jim Farley, William Crawford, Kris Magnusson
O'Reilly, ISBN: 1565924835E

"***Database Programming with JDBC and Java***", 2ª Edição
George Reese
O'Reilly, ISBN: 1565926161

"***The Java Tutorial – JDBC Database Access***"
Java Sun Microsystems
http://java.sun.com/docs/books/tutorial/jdbc/index.html

Bibliografia complementar

Acesso a Bases de Dados relacionais através de ODBC

"***Beginning Java Databases: JDBC, SQL, J2EE, EJB, JSP, XML***"
Kevin Mukhar, Todd Lauinger, John Carnell
Wrox, ISBN: 1861004370

"***Java Cookbook***“, Capítulo 20 “*Database Access*”
Ian Darwin
O'Reilly, ISBN: 0596001703