

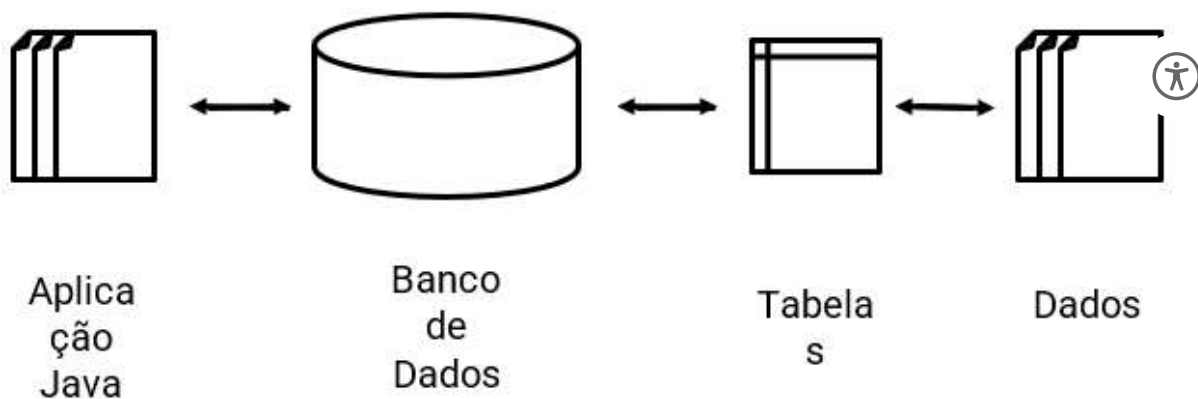


# Banco de Dados com Java

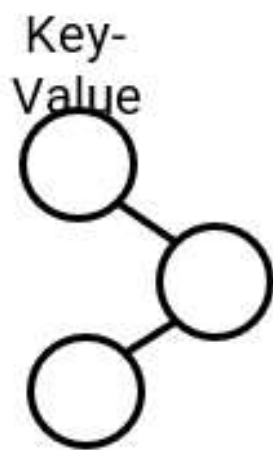
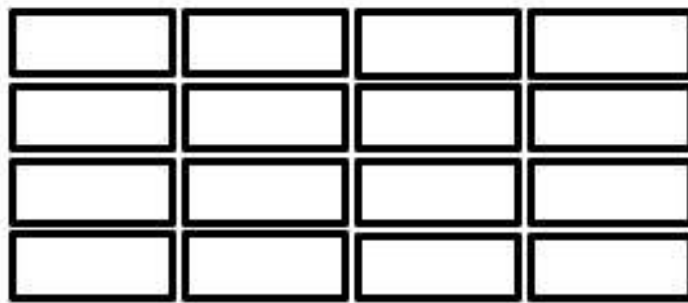
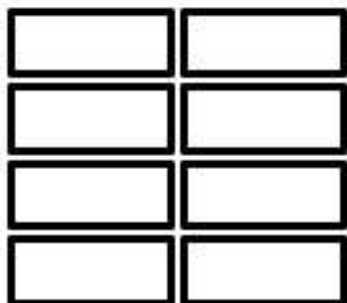


À medida que uma aplicação exige o registro e consulta de um grande volume de informações, apenas o uso de arquivos se torna algo complexo e custoso. Como alternativa, existem os **banco de dados**, ou seja, coleções organizadas de dados estruturados. Essa organização e estrutura dependem da escolha da pessoa programadora. Atualmente estão disponíveis no mercado dois tipos de bancos de dados: **relacional** e **não relacional**.

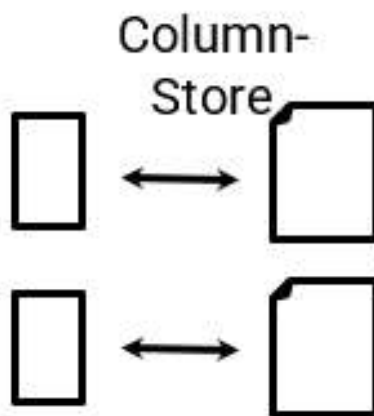
O **modelo relacional** é baseado na proposta de E.F. Codd de 1970, uma forma intuitiva e direta de representar os dados em tabelas. Neste modelo, cada linha da tabela é um registro com uma identificação única chamada de **chave primária**. As colunas contêm os atributos dos dados e, cada registro geralmente tem um valor para cada atributo, facilitando o estabelecimento das relações entre dados. Existem diversos sistemas diferentes para gerenciar bancos de dados relacionais. Esses sistemas são conhecidos como **sistemas de gerenciamento de bancos de dados relacionais (SGBDR)**. O mais popular entre eles é o MySQL, mas também há outras opções como: Oracle database, Microsoft SQL Server, e Postgres. A grande maioria dos SGBDRs oferecem a opção de usar a *Structured Query Language* (SQL) para consulta e manutenção dos bancos de dados.



As **bases de dados não relacionais** são caracterizadas exatamente pelo o que o nome já diz, ou seja, por não seguir o modelo relacional dos sistemas de gerenciamento tradicionais. Essa categoria também é conhecida por **NoSQL**. Os mais populares do mercado são: MongoDB, DocumentDB, Cassandra, Couchbase, HBase, Redis, e Neo4j. Estes bancos de dados são geralmente agrupados em quatro categorias: *Key-value stores*, *Graph stores*, *Column stores*, e *Document stores*.



Graph



Document

O processo de criação de um banco de dados depende muito do sistema a ser utilizado. Na maior parte deles, basta entrar no site oficial e utilizar o instalador oficial. Feito isso, um usuário administrador será criado (com a senha que o usuário inserir) e, através da linha de comando no terminal ou por meio de uma ferramenta visual, o usuário já tem a possibilidade de criar bancos, tabelas, inserir dados, consultar dados e assim por diante; seja em um banco relacional ou não.

## O que é o JDBC?

JDBC meios Java Data Base Connectivity, que é uma API Java padrão para conectividade independente do banco de dados entre a linguagem de programação Java e uma vasta gama de bases de dados.

A biblioteca JDBC inclui APIs para cada uma das tarefas mencionadas abaixo, geralmente associadas ao uso do banco de dados.

- Fazendo uma conexão com um banco de dados.
- Criando instruções SQL ou MySQL.
- Executando consultas SQL ou MySQL no banco de dados.
- Visualizando e modificando os registros resultantes.

Fundamentalmente, o JDBC é uma especificação que fornece um conjunto completo de interfaces que permite acesso portátil a um banco de dados subjacente. Java pode ser usado para escrever diferentes tipos de executáveis, como -

- Aplicações Java
- Applets Java

- Servlets Java



- JSPs (Java ServerPages)
- Enterprise JavaBeans (EJBs).

Todos esses executáveis diferentes podem usar um driver JDBC para acessar um banco de dados e tirar proveito dos dados armazenados.

O JDBC fornece os mesmos recursos que o ODBC, permitindo que os programas Java contenham código independente do banco de dados.

### **Os pacotes JDBC 4.0**

O `java.sql` e o `javax.sql` são os principais pacotes do JDBC 4.0. Ele oferece as principais classes para interagir com suas fontes de dados.

Os novos recursos desses pacotes incluem alterações nas seguintes áreas:

- Carregamento automático do driver de banco de dados.
- Exceção ao lidar com melhorias.
- Funcionalidade aprimorada de BLOB / CLOB.
- Aprimoramentos na interface de conexão e instrução.
- Suporte ao conjunto de caracteres nacionais.
- Acesso SQL ROWID.
- Suporte ao tipo de dados XML do SQL 2003.
- Anotações.

## Arquitetura JDBC



A API JDBC suporta modelos de processamento de duas e três camadas para acesso ao banco de dados, mas, em geral, a arquitetura JDBC consiste em duas camadas -

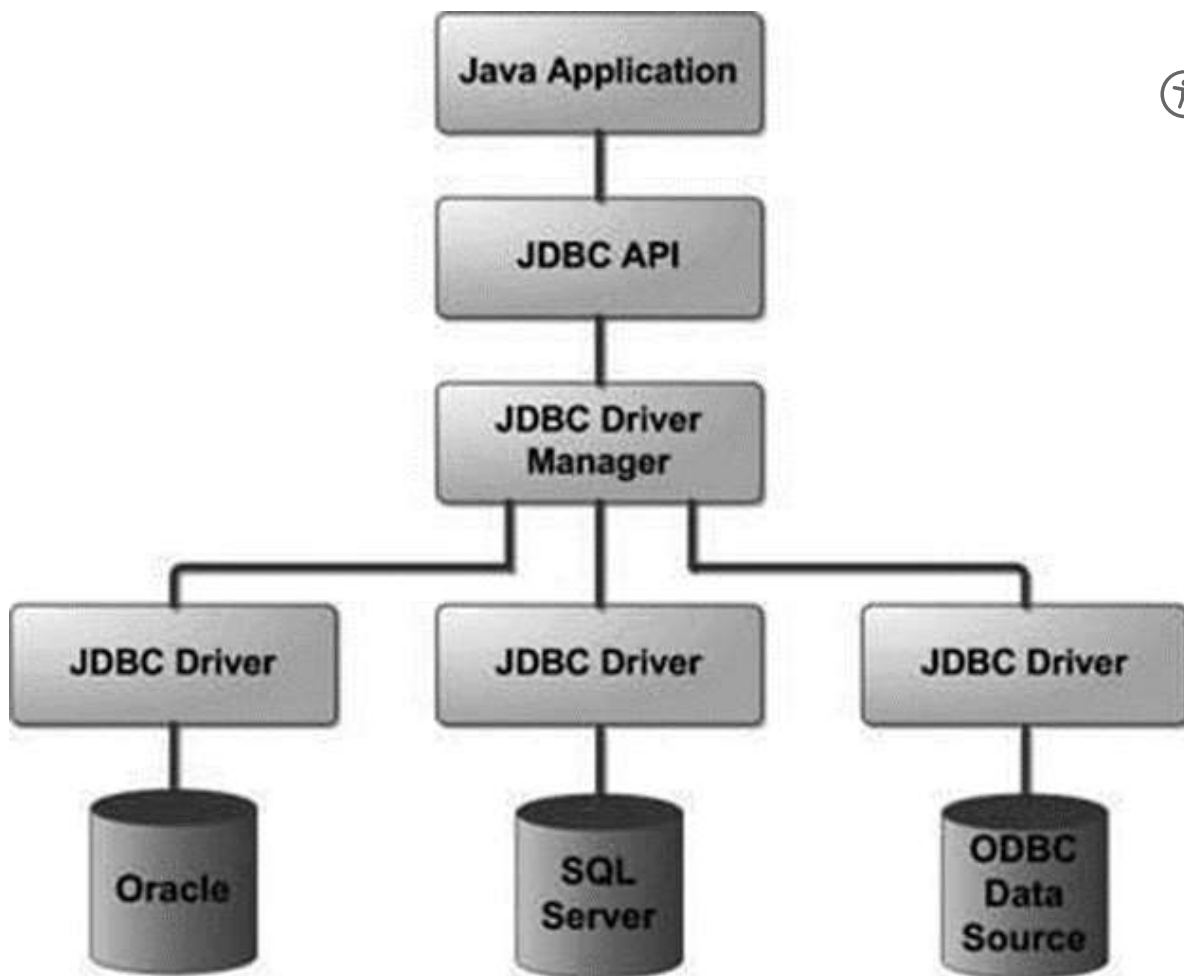
API JDBC: fornece a conexão do aplicativo ao JDBC Manager.

API do driver JDBC: suporta a conexão JDBC Manager-to-Driver.

A API JDBC usa um gerenciador de drivers e drivers específicos do banco de dados para fornecer conectividade transparente a bancos de dados heterogêneos.

O gerenciador de drivers JDBC assegura que o driver correto seja usado para acessar cada fonte de dados. O gerenciador de drivers é capaz de suportar vários drivers simultâneos conectados a vários bancos de dados heterogêneos.

A seguir, está o diagrama da arquitetura, que mostra a localização do gerenciador de drivers em relação aos drivers JDBC e ao aplicativo Java.



## Componentes JDBC comuns

A API JDBC fornece as seguintes interfaces e classes -

## Os pacotes JDBC 4.0

O `java.sql` e o `javax.sql` são os principais pacotes do JDBC 4.0. Ele oferece as principais classes para interagir com suas fontes de dados.

Os novos recursos desses pacotes incluem alterações nas seguintes áreas -

## JDBC - Conexões com o Banco de Dados

Depois de instalar o driver apropriado, é hora de estabelecer uma conexão com o banco de dados usando o JDBC.



A programação envolvida para estabelecer uma conexão JDBC é bastante simples. Aqui estão quatro etapas simples -

## **Importar pacotes JDBC**

As instruções Import informam ao compilador Java onde encontrar as classes mencionadas no seu código e são colocadas no início do seu código-fonte.

Para usar o pacote JDBC padrão, que permite selecionar, inserir, atualizar e excluir dados nas tabelas SQL, adicione as seguintes importações ao seu código-fonte.

```
import java.sql.* ; // para programas padrão JDBC
```

```
import java.math.* ; // para suporte BigDecimal e BigInteger
```

## **Registrar Driver JDBC**

Você deve registrar o driver no seu programa antes de usá-lo. Registrar o driver é o processo pelo qual o arquivo de classe do driver Oracle é carregado na memória, para que possa ser utilizado como uma implementação das interfaces JDBC.

Você precisa fazer esse registro apenas uma vez no seu programa. Você pode registrar um driver de duas maneiras.

## **Abordagem I - Class.forName()**

A abordagem mais comum para registrar um driver é usar o método Class.forName () de Java , para carregar dinamicamente o arquivo de classe do driver na memória,

que o registra automaticamente. Esse método é preferível, pois permite que você faça o registro do driver configurável e portátil.



O exemplo a seguir usa `Class.forName ()` para registrar o driver Oracle.

- **DriverManager:** Esta classe gerencia uma lista de drivers de banco de dados. Corresponde às solicitações de conexão do aplicativo java com o driver de banco de dados apropriado usando o subprotocolo de comunicação. O primeiro driver que reconhece um determinado subprotocolo no JDBC será usado para estabelecer uma conexão com o banco de dados.
- **Driver:** Essa interface lida com as comunicações com o servidor de banco de dados. Você interage diretamente com os objetos Driver muito raramente. Em vez disso, você usa objetos DriverManager, que gerencia objetos desse tipo. Ele também abstrai os detalhes associados ao trabalho com objetos Driver.
- **Connection:** Essa interface com todos os métodos para entrar em contato com um banco de dados. O objeto de conexão representa o contexto de comunicação, ou seja, toda a comunicação com o banco de dados é apenas através do objeto de conexão.
- **Statement:** Você usa objetos criados nessa interface para enviar as instruções SQL ao banco de dados. Algumas interfaces derivadas aceitam parâmetros além de executar procedimentos armazenados.
- **ResultSet:** esses objetos mantêm os dados recuperados de um banco de dados após a execução de uma consulta SQL usando objetos Statement. Ele atua como um iterador para permitir que você mova seus dados.
- **SQLException:** esta classe lida com quaisquer erros que ocorram em um aplicativo de banco de dados.
- **Carregamento automático do driver de banco de dados.**





- Exceção ao lidar com melhorias.
- Funcionalidade aprimorada de BLOB / CLOB.
- Aprimoramentos na interface de conexão e instrução.
- Suporte ao conjunto de caracteres nacionais.
- Acesso SQL ROWID.
- Suporte ao tipo de dados XML do SQL 2003.
- Anotações.
- Importar Pacotes JDBC: Inclua instruções de importação em seu programa Java para importar as classes necessárias em seu código Java.
- Registrar driver JDBC: Esta etapa faz com que a JVM carregue a implementação do driver desejado na memória para que possa atender às suas solicitações JDBC.
- Formulação de URL do banco de dados: serve para criar um endereço formatado corretamente que aponte para o banco de dados ao qual você deseja se conectar.
- Criar objeto de conexão: Finalmente, codifique uma chamada para o método `getConnection()` do objeto `DriverManager` para estabelecer a conexão real com o banco de dados.

```
try {
    Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException ex) {
    System.out.println("Erro: incapaz de carregar a classe do driver!");
    System.exit(1);
}
```



Você pode usar o método `getInstance ()` para solucionar JVMs não compatíveis, mas precisará codificar duas exceções extras da seguinte maneira:

```
try {
    Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
}
catch(ClassNotFoundException ex) {
    System.out.println("Erro: incapaz de carregar a classe do driver!");
    System.exit(1);
catch(IllegalAccessException ex) {
    System.out.println("Erro: problema de acesso ao carregar!");
    System.exit(2);
catch(InstantiationException ex) {
    System.out.println("Error: incapaz de carregar o driver!");
    System.exit(3);
}
```

## Abordagem II - `DriverManager.registerDriver ()`

A segunda abordagem que você pode usar para registrar um driver é usar o método estático `DriveManager.regiserDriver ()`.

Você deve usar o método `registerDriver ()` se estiver usando uma JVM não compatível com JDK, como a fornecida pela Microsoft.

O exemplo a seguir usa `registerDriver ()` para registrar o driver Oracle.



```
try {
    Driver meuDriver = new oracle.jdbc.driver.OracleDriver();
    DriverManager.registerDriver( meuDriver );
}
catch(ClassNotFoundException ex) {
    System.out.println("Error: incapaz de carregar o driver!");
    System.exit(1);
}
```

## Formulação de URL do banco de dados

Depois de carregar o driver, você pode estabelecer uma conexão usando o método `DriverManager.getConnection ()`. Para facilitar a referência, deixe-me listar os três métodos `DriverManager.getConnection ()` sobrecarregados -

- `getConnection (URL da string)`
- `getConnection (URL da string, propriedades Prop)`
- `getConnection (URL da string, usuario da String, senha da String)`

Aqui cada formulário requer um URL do banco de dados. Um URL de banco de dados é um endereço que aponta para seu banco de dados.

Formular uma URL de banco de dados é onde ocorre a maioria dos problemas associados ao estabelecimento de uma conexão.

A tabela a seguir lista os nomes populares dos drivers JDBC e a URL do banco de dados.

RDBMS	Nome do driver JDBC	Formato de URL
mySQL	com.mysql.jdbc.Driver	jdbc:mysql://hostname/databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname:portNumber:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:portNumber/databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:hostname: portNumber/databaseName



Toda a parte destacada no formato de URL é estática e você precisa alterar apenas a parte restante conforme a configuração do banco de dados.

## Criar objeto de conexão

Listamos três formas do método `DriverManager.getConnection ()` para criar um objeto de conexão.

## Usando um URL de banco de dados com um nome de usuário e senha

A forma mais usada de `getConnection ()` requer que você transmita uma URL de banco de dados, um nome de usuário e uma senha :

Supondo que você esteja usando o driver thin do Oracle , você especificará um valor `host: port: databaseName` para a parte do banco de dados da URL.

Se você tiver um host no endereço TCP / IP 192.0.0.1 com o nome de host root e o ouvinte do Oracle estiver configurado para escutar na porta 1521, e o nome do

banco de dados para FUNC, o URL completo do banco de dados será:



```
jdbc:oracle:thin:@root:1521:FUNC
```

Agora você precisa chamar o método `getConnection ()` com nome de usuário e senha apropriados para obter um objeto `Connection` da seguinte maneira -

```
String URL = "jdbc:oracle:thin:@root:1521:FUNC";  
String USER = "usuário";  
String PASS = "senha";  
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

### Usando apenas um URL de banco de dados

Uma segunda forma do método `DriverManager.getConnection ()` requer apenas uma URL do banco de dados.

```
DriverManager.getConnection(String url);
```

No entanto, nesse caso, o URL do banco de dados inclui o nome de usuário e a senha e possui a seguinte forma geral.

```
jdbc:oracle:driver:username/password@database
```

Portanto, a conexão acima pode ser criada da seguinte maneira.



```
String URL = "jdbc:oracle:thin:username/password@root:1521:FUNC";  
Connection conn = DriverManager.getConnection(URL);
```

## Usando uma URL de banco de dados e um objeto de propriedades

Uma terceira forma do método `DriverManager.getConnection()` requer uma URL do banco de dados e um objeto `Properties`.


```
DriverManager.getConnection(String url, Properties info);
```

Um objeto `Properties` contém um conjunto de pares de valores de palavras-chave. É usado para passar propriedades do driver para o driver durante uma chamada para o método `getConnection()`.

Para fazer a mesma conexão estabelecida pelos exemplos anteriores, use o seguinte código:

```
import java.util.*;  
  
String URL = "jdbc:oracle:thin:@root:1521:FUNC";  
Properties info = new Properties();  
info.put("user", "username");  
info.put("password", "password");  
  
Connection conn = DriverManager.getConnection(URL, info);
```

## Fechando conexões JDBC

No final do seu programa JDBC, é necessário fechar explicitamente todas as conexões com o banco de dados para finalizar cada sessão do banco de dados.   
entanto, se você esquecer, o coletor de lixo do Java fechará a conexão quando limpar objetos obsoletos.

Confiar na coleta de lixo, especialmente na programação de bancos de dados, é uma prática de programação muito ruim. Você deve criar o hábito de sempre fechar a conexão com o método `close()` associado ao objeto de conexão.

Para garantir que uma conexão seja fechada, você pode fornecer um bloco 'finalmente' no seu código. Um bloco final sempre é executado, independentemente de uma exceção ocorrer ou não.

Para fechar a conexão aberta acima, você deve chamar o método `close()` da seguinte maneira:

```
conn.close();
```

O fechamento explícito de uma conexão economiza recursos do DBMS, o que tornará seu administrador de banco de dados satisfeito.

Ao programar uma aplicação Java para usar um banco de dados, o ideal é que ela seja estruturada para que funcione em opções diversas de banco de dados, afinal, é sempre do desejo da pessoa programadora atender ao maior número de casos possíveis de infraestrutura. Para que isso seja possível sem a necessidade de se fazer várias implementações diferentes, o Java fornece uma API chamada *Java Database Connectivity* (JDBC) que centraliza e formaliza os comandos de comunicação com os bancos de dados. Com os métodos disponíveis nesta API, é possível se conectar a qualquer banco de dados e fazer operações neles. O único requisito para que o JDBC funcione é encontrar um *driver* que consiga fazer a

comunicação entre o banco de dados e o Java (e todos os grandes bancos de dados do mercado possuem *drivers* JDBC).



Uma vez encontrado o JDBC e ele foi incluído no *classpath* da aplicação, é possível testar a conexão alterando os dados abaixo:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;

public class Main {

    public static void main(String[] args) {
        try {
            System.out.println("Tentando conexão com o db...");

            String url = "jdbc:oracle:thin:@root:1521:FUNC";
            Properties props = new Properties();
            props.setProperty("user", "username");
            props.setProperty("password", "password");

            Connection conn = DriverManager.getConnection(url, props);

            System.out.println("Conexão realizada com sucesso!");
        } catch (Exception e) {
            System.err.println("Erro na conexão! " + e.getMessage());
        }
    }
}
```

Uma vez conectado ao banco de dados, para executar comandos de leitura e escrita nele é necessário usar as classes que implementam a interface *Statement*.



Esta interface possui o método `execute()`, responsável por consolidar uma instrução SQL.



## DBC - Declarações, `PreparedStatement` e `CallableStatement`

Depois que uma conexão é obtida, podemos interagir com o banco de dados. As interfaces `Statement`, `PreparedStatement` e `CallableStatement` definem os métodos e propriedades que permitem enviar comandos SQL ou PL / SQL e receber dados do seu banco de dados.

Eles também definem métodos que ajudam a colmatar as diferenças de tipos de dados entre os tipos de dados Java e SQL usados em um banco de dados.

A tabela a seguir fornece um resumo do objetivo de cada interface para decidir sobre a interface a ser usada.

Interfaces	Uso recomendado
<code>Statement</code>	Use isso para acesso de uso geral ao seu banco de dados. Útil quando você está usando instruções SQL estáticas em tempo de execução. A interface <code>Statement</code> não pode aceitar parâmetros.
<code>PreparedStatement</code>	Use isso quando planejar usar as instruções SQL várias vezes. A interface <code>PreparedStatement</code> aceita parâmetros de entrada em tempo de execução.
<code>CallableStatement</code>	Use isso quando desejar acessar os procedimentos armazenados do banco de dados. A interface <code>CallableStatement</code> também pode aceitar parâmetros de entrada de tempo de execução.

## Objetos Statement



Antes de poder usar um objeto Statement para executar uma instrução SQL, é necessário criar um usando o método `createStatement()` do objeto `Connection`, como no exemplo a seguir:

```
Statement stmt = null;
try {
    stmt = conn.createStatement();
    ...
}
catch (SQLException e) {
    ...
}
finally {
    ...
}
```

Depois de criar um objeto Statement, você poderá usá-lo para executar uma instrução SQL com um de seus três métodos de execução.

### Objeto de instrução de fechamento

Assim como você fecha um objeto `Connection` para economizar recursos do banco de dados, pelo mesmo motivo, você também deve fechar o objeto `Statement`.

- `boolean execute (String SQL)`: retorna um valor booleano `true` se um objeto `ResultSet` puder ser recuperado; caso contrário, ele retornará `false`. Use este método para executar instruções DDL SQL ou quando você precisar usar SQL verdadeiramente dinâmico.
- `int executeUpdate (String SQL)`: Retorna o número de linhas afetadas pela execução da instrução SQL. Use este método para executar instruções SQL para

as quais você espera que várias linhas sejam afetadas - por exemplo, uma instrução INSERT, UPDATE ou DELETE.



- `ResultSet executeQuery (String SQL)`: Retorna um objeto `ResultSet`. Use esse método quando você espera obter um conjunto de resultados, como faria com uma instrução `SELECT`.

Uma simples chamada para o método `close ()` fará o trabalho. Se você fechar o objeto `Connection` primeiro, ele também fechará o objeto `Statement`. No entanto, você sempre deve fechar explicitamente o objeto `Statement` para garantir uma limpeza adequada.

```
Statement stmt = null;
try {
    stmt = conn.createStatement( );
    ...
}
catch (SQLException e) {
    ...
}
finally {
    stmt.close();
}
```

No exemplo abaixo, um comando de `INSERT` é feito na tabela `alunos` do banco `descomplica_db`.



```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.util.Properties;

public class Main {

    public static void main(String[] args) {

        String query = "INSERT INTO alunos (id, nome, sobrenome) VALUES (1,
'Faculdade', 'Descomplica)";
        Statement st = null;


        try {
```

```
        System.out.println("Tentando conexão com o db...");

        String url = "jdbc:oracle:thin:@root:1521:FUNC";
        Properties props = new Properties();
        props.setProperty("user", "username");
        props.setProperty("password", "password");
        Connection conn = DriverManager.getConnection(url, props);

        System.out.println("Executando o comando SQL no db...");
        st = conn.createStatement();
        st.executeUpdate(query);
        conn.close();

        System.out.println("Conexão realizada com sucesso!");
    } catch (Exception e) {
        System.err.println("Erro na conexão! " + e.getMessage());
    }
}
}
```

Um comando de leitura é muito semelhante, mas é necessário o uso da interface *ResultSet* para iterar os possíveis resultados da consulta, como mostra o  código a seguir:

```
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.ResultSet;  
import java.sql.Statement;  
import java.util.Properties;
```



```
public class Main {

    public static void main(String[] args) {

        String query = "SELECT * from alunos";
        Statement st = null;
        ResultSet rs = null;

        try {
            System.out.println("Tentando conexão com o db...");

            String url = "jdbc:oracle:thin:@root:1521:FUNC";
            Properties props = new Properties();
            props.setProperty("user", "username");
            props.setProperty("password", "password");
            Connection conn = DriverManager.getConnection(url, props);

            System.out.println("Executando o comando SQL no db...");
            st = conn.createStatement();
            rs = st.executeQuery(query);

            while(rs.next()) {
                System.out.println("\nRegistro");
                System.out.println("id: " + rs.getString(1));
                System.out.println("nome: " + rs.getString(2));
                System.out.println("sobrenome: " + rs.getString(3));
            }

            rs.close();
            st.close();
            conn.close();
        }
    }
}
```

```
} catch (Exception e) {  
    System.err.println("Erro na conexão! " + e.getMessage());  
}  
}  
}
```



## Atividade Extra

Artigo: Qual a diferença entre base de dados relacional e não relacional?

## Referência Bibliográfica

BARNES, D. J. KOLLING, M. **Programação orientada a objetos com java: uma introdução prática usando o bluej**. 4.ed. Pearson: 2009.

FELIX, R. (Org.). **Programação orientada a objetos**. Pearson: 2017.

MEDEIROS, L. F. de. **Banco de dados: princípios e prática**. Intersaberes: 2013;

ORACLE. Java Documentation, 2021. **Documentação oficial da plataforma Java**. Disponível em: <<https://docs.oracle.com/en/java/>>.

**Ir para exercício**