





Acesso a dados com JDBC

Acessando dados em Java com JDBC

Daniel Destro do Carmo
Softech Network Informática
daniel@danieldestro.com.br





Acesso a dados com JDBC

Objetivo do curso

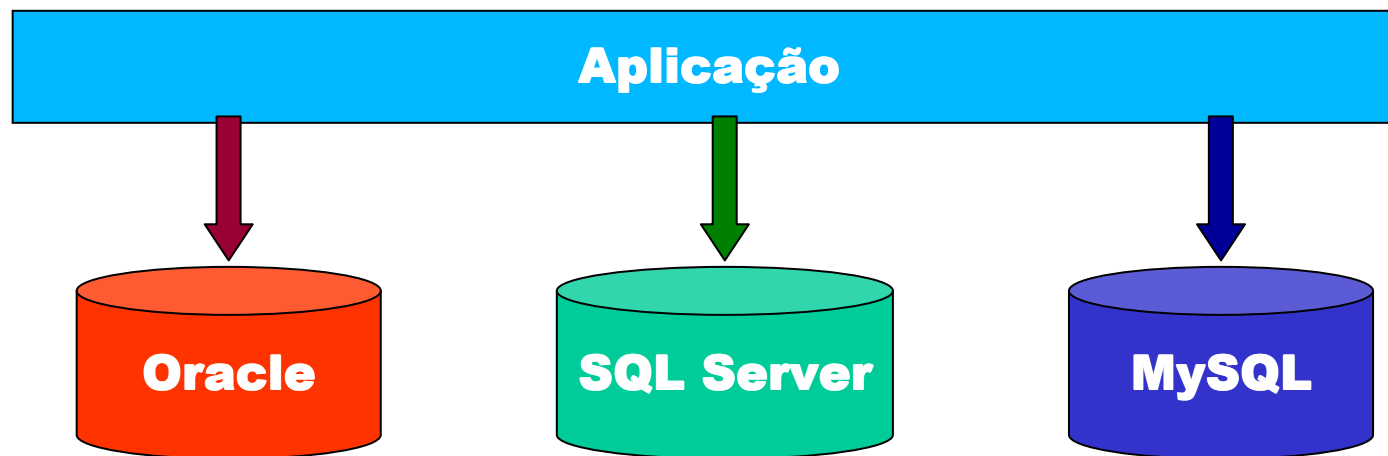
- ✓ O que é JDBC?
- ✓ Arquitetura da API
- ✓ Detalhes e uso da API



Acesso a dados com JDBC

O que é JDBC?

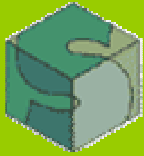
Diferentes bancos de dados relacionais possuem diferentes formas de se comunicar com uma aplicação que necessite acessar os seus dados.



Isto causa um grande problema de codificação e manutenção nas aplicações que precisam trabalhar com diversos banco de dados e também requer o aprendizado de uma nova API para cada BD diferente.

Isso não torna a aplicação flexível.



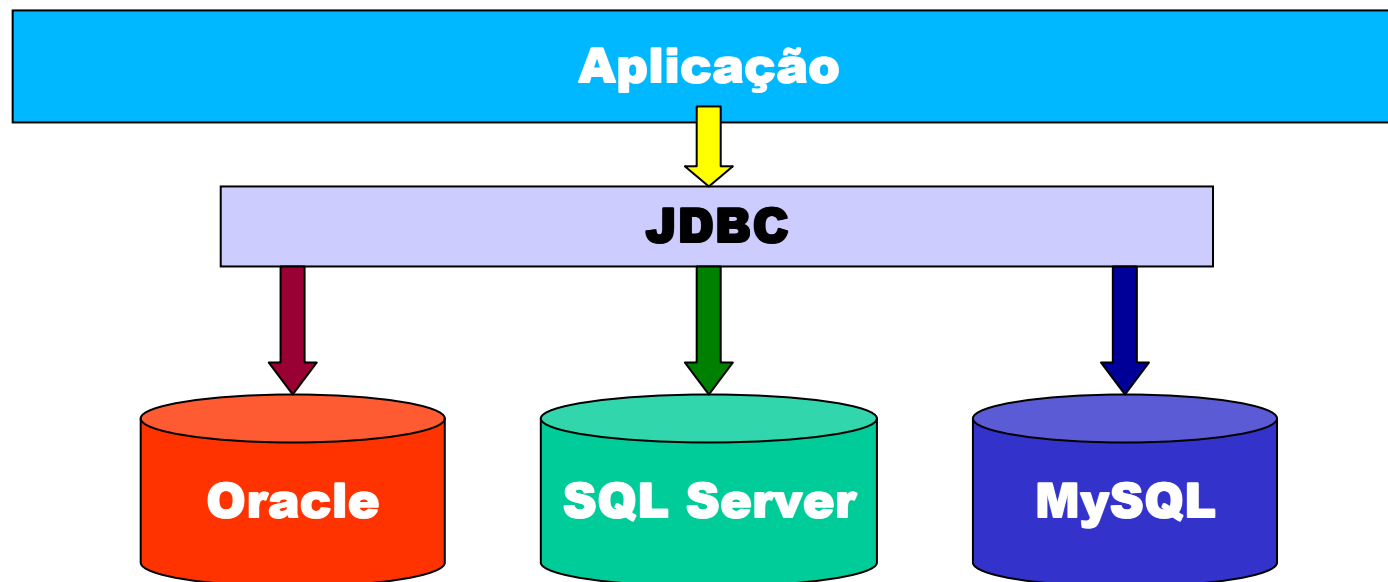


Acesso a dados com JDBC

O que é JDBC?

A Sun desenvolveu a API JDBC, com a intenção de uniformizar os acessos aos diferentes bancos de dados relacionais, dando maior flexibilidade aos sistemas.

JDBC = Java DataBase Connectivity



Com JDBC as chamadas ao BD são padronizadas, apesar de que os comandos SQL podem variar de banco para banco, se não for usado o SQL padrão.





Acesso a dados com JDBC

O que é JDBC?

A biblioteca da JDBC provê um conjunto de interfaces de acesso ao BD.

Uma implementação em particular dessas interfaces é chamada de **driver**.

Os próprios fabricantes dos bancos de dados (ou terceiros) são quem implementam os drivers JDBC para cada BD, pois são eles que conhecem detalhes dos BDs.

Cada BD possui um Driver JDBC específico (que é usado de forma padrão - JDBC).

A API padrão do Java já vem com o driver JDBC-ODBC, que é uma ponte entre a aplicação Java e o banco através da configuração de um recurso ODBC na máquina.

O drivers de outros fornecedores devem ser adicionados ao CLASSPATH da aplicação para poderem ser usados.

Desta maneira, pode-se mudar o driver e a aplicação não muda.





Acesso a dados com JDBC

O que é JDBC?

Tipos de Drivers JDBC:

Tipo 1 - Driver Ponte JDBC-ODBC

É uma implementação nativa que conecta uma aplicação Java a um banco de dados através de ODBC configurado na máquina.

Tipo 2 - Driver API-Nativa Parcialmente Java

É uma “casca” sobre uma implementação nativa de um driver de acesso ao banco (ex: este driver utiliza o OCI para Oracle). Um erro neste driver nativo pode derrubar a JVM.

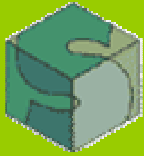
Tipo 3 - Driver Java c/ Net-Protocol

Utiliza um middleware para a conexão com o banco de dados.

Tipo 4 - Driver Java Puro

Driver totalmente implementado em Java. Conhece todo o protocolo de comunicação com o BD e pode acessar o BD sem software extra. É o tipo de driver mais indicado.

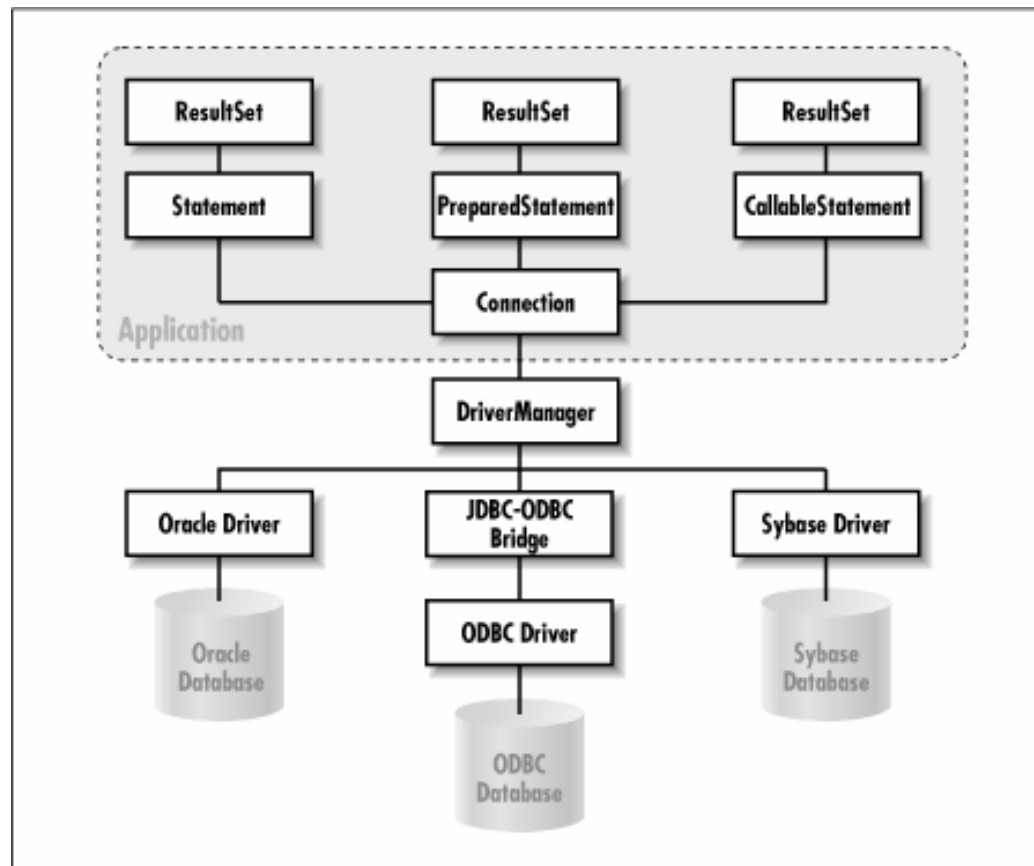




Acesso a dados com JDBC

Arquitetura da JDBC

Arquitetura básica da API JDBC:
pacote: *java.sql*





Acesso a dados com JDBC

Arquitetura da JDBC

As principais classes e interfaces do pacote *java.sql* são:

DriverManager - gerencia o driver e cria uma conexão com o banco.

Connection - é a classe que representa a conexão com o bando de dados.

Statement - controla e executa uma instrução SQL .

PreparedStatement - controla e executa uma instrução SQL. É melhor que Statement.

ResultSet - contém o conjunto de dados retornado por uma consulta SQL.

ResultSetMetaData - é a classe que trata dos metadados do banco.

A interface Connection possui os métodos para criar um Statement, fazer o commit ou rollback de uma transação, verificar se o auto commit está ligado e poder (des)ligá-lo, etc.

As interfaces Statement e PreparedStatement possuem métodos para executar comandos SQL.

ResultSet possui método para recuperar os dados resultantes de uma consulta, além de retornar os metadados da consulta.

ResultSetMetaData possui métodos para recuperar as meta informações do banco.





Acesso a dados com JDBC

Utilizando a JDBC

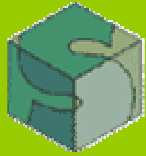
Para a aplicação Java se comunicar com um banco de dados e acessar os seus dados, uma conexão com o BD deve ser estabelecida.

A conexão é estabelecida de seguinte forma:

- Carregamento do driver JDBC específico
- Criação da conexão com o BD

A partir da conexão é possível interagir com o BD, fazendo consultas, atualização e busca às meta informações da base e das tabelas.





Acesso a dados com JDBC

Utilizando a JDBC

Carregamento do driver JDBC específico:

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

A String passada ao método `forName()` é o nome completo qualificado (*fully qualified name*) da classe que implementa o Driver JDBC de cada banco de dados. No exemplo é usado o nome do driver da ponte JDBC-ODBC. Cada driver possui um nome diferente, consulte a documentação do fabricante.

Desta forma o carregamento do driver é feito de forma dinâmica (via `Class.forName()`), dando flexibilidade ao código.

Para maior flexibilidade, o nome poderia estar contido em um arquivo de configuração externo, podendo ser alterado sem a necessidade de recompilar o fonte Java.





Acesso a dados com JDBC

Utilizando a JDBC

Criação da conexão com o BD:

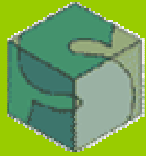
```
Connection conn = DriverManager.getConnection(  
    "url",  
    "usuario",  
    "senha"  
);
```

Após o carregamento do driver, a classe DriverManager é a responsável por se conectar ao banco de dados e devolver um objeto Connection, que representa a conexão com o BD.

O parâmetro "url" também é específico de cada fornecedor de driver, consulte a documentação. Geralmente, na url são informados o IP ou nome do servidor, porta e o nome da base de dados (database ou instância).

Os outros argumentos são o nome do usuário do banco e sua senha de acesso.





Acesso a dados com JDBC

Utilizando a JDBC

Com a conexão estabelecida já é possível interagir com o BD de várias formas:

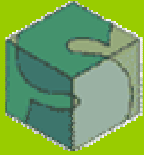
- Criar tabelas e outros elementos
- Inserir, Alterar e Remover registros
- Buscar registros
- Buscar as meta informações do banco

As três primeiras interações se dão por meio das interfaces Statement ou PreparedStatement, que veremos a seguir.

Todos os exemplos estão baseados no uso do driver JDBC fornecido pela Oracle. Porém, os mesmos exemplos podem ser usados com qualquer banco de dados relacional que possua um driver JDBC, necessitando apenas trocar o nome do driver e a URL de conexão.

Lembrando que o JAR do driver JDBC, fornecido pelo fabricante, precisa ser disponibilizado no CLASSPATH da aplicação.





Acesso a dados com JDBC

Statement

O exemplo abaixo se conecta ao banco de dados local, na instância **orcl** e cria uma tabela chamada PESSOA, com os campos ID e NOME.

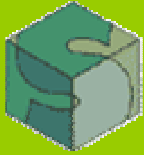
```
String driver = "oracle.jdbc.driver.OracleDriver";
String url = "jdbc:oracle:thin:@127.0.0.1:1521:orcl";
String user = "usuario";
String password = "senha";
Class.forName( driver );
Connection conn = DriverManager.getConnection( url, user, password );
String sql = "CREATE TABLE PESSOA ( ID NUMBER, NOME VARCHAR2(100) )";
Statement st = conn.createStatement();
st.executeUpdate( sql );
st.close();
conn.close();
```

Através do método `createStatement()` de `Connection` é retornado um objeto `Statement`, que é usado para executar um comando SQL no BD.

O método `executeUpdate()` de `Statement` recebe o SQL que será executado. Este método deve ser usado para DDLs e comandos SQL de INSERT, UPDATE ou DELETE.

Depois os métodos `close()` de `Statement` e `Connection` são invocados para liberar os recursos.





Acesso a dados com JDBC

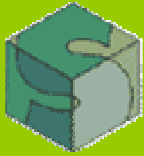
Statement

O objeto Statement criado pode ser reusado várias vezes para executar diferentes comandos SQL. Isto é até recomendado.

```
String driver = "oracle.jdbc.driver.OracleDriver";
String url = "jdbc:oracle:thin:@127.0.0.1:1521:orcl";
String user = "usuario";
String password = "senha";
Class.forName( driver );
Connection conn = DriverManager.getConnection( url, user, password );
Statement st = conn.createStatement();
String sql = "INSERT INTO PESSOA VALUES (1,'FERNANDO HENRIQUE')";
st.executeUpdate( sql );
sql = "INSERT INTO PESSOA VALUES (2,'LUÍS INÁCIO')";
st.executeUpdate( sql );
sql = "INSERT INTO PESSOA VALUES (3,'ANTÔNIO CARLOS')";
st.executeUpdate( sql );
st.close();
conn.close();
```

Porém, o Statement só pode ser liberado com o método close() ao final das execuções de todos os comandos SQL.





Acesso a dados com JDBC

ResultSet

A recuperação de dados do BD é trivial e é feita através da execução de uma *query* SQL, e os dados podem ser recuperados através da interface ResultSet, retornada na execução da query.

```
Connection conn = ...
String sql = "SELECT ID, NOME FROM PESSOA";
Statement st = conn.createStatement();
ResultSet rs = st.executeQuery( sql );
while( rs.next() ) {
    System.out.println( rs.getString("ID") );
    System.out.println( rs.getString("NOME") );
}
rs.close();
st.close();
conn.close();
```

O método `executeQuery()` de `Statement` executa uma consulta (query) SQL e retorna um objeto `ResultSet`, que contém os dados recuperados.

Através do método `next()` de `ResultSet`, o cursor interno é movido para o próximo registro. O método retorna `false` caso não existam mais registros ou `true` caso contrário.

Os valores dos registros podem ser recuperados como o método `getString()`, que recebe o nome do campo ou seu *alias*.





Acesso a dados com JDBC

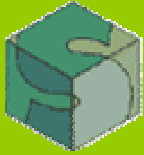
ResultSet

Além do método `getString()`, a interface `ResultSet` dispõe de outros métodos para recuperar os dados do BD diretamente nos tipos mais adequados. Os tipos relacionados são os mais indicados, porém outros tipos podem ser usados.

Todos os métodos recebem o nome do campo (ou *alias*) ou o número do campo no select, começando por 1 (um).

Método	Tipos (<code>java.sql.Types</code>)
<code>getByte</code>	TINYINT
<code>getShort</code>	SMALLINT
<code>getInt</code>	INTEGER
<code>getLong</code>	BIGINT
<code>getFloat</code>	REAL
<code>getDouble</code>	FLOAT, DOUBLE
<code>getBigDecimal</code>	DECIMAL, NUMERIC
<code>getBoolean</code>	CHAR
<code>getString</code>	VARCHAR, LONGVARCHAR
<code>getBytes</code>	BINARY, VARBINARY
<code>getDate</code>	DATE
<code>getTime</code>	TIME
<code>getTimestamp</code>	TIMESTAMP
<code>getAsciiStream</code>	LONGVARCHAR
<code>getUnicodeStream</code>	LONGVARCHAR
<code>getBinaryStream</code>	LONGVARBINARY
<code>getObject</code>	Todos os tipos





Acesso a dados com JDBC

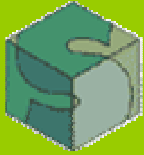
ResultSet

Exemplos:

```
String sql = "SELECT ID, NOME FROM PESSOA";
Statement st = conn.createStatement();
ResultSet rs = st.executeQuery( sql );
while( rs.next() ) {
    System.out.println( rs.getInt("ID") );
    System.out.println( rs.getString("NOME") );
}
rs.close();
st.close();
conn.close();
```

```
String sql = "SELECT ID, NOME FROM PESSOA";
Statement st = conn.createStatement();
ResultSet rs = st.executeQuery( sql );
while( rs.next() ) {
    System.out.println( rs.getInt(1) );
    System.out.println( rs.getString(2) );
}
rs.close();
st.close();
conn.close();
```





Acesso a dados com JDBC

Adicionando parâmetros

Geralmente os comandos SQL recebem valores externos para que trabalhem com dados dinâmicos na aplicação.

Com o uso de Statement, isso é feito simplesmente concatenando o valor ao comando SQL a ser executado.

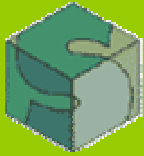
```
String id = "1";  
String sql = "SELECT ID, NOME FROM PESSOA WHERE ID=" + id;  
Statement st = conn.createStatement();  
ResultSet rs = st.executeQuery( sql );
```

```
String sql = "INSERT INTO PESSOA VALUES (" + id + ", ' " + nome + " ' )";  
Statement st = conn.createStatement();  
ResultSet rs = st.executeQuery( sql );
```

Esta forma, apesar de ser correta, não é a melhor e nem a mais indicada, pois torna o trabalho de concatenação extremamente chato e passível de erros na montagem do comando SQL, além de permitir que outros comandos SQL sejam embutidos e executados maliciosamente.

A melhor maneira é, certamente, trocar o uso de Statement por PreparedStatement.





Acesso a dados com JDBC

PreparedStatement

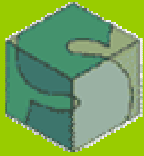
PreparedStatement tem vantagens sobre Statement, pois ela cria instruções SQL pré-compiladas, economizando recursos do próprio banco de dados, otimizando o programa.

Além da facilidade de se adicionar parâmetros aos comandos SQL.

```
String sql = "INSERT INTO PESSOA VALUES ( ?, ? )";
PreparedStatement pst = conn.prepareStatement( sql );
pst.setInt(1, 4);
pst.setString(2, "Mário");
pst.executeUpdate();
pst.close();
conn.close();
```

```
String sql = "SELECT ID, NOME FROM PESSOA WHERE NOME LIKE ?";
PreparedStatement pst = conn.prepareStatement( sql );
pst.setString(1, "%Fer%");
ResultSet rs = pst.executeQuery();
while( rs.next() ) {
    System.out.println( rs.getInt(1) + " / " + rs.getString(2) );
}
rs.close();
pst.close();
conn.close();
```





Acesso a dados com JDBC

PreparedStatement

Os parâmetros, com PreparedStatement, podem ser passados diretamente com o próprio tipo que o dado foi definido, evitando assim ter que ficar formatando ou convertendo campos de data e decimais, além de poder passar inclusive valores nulos (NULL).

```
Integer id = new Integer(5);
String nome = null;
String sql = "INSERT INTO PESSOA VALUES ( ?, ? )";
PreparedStatement pst = conn.prepareStatement( sql );
pst.setObject(1, id, java.sql.Types.NUMERIC );
pst.setString(2, nome, java.sql.Types.VARCHAR );
pst.executeUpdate();
pst.close();
conn.close();
```





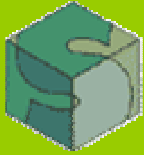
Acesso a dados com JDBC

Controle de Transações

A interface Connection oferece métodos para o controle transacional. Primeiro é necessário definir o auto-commit como *false*. Para confirmar a transação o método commit() deve ser invocado. O método rollback() desfaz a transação.

```
conn.setAutoCommit( false );
String sql = "INSERT INTO PESSOA (ID,NOME) VALUES (?,?)";
PreparedStatement pst = conn.prepareStatement( sql );
try {
    pst.setInt(1, 5);
    pst.setString(2, "Carlos");
    pst.executeUpdate();
    sql = "UPDATE PESSOA SET NOME=? WHERE ID=?";
    pst = conn.prepareStatement( sql );
    pst.setInt(1, 3);
    pst.setString(2, "Jorge");
    pst.executeUpdate();
    conn.commit();
} catch(SQLException e) {
    conn.rollback();
}
pst.close();
conn.close();
```





Acesso a dados com JDBC

Acessando Procedures

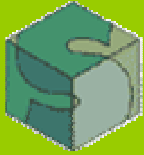
Algumas empresas possuem grande parte do seu código legado em Stored Procedures, e trocar estas SPs por Java seria muito caro, portanto é mais adequado utilizar o que já existe.

As chamadas a Stored Procedures são possíveis por meio da interface CallableStatement.

O exemplo abaixo executa uma simples procedure no BD, sem parâmetros e sem retorno.

```
Connection conn = ...  
String sp = "{ call MINHA_STORED_PROCEDURE }";  
CallableStatement cs = conn.prepareCall( sp );  
cs.execute();  
cs.close();  
conn.close();
```





Acesso a dados com JDBC

Acessando Procedures

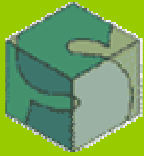
Geralmente as Stored Procedures recebem argumentos e também retornam valores através desses argumentos. Isso tudo é possível com a CallableStatement.

O exemplo abaixo passa dois parâmetros de entrada para a procedure. O segundo exemplo passa um argumento de entrada e recupera ao segundo argumento, do tipo DATE.

```
Connection conn = ...
String sp = "{ call MINHA_STORED_PROCEDURE(?,?) }";
CallableStatement cs = conn.prepareCall( sp );
cs.setInt(1, 10);
cs.setString(2, 99);
cs.execute();
cs.close();
conn.close();
```

```
Connection conn = ...
String sp = "{ call MINHA_STORED_PROCEDURE(?,?) }";
CallableStatement cs = conn.prepareCall( sp );
cs.setInt(1, 10);
cs.registerOutParameter(2, java.sql.Types.DATE);
cs.execute();
java.sql.Date d = cs.getDate(2);
cs.close();
conn.close();
```





Acesso a dados com JDBC

Acessando Procedures

As Stored Procedures (ou Functions) ainda podem retornar valores e cursores com dados.

```
String sp = "{ ? = call SOMAR_VALORES(?,?) }";
CallableStatement cs = conn.prepareCall( sp );
cs.setInt(2, 10);
cs.setString(3, 99);
cs.registerOutParameter(1, java.sql.Types.NUMBER);
cs.execute();
int total = cs.getInt(1);
System.out.println( "Total: " + i );
cs.close();
conn.close();
```

```
String sp = "{ ? = call SP_QUE_BUSCA_DADOS(?) }";
CallableStatement cs = conn.prepareCall( sp );
cs.setString(2, "São Paulo");
cs.registerOutParameter(1, oracle.jdbc.driver.OracleTypes.CURSOR);
cs.execute();
ResultSet rs = (ResultSet) cs.getObject(1);
// Trabalha os dados do ResultSet ...
rs.close();
cs.close();
conn.close();
```





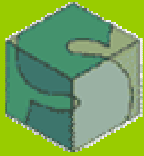
Acesso a dados com JDBC

Tratamento de Erros

Quase todos os métodos da API JDBC podem lançar uma exceção do tipo `SQLException`, que representa algum tipo de erro ocorrido no acesso ao banco, seja falha na conexão, SQL mal formado, até violação de PK etc.

A classe `SQLException` possui os métodos:

- `getMessage()` - retorna a mensagem de erro.
- `getSQLState()` - retorna um dos códigos de estado do padrão ANSI-92 SQL.
- `getErrorCode()` - retorna o código de erro específico do fornecedor.
- `getNextException()` - retorna a exceção aninhada (encadeada), se houver.



Acesso a dados com JDBC

Tratamento de Erros

Exemplo de tratamento de erro:

```
try {  
  
    //simula um erro de SQL mal formado  
    String sql = "INSERT PESSOA (ID,NOME) VALUES (?,?)";  
    PreparedStatement ps = conn.prepareStatement( sql );  
    ps.executeUpdate();  
  
} catch( SQLException ex ) {  
  
    System.out.println("\n--- SQLException ---\n");  
    while( ex != null ) {  
        System.out.println("Mensagem: " + ex.getMessage());  
        System.out.println("SQLState: " + ex.getSQLState());  
        System.out.println("ErrorCode: " + ex.getErrorCode());  
        ex = ex.getNextException();  
        System.out.println("");  
    }  
  
}
```





Acesso a dados com JDBC

Websites Relacionados

Site Oficial do Java

<http://java.sun.com>

Tutorial Oficial da JDBC

<http://java.sun.com/docs/books/tutorial/jdbc/index.html>

Documentação da API da JDBC

<http://java.sun.com/j2se/1.4.2/docs/api/java/sql/package-summary.html>

