

JDBC

JDBC é uma interface baseada em Java para acesso a bancos de dados através de SQL.

- *Pacote Java padrão: **java.sql***
- *Baseada em ODBC*

Usando JDBC, pode-se obter acesso direto a bancos de dados através de applets e outras aplicações Java. Este módulo apresenta uma introdução superficial do JDBC mas suficiente para integrar aplicações Java com bancos de dados relacionais que possuam drivers JDBC.

- *Não são abordados Connection Pools nem DataSources*

JDBC é uma interface de **nível de código**

- Código SQL é usado explicitamente dentro do código Java
- O pacote java.sql consiste de um conjunto de **classes e interfaces** que permitem embutir código SQL em métodos.

Com JDBC é possível construir uma aplicação Java para acesso a **qualquer** banco de dados SQL.

- O banco deve ter pelo menos um driver ODBC, se não tiver driver JDBC

Para usar JDBC é preciso ter um **driver JDBC**

- O J2SE distribui um driver ODBC que permite o acesso a bancos que não suportam JDBC mas suportam ODBC

Tipo 1: ponte ODBC-JDBC

- Usam uma ponte para ter acesso a um banco de dados. Este tipo de solução requer a instalação de software do lado do cliente.

Tipo 2: solução com código nativo

- Usam uma API nativa. Esses drivers contém métodos Java implementados em C ou C++. Requer software no cliente.

Tipo 3: solução 100% Java no cliente

- Oferecem uma API de rede via middleware que traduz requisições para API do driver desejado. Não requer software no cliente.

Tipo 4: solução 100% Java

- Drivers que se comunicam diretamente com o banco de dados usando soquetes de rede. É uma solução puro Java. Não requer código adicional do lado do cliente.

Uma aplicação JDBC pode carregar ao mesmo tempo diversos drivers.

Para determinar qual driver será usado usa-se uma URL:

jdbc:<subprotocolo>:<dsn>

- *A aplicação usa o subprotocolo para identificar e seleccionar o driver a ser instanciado.*
- *O dsn é o nome que o subprotocolo utilizará para localizar um determinado servidor ou base de dados.*
- *Sintaxe dependente do fabricante. Veja alguns exemplos:*

jdbc:odbc:anuncios

jdbc:oracle:thin:@200.206.192.216:1521:exemplo

jdbc:mysql://alnitak.orion.org/clientes

jdbc:cloudscape:rmi://host:1098/MyDB;create=true

A interface **Driver** é utilizada apenas pelas implementações de drivers JDBC

- É preciso carregar a classe do driver na aplicação que irá utilizá-lo. Isto pode ser feito com `Class.forName()`:

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

A classe **DriverManager** manipula objetos do tipo **Driver**.

- Possui métodos para registrar drivers, removê-los ou listá-los.
- É usado para retornar **Connection**, que representa uma conexão a um banco de dados, a partir de uma URL JDBC recebida como parâmetro

```
Connection con =  
    DriverManager.getConnection  
        ("jdbc:odbc:dados",  
         "nome", "senha");
```

Interfaces que contém métodos implementados em todos os drivers JDBC.

Connection

- *Representa uma conexão ao banco de dados, que é retornada pelo DriverManager na forma de um objeto.*

Statement

- *Oferece meios de passar instruções SQL para o sistema de bancos de dados.*

ResultSet

- *É um cursor para os dados recebidos.*

Obtendo-se um objeto *Connection*, chama-se sobre ele o método *createStatement()* para obter um objeto do tipo *Statement*:

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

que poderá usar métodos como *execute()*, *executeQuery()*, *executeBatch()* e *executeUpdate()* para enviar instruções SQL ao BD.

Subinterfaces:

- *PreparedStatement* e *CallableStatement*

```
PreparedStatement pstmt =  
    con.prepareStatement(...);  
CallableStatement cstmt = con.prepareCall(...);
```


■ *Exemplo de uso de Statement*

```
stmt.execute("CREATE TABLE dinossauros "  
            + "(codigo INT PRIMARY KEY, "  
            + "genero CHAR(20), "  
            + "especie CHAR(20));");
```

```
int linhasModificadas =  
    stmt.executeUpdate("INSERT INTO dinossauros "  
        + "(codigo, genero, especie) VALUES "  
        + "(499, 'Fernandosaurus', 'brasiliensis')");
```

```
ResultSet cursor =  
    stmt.executeQuery("SELECT genero, especie " +  
        " FROM dinossauros " +  
        " WHERE codigo = 355");
```

O método `executeQuery()`, da interface `Statement`, retorna um objeto `ResultSet`.

- *Cursor para as linhas de uma tabela.*
- *Pode-se navegar pelas linhas da tabela recuperar as informações armazenadas nas colunas*

Os métodos de navegação são

- `next()`, `previous()`, `absolute()`, `first()` e `last()`

Métodos para obtenção de dados:

- `getInt()`
- `getString()`
- `getDate()`
- `getXXX()`, ...

<i>Método de ResultSet</i>	<i>Tipo de dados SQL92</i>
<code>getInt()</code>	INTEGER
<code>getLong()</code>	BIG INT
<code>getFloat()</code>	REAL
<code>getDouble()</code>	FLOAT
<code>getBignum()</code>	DECIMAL
<code>getBoolean()</code>	BIT
<code>getString()</code>	CHAR, VARCHAR
<code>getDate()</code>	DATE
<code>getTime()</code>	TIME
<code>getTimestamp()</code>	TIME STAMP
<code>getObject()</code>	<i>Qualquer tipo (Blob)</i>


```
ResultSet rs =  
    stmt.executeQuery("SELECT Numero, Texto, "  
                        + " Data FROM Anuncios");  
  
while (rs.next()) {  
    int x = rs.getInt("Numero");  
    String s = rs.getString("Texto");  
    java.sql.Date d = rs.getDate("Data");  
    // faça algo com os valores obtidos...  
}
```

Permite a execução atômica de comandos enviados ao banco.
Implementada através dos métodos de Connection

- `commit()`
- `rollback()`
- `setAutoCommit(boolean autoCommit)`: default é true.

Por default, as informações são processadas a medida em que são recebidas. Para mudar:

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

Agora várias instruções podem ser acumuladas.

Para processar:

```
con.commit();
```

Se houver algum erro e todo o processo necessitar ser desfeito, pode-se emitir um ROLLBACK usando:

```
con.rollback();
```

Statement **pré-compilado** que é mais eficiente quando várias queries similares são enviadas com parâmetros diferentes
String com instrução SQL é preparado previamente, deixando-se "?" no lugar dos parâmetros

Parâmetros são inseridos em ordem, com **setXXX()** onde XXX é um tipo igual aos retornados pelos métodos de ResultSet

```
String sql = "INSERT INTO Livros VALUES(?, ?, ?)";  
PreparedStatement cstmt = con.prepareStatement(sql);  
cstmt.setInt(1, 18943);  
cstmt.setString(2, "Lima Barreto");  
cstmt.setString(3, "O Homem que Sabia Javanês");  
cstmt.executeUpdate();  
...
```


Procedimentos desenvolvidos em linguagem proprietária do SGBD (*stored procedures*) podem ser chamados através de objetos `CallableStatement`

Parâmetros são passados da mesma forma que em instruções `PreparedStatement`

Sintaxe

- `con.prepareCall("{ call proc_update(?, ?, ...) })`;
- `con.prepareCall("{ ? = call proc_select(?, ?, ...) })`;

```
CallableStatement cstmt =
```

```
    con.prepareCall("{ ? = call sp_porAssunto(?) }");
```

```
cstmt.setString(2, "520.92");
```

```
ResultSet rs = cstmt.executeQuery();
```

Após o uso, os objetos *Connection*, *Statement* e *ResultSet* devem ser *fechados*. Isto pode ser feito com o método *close()*:

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