



# Desenvolvimento de Sistemas

---

## Conexão com banco de dados: bibliotecas e operações de conexão

### Introdução

Ao desenvolver uma aplicação que visa solucionar uma situação-problema, busca-se primeiro encontrar uma linguagem de programação para colocar em prática a solução do projeto. Para isso, utiliza-se uma IDE que propicia ao desenvolvedor criar seu código implementando funções que foram determinadas após a elaboração da documentação do projeto. Ao executar o código-fonte criado, um programa é processado, permitindo a realização de uma ou mais funções para as quais foi desenvolvido.

É possível, por exemplo, realizar um cálculo de conversão monetária, traduzir uma frase, manter atualizado o estoque de um comércio ou gerenciar a agenda de um consultório médico. Porém, o que acontece quando se encerra a execução do código-fonte? Para onde vão os dados que estavam em memória? É isso mesmo, as informações produzidas durante a execução são perdidas da memória do computador, afinal, o código foi desenvolvido apenas para executar a tarefa que foi proposta. Não foi informado nenhum local para que os dados ficassem armazenados e salvos para consulta posterior.

Assim, é necessário que o *software* tenha conexão com um banco de dados para manter o registro das movimentações realizadas. É possível determinar que os dados fiquem armazenados em um disco rígido em um servidor local ou armazenados em um servidor na nuvem. Esse processo é chamado de “persistência de dados” e significa garantir que as informações sejam armazenadas adequadamente em uma memória persistente e não temporária como a RAM (*random access memory*).

A persistência de dados pode ser feita de diversas formas, porém, as formas mais comuns são por meio de banco de dados e de arquivo (arquivo de texto, XML – *extensible markup language* etc.). A forma mais comum de se persistirem dados em aplicações é por meio de banco de dados, devido à facilidade de operações CRUD (*create, read, update, delete*) com a linguagem SQL (*structured query language*).

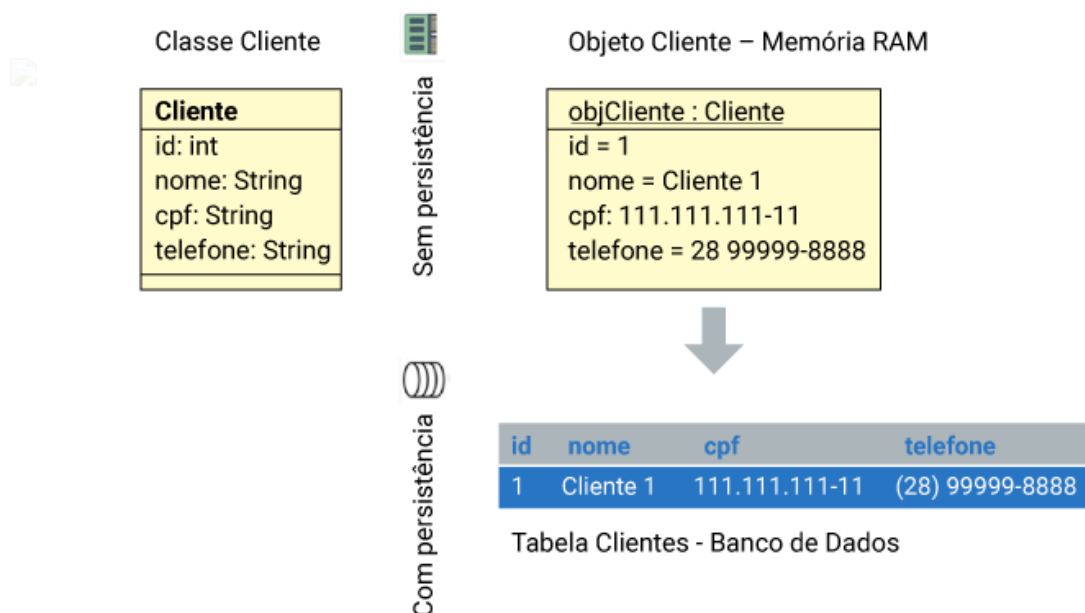


Figura 1 – Persistência de dados em sistemas

Fonte: Senac EAD (2022)

O banco de dados deve ser outra implementação do projeto, ou seja, um sistema desenvolvido em paralelo ao código-fonte criado a partir de uma linguagem de programação (neste curso é utilizada a linguagem Java), que tem peculiaridades próprias, como uma linguagem para criação e manipulação do código-fonte e da interface de *design* para criação da estrutura do banco. Portanto, é preciso escolher uma linguagem para o banco ser criado. Neste curso, trabalha-se com o MySQL, da Oracle, mas também é possível utilizar PostgreSQL, do grupo de banco de dados relacionais, ou implementar um banco NoSQL, como o MongoDB.

Neste conhecimento, serão apresentados modos de realizar a conexão com o banco de dados, sendo possível assim estabelecer comunicação por meio de bibliotecas específicas. Assim, o programa desenvolvido estabelecerá um canal de confiança para que os dados gerados durante a utilização do sistema permaneçam armazenados no banco de dados mesmo após o encerramento da aplicação.

## Comandos básicos no MySQL

Antes de iniciar a comunicação com o banco de dados, que tal relembrar alguns conceitos básicos da linguagem SQL para tornar possível a criação de um CRUD. Essa revisão também servirá para criar o banco de dados que será usado nos exemplos posteriores, por isso, recomenda-se que você execute os códigos utilizando um editor como o MySQL Workbench.

O primeiro comando a ser lembrado é o de criação do banco, o **create database**, informando o nome que a base de dados receberá. Portanto, será criada uma base para ser o exemplo:

```
create database exemplo_senac
```

Após criar o banco de dados, é preciso indicar ao MySQL Workbench qual é o banco que será utilizado, então deve-se executar o comando **use**:

```
Use exemplo_senac
```

Agora, cria-se uma tabela com o comando **create table**. O comando de criação de tabela deve conter o nome da tabela e as colunas que ela terá com o tipo dos dados que serão armazenados. Também é possível realizar configurações para definir chaves primárias, chaves estrangeiras, colunas que não devem ficar vazias, entre outros detalhes.

Neste exemplo, cria-se uma tabela chamada “usuário”, que armazenará apenas dois dados, uma identificação do tipo “serial” e um nome do tipo “varchar” (para armazenar no máximo 100 caracteres), tornando a identificação uma chave primária:

```
create table usuario(  
  id int not null auto_increment,  
  nome varchar(70),  
  PRIMARY KEY (id)  
);
```

Após criar a tabela, é necessário realizar a inserção de dados utilizando o comando **INSERT INTO**. No comando, é necessário informar o nome da tabela e os campos que serão populados. Além disso, é preciso informar a palavra reservada **VALUES** junto aos valores que serão inseridos, como neste exemplo:

```
INSERT INTO usuario (nome) VALUES ('João')
```

Perceba que o **id** não foi utilizado, pois se utilizou **auto\_increment** na coluna de chave primária, o que faz a coluna ser preenchida com valor sequencial. Um comando importante a ser recordado também é o de consulta a uma tabela do banco. Para isso, utiliza-se o **SELECT \* FROM**, informando o nome da tabela:

```
SELECT * FROM usuario
```

Ainda, é possível substituir o asterisco pelo nome de uma coluna para exibir os dados armazenados nela:

```
SELECT nome FROM usuario
```

Para atualizar os dados em um registro de uma tabela, é necessário especificar qual é o critério de seleção e, para isso, utiliza-se o **UPDATE** com nome da tabela, a palavra reservada **SET** junto aos campos com os dados para atualização e a condição por meio da palavra reservada **WHERE**. Neste exemplo, será adicionado um sobrenome ao usuário:

```
UPDATE usuario SET nome = 'João da Silva' WHERE id = 1;
```

Por fim, pode ser necessário excluir um registro do armazenamento, logo, utiliza-se o comando **DELETE FROM** com o nome da tabela e a condição para exclusão utilizando a palavra reservada **WHERE**. Agora, exclui-se o usuário de identificação 1:

```
DELETE FROM usuario WHERE id = 1;
```

Com os comandos revisados, agora é possível partir para a conexão com o banco de dados.

## Bibliotecas: ODBC e JDBC

Utilizam-se bibliotecas para realizar a conexão com diferentes tipos de bancos de dados, tornando assim o desenvolvimento menos trabalhoso. Isso porque, ao utilizar um *driver* ODBC ou JDBC, é possível realizar a conexão sem a necessidade de reestruturação do código desenvolvido, já que cada linguagem tem seus parâmetros e suas peculiaridades. Com eles, não há a preocupação de qual sistema operacional o *software* operará, pois o *driver* contém uma gama de métodos e interfaces que padroniza a comunicação.

**ODBC**, ou Open Database Connectivity, é um conjunto de classes-modelo para conexão disponível em qualquer linguagem de programação, diferindo-se da JDBC, que é exclusiva para Java.

**JDBC**, ou Java EE Database Connectivity, é uma **API (*application programming interfaces*)** que **possibilita a interação do código Java com um banco de dados**, na qual as instruções SQL são executadas por meio de um *driver*, composto por um grupo de classes e interfaces desenvolvido em Java. Esse processo possibilita a execução de instruções SQLs que se conectarão com a base de dados, permitindo a manipulação dos valores.

Os pacotes **java.sql** e **javax.sql** são disponibilizados pela JDBC, viabilizando o acesso a diferentes bases. O resultado da implementação da interface **java.sql.Driver** por meio de uma classe é o *driver* JDBC. A conexão é estabelecida por intermédio da classe **DriverManager**, a partir dos métodos padronizados por ela para o gerenciamento de uma base de dados.

## Estabelecendo conexão com o banco de dados

Cada empresa mantenedora de um SGBD (Sistema de Gerenciamento de Banco de Dados) contém um *driver* específico e geralmente o distribui para *download* em suas plataformas, possibilitando assim acesso às versões atualizadas para uso. Neste material, será utilizado o *driver* do banco MySQL, que é um arquivo .jar.

O arquivo está disponível para *download* na Internet e você pode encontrá-lo buscando por “mysql community downloads” em seu buscador de preferência.

Ao acessar a página, perceba que as principais linguagens de programação contêm seu *driver* específico para utilização. Neste momento, será utilizado o Connector/J, pois esta é a versão para o Java.

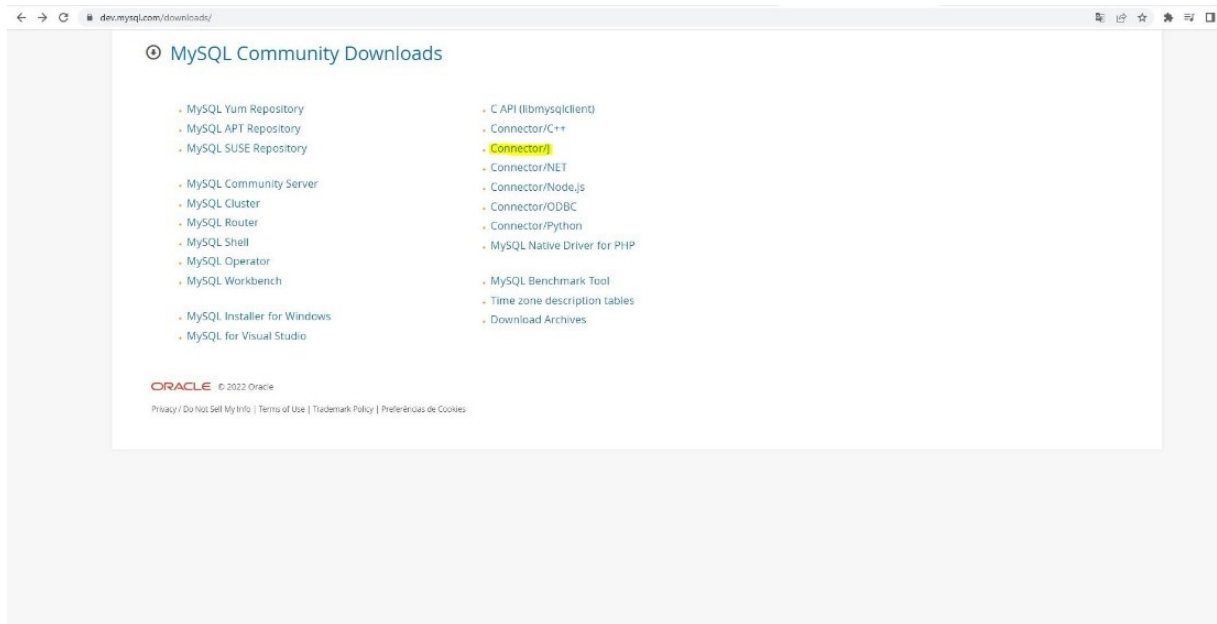


Figura 2 – Página de *download* da comunidade MySQL

Fonte: Senac EAD (2022)

Ao escolher o *driver* para Java, o redirecionamento para a página de *download* padrão é efetuado. Nessa tela, sugere-se o *download* do MySQL Installer, conforme o sistema operacional que você está utilizando, porém é possível selecionar versão para outros sistemas. Utilizar o instalador fará com que seja instalada a versão completa do MySQL, incluindo, talvez, funcionalidades já instaladas no computador e/ou desnecessárias nesse momento. Portanto, em **Select Operating System**, escolha a opção **Platform Independent**.

## MySQL Community Downloads

Connector/J

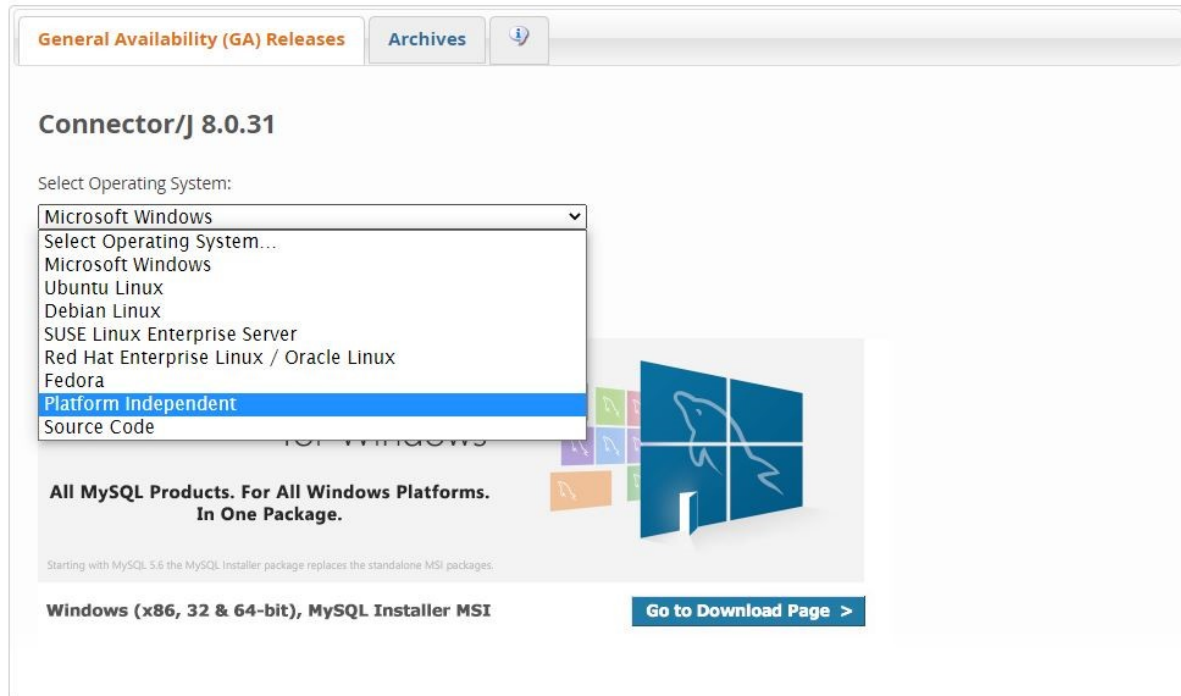


Figura 3 – Página de seleção de sistema operacional

Fonte: Senac EAD (2022)

Na próxima tela, as opções de *download* para um arquivo ZIP ou TAR são exibidas. Com isso, é possível baixar apenas o *driver* necessário para efetuar a configuração posteriormente no ambiente de desenvolvimento.

MySQL Community Downloads

Connector/J

General Availability (GA) ReleasesArchives

### Connector/J 8.0.31

Select Operating System:

Platform Independent

<b>Platform Independent (Architecture Independent), Compressed TAR Archive</b> (mysql-connector-j-8.0.31.tar.gz)	8.0.31	4.1M	<a href="#">Download</a>
MD5: fcef1e060585bf70659c87436bd1722c   <a href="#">Signature</a>			
<b>Platform Independent (Architecture Independent), ZIP Archive</b> (mysql-connector-j-8.0.31.zip)	8.0.31	4.9M	<a href="#">Download</a>
MD5: 3ef8aabde9bc87fbd9a260155341074d   <a href="#">Signature</a>			

We suggest that you use the MD5 checksums and GnuPG signatures to verify the integrity of the packages you download.

Figura 4 – Tela para escolha do arquivo JAR/TAR

Fonte: Senac EAD (2022)

Depois de selecionar o pacote necessário para o sistema operacional, a Oracle sugerirá a realização de login na plataforma, mas é possível seguir apenas com a opção de baixar sem efetuar login (link **No thanks, just start my download.**).



## MySQL Community Downloads

Login Now or Sign Up for a free account.

An Oracle Web Account provides you with the following advantages:

- Fast access to MySQL software downloads
- Download technical White Papers and Presentations
- Post messages in the MySQL Discussion Forums
- Report and track bugs in the MySQL bug system

**Login »**  
using my Oracle Web account

**Sign Up »**  
for an Oracle Web account

MySQL.com is using Oracle SSO for authentication. If you already have an Oracle Web account, click the Login link. Otherwise, you can signup for a free account by clicking the Sign Up link and following the instructions.

[No thanks, just start my download.](#)

Figura 5 – Tela para *login*

Fonte: Senac EAD (2022)

Quando a pasta compactada estiver no computador, será necessário extrair os arquivos dela. Depois disso, alguns arquivos e pastas serão encontrados na raiz, mas apenas o **mysql-connector-j8.0.31.jar** será utilizado para adicioná-lo no projeto de desenvolvimento.

Nome	Data de modificação	Tipo	Tamanho
src	03/09/2022 21:54	Pasta de arquivos	
build.xml	03/09/2022 21:54	Documento XML	93 KB
CHANGES	03/09/2022 21:54	Arquivo	271 KB
INFO_BIN	03/09/2022 21:54	Arquivo	1 KB
INFO_SRC	03/09/2022 21:54	Arquivo	1 KB
LICENSE	03/09/2022 21:54	Arquivo	70 KB
mysql-connector-j-8.0.31.jar	03/09/2022 21:54	Executable Jar File	2.457 KB
README	03/09/2022 21:54	Arquivo	2 KB

Figura 6 – **MySQL-Connector-j8.0.31.jar** descompactado

Fonte: Senac EAD (2022)

O próximo passo é abrir o NetBeans Apache 14, escolher a opção **File** e, em seguida, selecionar a opção **New Project**.

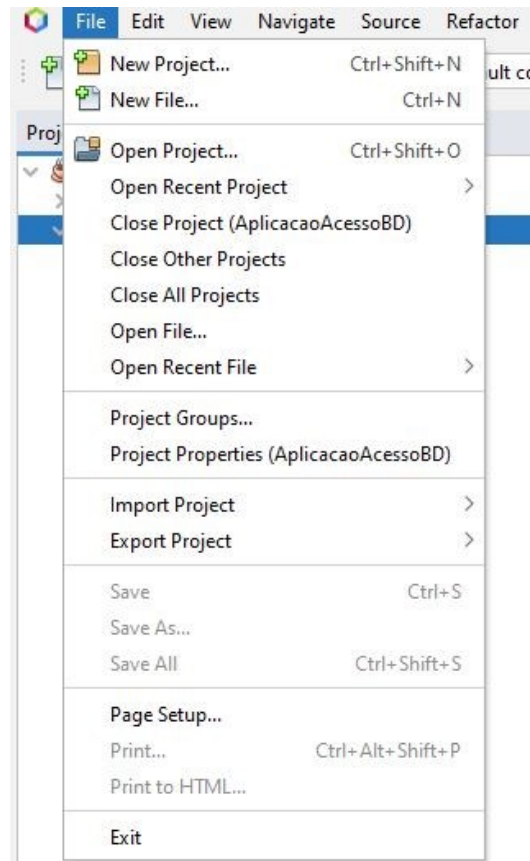
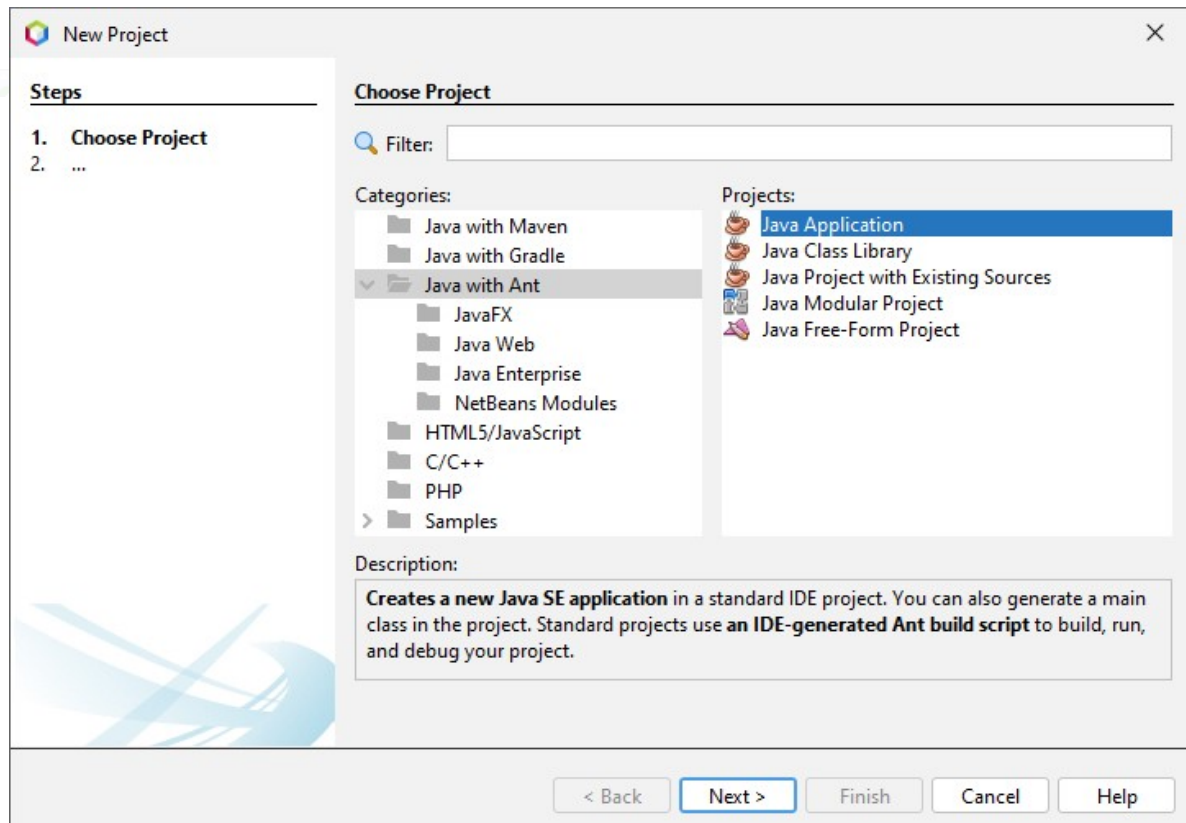


Figura 7 – Tela para criar novo projeto NetBeans

Fonte: Senac EAD (2022)

Agora, para realizar a configuração do projeto, escolha a opção **Java with Ant** e depois a opção **Java Application**. Clique em **Next**, nomeie o projeto de **AplicacaoAcessoBd** e nomeie a classe principal de “conexão”.

Figura 8 – Configuração do projeto **Java with Ant**

Fonte: Senac EAD (2022)

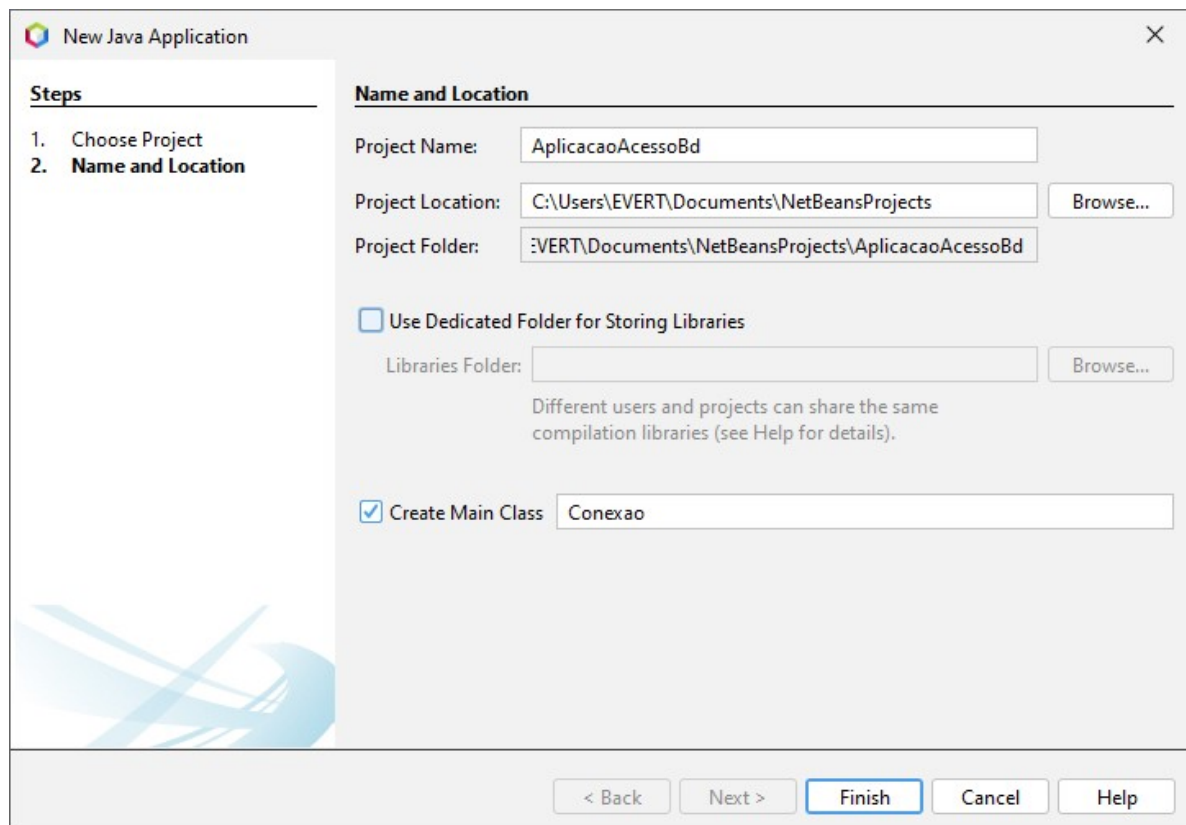


Figura 9 – Segunda tela de configuração **Java with Ant**

Fonte: Senac EAD (2022)

Agora, copie o arquivo **mysql-connector-j8.0.31.jar**, descompactado anteriormente, e cole-o na pasta raiz do projeto recém-criado.

Para que seja possível continuar a classe de conexão com a base de dados, é necessário adicionar o *driver* MySQL ao projeto no NetBeans, portanto, em **Libraries**, no pacote do projeto, clique com o botão direito e selecione a opção **Add JAR/Folder**.

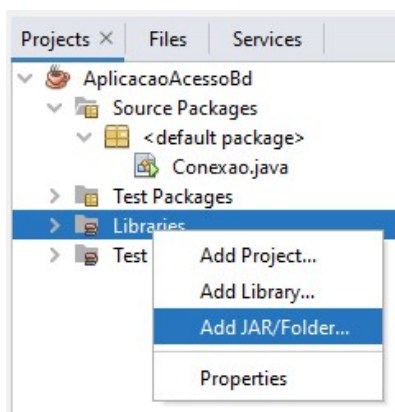


Figura 10 – Tela de bibliotecas do projeto

Fonte: Senac EAD (2022)

Uma janela de seleção de pasta será exibida na tela. Busque a pasta raiz de seu projeto e selecione o *driver* que foi descompactado anteriormente, escolhendo-o para adicionar ao projeto. Observe a figura a seguir e depois o GIF do autor.

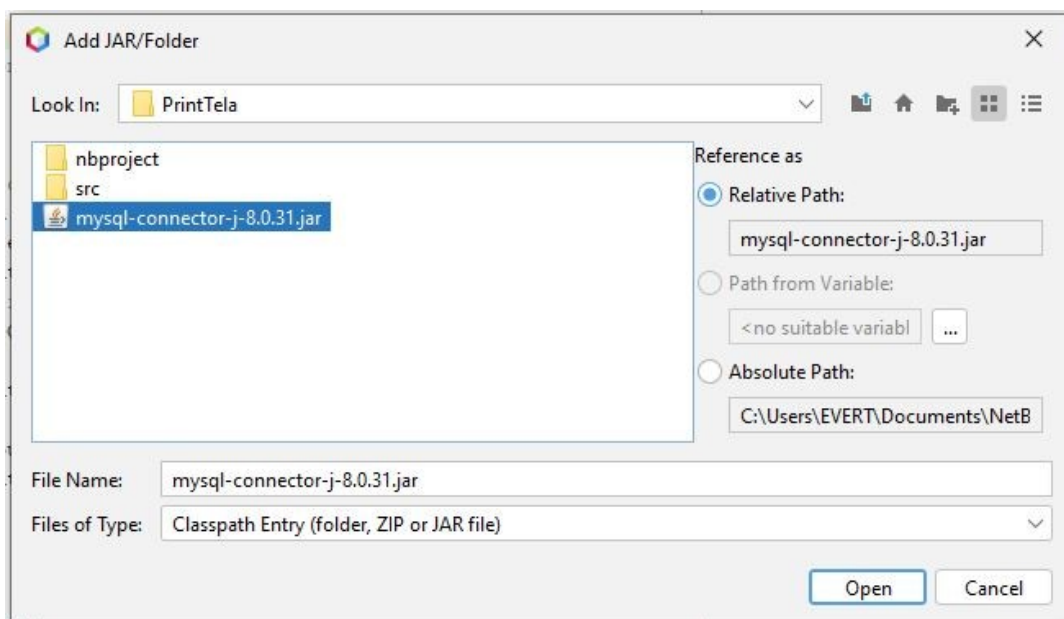
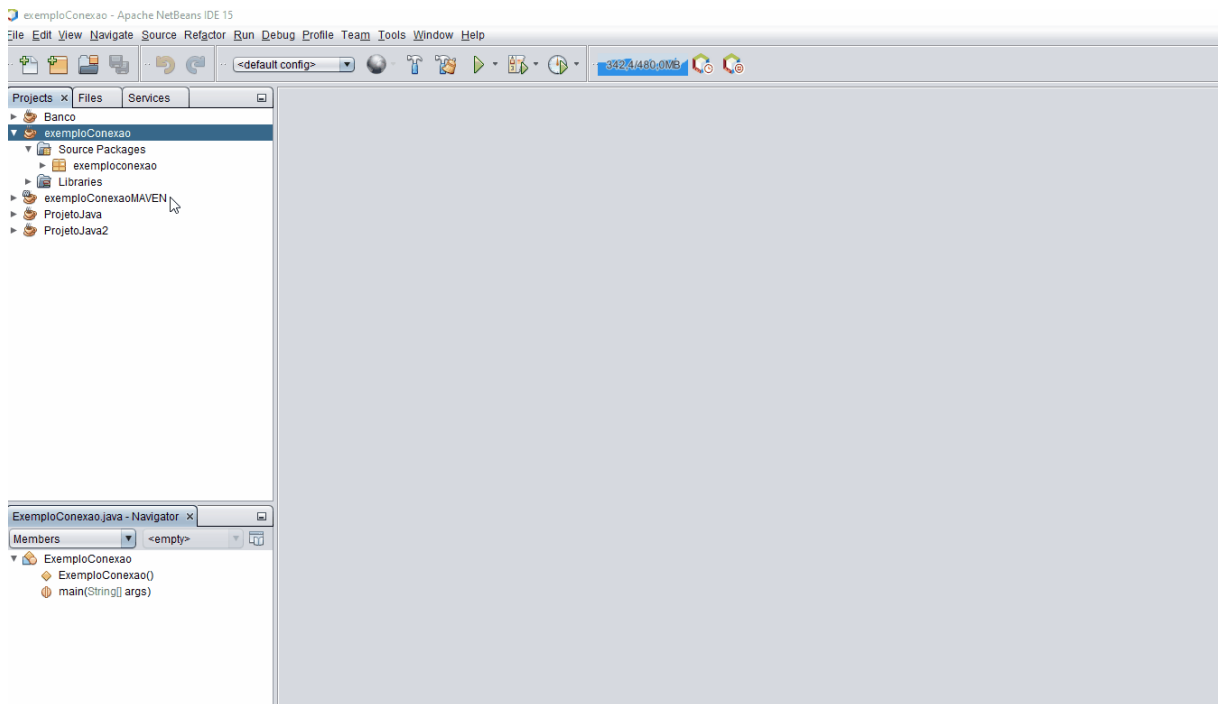


Figura 11 – Pasta do *driver* MySQL-Connector

Fonte: Senac EAD (2022)



Incluída a biblioteca, agora é o momento de implementar o método **main()** da classe principal para testar a conexão com o banco de dados. É importante incluir manualmente as cláusulas de “import”, pois elas podem não ser reconhecidas automaticamente pelo NetBeans.

```
package aplicacaoacessobd;

import java.sql.Connection;
import java.sql.Statement;

public class AplicacaoAcessoBd {

    public static void main(String[] args) {
        try {
            Connection conn; //criando um objeto do tipo connection chamado conn
            Statement st; //criando um objeto do tipo Statement chamado st para execução de comando
            SQL
            Class.forName("com.mysql.cj.jdbc.Driver");
        } catch (ClassNotFoundException ex) {
            System.out.println("driver não está disponível para acesso ou não existe");
        }
    }
}
```

No código, primeiro é preciso declarar um objeto do tipo **Connection**, que será necessário para manter a conexão com o banco. O objeto do tipo **Statement** será usado como comando para o banco de dados. Atente-se, porém, à linha **Class.forName("com.mysql.cj.jdbc.Driver")**. Ela é usada para carregar o *driver* que conseguirá se comunicar com o banco de dados MySQL. Essa classe *driver* fica dentro do pacote **com.mysql.cj.jdbc** e você pode visualizá-la expandindo a biblioteca MySQL-Conector e verificando-a nos pacotes exibidos.

Perceba que, além da linha que contém o método **forName()**, foi adicionado ainda um bloco *try-catch*. No exemplo, pode ser que a classe *driver* não exista e, dessa forma, interrompa-se a execução do programa. Nesse caso, uma mensagem é exibida no terminal console informando que o *driver* não está disponível para acesso ou não existe. Como padrão, a interface de desenvolvimento pode sugerir que um bloco precisa de um tratamento, criando assim uma mensagem de erro genérica. Porém, para evitar que o usuário se depare com um texto técnico, substitui-se esse texto por uma mensagem **agradável**. Também como padrão, será adicionada a importação das bibliotecas.

```
import java.util.logging.Level;  
import java.util.logging.Logger;
```

Porém, por não serem utilizadas neste código de exemplo, as bibliotecas podem ser excluídas, caso sejam adicionadas como padrão.

Voltando ao código, apenas foi indicada a classe da biblioteca do *driver* que fará a conexão do Java com a base de dados MySQL. Logo, é preciso adicionar o comando que estabelecerá a ligação com o banco de dados. Isso ocorre pelo método **getConnection()** da classe **DriverManager** (necessário **import java.sql.DriverManager**).

```

package aplicacaoacessobd;

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

public class AplicacaoAcessoBd {

    public static void main(String[] args) {
        try {
            Connection conn; //criando um objeto do tipo connection chamado conn
            Statement st; //criando um objeto do tipo Statement chamado st para
            execução de comando SQL
            Class.forName("com.mysql.cj.jdbc.Driver");
            DriverManager.getConnection("jdbc:mysql://localhost:3306/exemplo_senac",
            "root","abcd1234");
        } catch (ClassNotFoundException ex) {
            System.out.println("O Driver não está disponível na biblioteca");
        } catch (SQLException ex) {
            System.out.println("Sintaxe de comando invalida ");
        }
    }
}

```

Atente-se à linha  
**DriverManager.getConnection("jdbc:mysql://localhost:3306/exemplo\_senac","root","abcd1234");**.

O comando **DriverManager.getConnection()** recebe como padrão três parâmetros. O primeiro parâmetro deve indicar o **nome do banco de dados** que será utilizado para estabelecer a conexão junto à porta que é utilizada – no caso do MySQL, a porta padrão é a 3306. Além de informar o número da porta, você também deve informar o local em que o MySQL está sendo executado. Neste exemplo, será utilizado o **localhost**, ou seja, a execução do MySQL ocorre em sua própria máquina; caso estivesse sendo executado em um servidor que opera na rede, seria necessário informar o endereço IP do equipamento.

O segundo parâmetro é o **usuário**. Como padrão, o MySQL instala o usuário “root”. O terceiro parâmetro é a **senha** que foi configurada na instalação do MySQL para o usuário “root”. Caso o usuário tenha sido criado sem senha, não é necessário informar. Esses parâmetros devem estar entre aspas duplas e, caso não exista senha, basta você apenas abrir e fechar aspas vazias.

Está sendo utilizada a base de dados criada na revisão de comandos SQL: “**exemplo\_senac**”. Para utilizar o comando **DriverManager**, importe a classe **java.sql.DriverManager**. Ainda será necessário tratar uma exceção do tipo **SQLException**, pois o banco pode talvez não existir. Portanto, é preciso adicionar um novo *catch* para exibir uma mensagem informando que ocorreu uma falha de conexão com a base de dados.

Anteriormente, foi criada uma variável para armazenar a conexão e ela ainda não está sendo utilizada (**conn**). Por isso, faça uma alteração na linha que contém o método de conexão, também chamada de *string* de conexão, pois ela tem os parâmetros de acesso ao banco de dados. A variável **conn** passa a receber *string* de conexão e, para definir que o objeto **Statement st** receberá a conexão criada, utiliza-se o método **createStatement()**, responsável por estabelecer a conexão ao gerar a comunicação com a base de dados por meio da instrução SQL presente no **conn**, ficando o código do seguinte modo:

```
package aplicacaoacessobd;

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

public class AplicacaoAcessoBd {

    public static void main(String[] args) {
        try {
            Connection conn; //criando um objeto do tipo connection chamado conn
            Statement st; //criando um objeto do tipo Statement chamado st para
            execução de comando SQL
            Class.forName("com.mysql.cj.jdbc.Driver");
            conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/exemplo_senac",
            "root","abcd1234");
            st = conn.createStatement();
        } catch (ClassNotFoundException ex) {
            System.out.println("O Driver não está disponível na biblioteca");
        } catch (SQLException ex) {
            System.out.println("Sintaxe de comando invalida");
        }
    }
}
```

Agora, seu código já tem uma conexão estabelecida com a base de dados e você pode testá-la rodando a aplicação – o resultado esperado é a ausência de mensagens de erro. Com a conexão estabelecida, será possível realizar inserções, consultas, atualizações e exclusões por meio de métodos que ainda serão apresentados no decorrer deste conhecimento. Com relação aos métodos, o ideal seria que este código estivesse em um método de uma classe específica, assim, sempre fosse necessário um acesso ao banco, bastaria chamar o método de conexão. Perceba que foi criada uma conexão em uma classe **main()**, o que não é o mais adequado. Por isso, você verá um novo exemplo, mas antes, que tal um desafio?

Crie uma nova base de dados chamada “Desafio no MySQL Workbench”.

Crie um novo projeto e uma nova classe de conexão chamada “ConectarDesafio”, mas **sem** que esta seja a classe principal (**main**), e adicione a biblioteca de conexão ao MySQL.



Agora, estabeleça a conexão do seu projeto com a base de dados criada.



Crie no pacote principal do projeto uma nova classe chamada **Conexao** e comece a implementação declarando o atributo que conterá as informações de conexão (Connection) e o que estabelecerá essa conexão (não esqueça de efetuar os *imports* necessários). Esse atributo público poderá ser usado por outras classes no sistema para utilizarem a conexão estabelecida.

Além disso, deixe pronta a assinatura do método que efetuará a conexão quando chamado por alguma outra classe. Cria-se esse método com um retorno booleano, para que, quando a conexão ocorrer, seja exibida uma mensagem de conexão com sucesso, ou seja, um retorno *true*; e quando ocorrer algum problema, uma mensagem de falha seja exibida, com um retorno *false*.

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

public class Conexao {

    Connection conn; //criando um objeto do tipo connection chamado conn

    public boolean conectar(){

    }

}
```

Agora, dentro do método **conectar()**, defina qual *driver* será utilizado por meio do **Class.forName**, em novo caso, o *driver* adicionado à biblioteca MySQL, defina a *string* de conexão, atribua a variável **conn** e estabeleça a conexão. Para isso, utilize o banco de dados criado como exemplo, o “**exemplo\_senac**”, mantendo as informações de usuário, a senha e a porta de conexão.

Não se esqueça de que é necessária a criação de *try-catches*, conforme o exemplo anterior, pois exceções podem ser lançadas devido a um comportamento inesperado do sistema.

```
public boolean conectar(){  
    try {  
        Class.forName("com.mysql.cj.jdbc.Driver");  
        conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/exemplo_senac","root","a  
bcd1234");  
        System.out.println("Conexão realizada com sucesso");  
        return true;  
    } catch (ClassNotFoundException | SQLException ex) {  
        System.out.println("Falha na conexão com o banco " + ex.getMessage());  
        return false;  
    }  
}
```

São necessários **import java.sql.DriverManager** e **import java.sql.SQLException**.

Perceba que, nesse exemplo, realizou-se o que é chamado de *multicatch*, isto é, mais de uma exceção é tratada ao mesmo tempo. Outro detalhe que deve ser verificado é o *return true* e o **System.out.println** com a mensagem de conexão bem sucedida no bloco *try* e, no *catch*, o *return false* com o **System.out.println** e uma mensagem de falha na conexão.

Para testar se o método está funcionando, altere o método **main** da classe principal e declare um objeto do tipo conexão chamado “conector”, inicializando com um construtor vazio. Na sequência, chame o método **conectar** para efetivar a conexão com a base de dados e execute para verificar o resultado da conexão no console. Se você obtiver êxito, a mensagem de sucesso será exibida na tela; caso contrário, haverá a mensagem de falha.

```
public class AplicacaoAcessoBd {  
    public static void main(String[] args){  
        Conexao conector = new Conexao();  
  
        conector.conectar();  
    }  
}
```

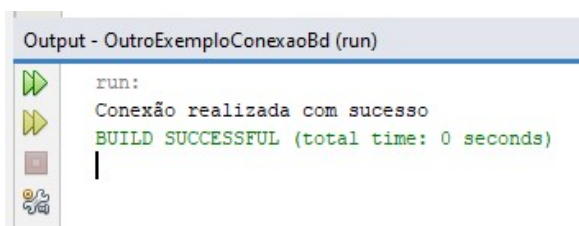


Figura 12 – Conexão realizada com sucesso

Fonte: Senac EAD (2022)

Contudo, as boas práticas de programação determinam que, ao estabelecer uma conexão com uma base de dados, após efetuar as transações esperadas, a conexão seja interrompida, ou seja, fechada, encerrando assim o período em que a porta está aberta.

Para isso, na classe **Conexao**, crie um método chamado **desconectar**, do tipo *void*, pois ele somente fechará o acesso à base. Nesse método, será utilizado um dos principais métodos da classe **Connection**, que é o **.close()**, que se responsabiliza por encerrar a ligação. Não esqueça do *try-catch* para cuidar das exceções que possam ocorrer, pois talvez o método seja chamado para encerrar uma conexão que não existe ou que não está aberta.

```
public void desconectar(){
    try {
        conn.close();
    } catch (SQLException ex) {

    }
}
```

Perceba que o *catch* desse método não contém nenhuma mensagem. Como a conexão está sendo fechada, se tudo ocorrer bem, ela será encerrada. Perceba que não há nenhum tipo de retorno, logo, se não existir uma conexão, não será adequado deixar a mensagem de erro padronizada do Java, evitando-se assim a possibilidade de a mensagem ser utilizada para encontrar vulnerabilidades no sistema.

Considerando deixar o projeto com menor vulnerabilidade a ataques e dentro das boas práticas de programação, pode-se também realizar modificações na *string* de conexão. É possível criar uma variável para atribuir o caminho do servidor e o nome do banco, uma para o usuário e outra para a senha, possibilitando assim que essas variáveis sejam concatenadas, formando a *string* de conexão. Ao longo desta unidade curricular serão abordadas mais vulnerabilidades e também como preveni-las.

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

public class Conexao {

    Connection conn; //criando um objeto do tipo connection chamado conn

    public String url = "jdbc:mysql://localhost:3306/exemplo_senac"; //Nome da base de dados
    public String user = "root"; //nome do usuário do MySQL
    public String password = "abcd1234"; //senha do MySQL

    public boolean conectar(){

        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            conn = DriverManager.getConnection(url, user, password);
            System.out.println("Conexão realizada com sucesso");
            return true;
        } catch (ClassNotFoundException | SQLException ex) {
            System.out.println("Falha na conexão com o banco "+ ex.getMessage());
            return false;
        }
    }
}
```

Perceba que foram criadas três variáveis, que indicam o caminho de acesso ao banco, o nome do usuário e a senha de acesso, quando esta existir. No método **conectar**, a linha que estabelece a conexão com o **DriverManager** sofreu a alteração necessária: ao invés de receber entre aspas o caminho, o nome e o usuário, agora informam-se a variável **url**, que contém o caminho até a base de dados, a variável **user**, que contém o nome do usuário, e a variável **password**, que contém a senha do usuário (lembrando que este é um item opcional e que depende da configuração de usuário criada na base de dados).

Agora, será realizada uma consulta no banco de dados para verificar a quantidade de registros existente na única tabela definida. Portanto, insira ao menos três registros na base **exemplo\_senac**, na tabela "usuario". Observe como ficará o código da classe **Conexao**:

```

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

public class Conexao {

    public Connection conn;
    public String url = "jdbc:mysql://localhost:3306/exemplo_senac"; //Nome da base de dados
    public String user = "root"; //nome do usuário do MySQL
    public String password = "abcd1234"; //senha do MySQL

    public boolean conectar(){

        try {

            Class.forName("com.mysql.cj.jdbc.Driver");
            conn = DriverManager.getConnection(url, user, password);
            System.out.println("Conexão realizada com sucesso");

            Statement st = conn.createStatement();
            ResultSet rs = st.executeQuery("SELECT COUNT(*) FROM usuario");
//variavel da classe ResultSet para receber o valor da consulta
            rs.next();
            System.out.println(rs.getInt("COUNT(*)"));
            return true;
        } catch (ClassNotFoundException | SQLException ex) {
            System.out.println("Falha na conexão com o banco" + " " + ex.getMessage());
            return false;
        }
    }

    public void desconectar(){
        try {
            conn.close();
        } catch (SQLException ex) {
            //posso deixar vazio para evitar uma mensagem de erro desnecessária ao usuário
        }
    }
}

```

É preciso definir primeiro um **Statement**, classe responsável por métodos de consulta em banco de dados. A referência “**st**” recebe um novo objeto **Statement** criado a partir da conexão, pelo método **createStatement()**. Na linha de baixo, consta a definição de “**rs**”, que será do tipo **ResultSet** e será responsável por navegar entre os registros recuperados do banco de dados (ou seja, é a partir de **ResultSet** que serão obtidos os valores que vieram da base de dados). Necessário **import java.sql.ResultSet**.

O objeto **ResultSet** “**rs**” receberá o retorno da consulta realizada por meio do objeto “**st**”, com o método **executeQuery()**. Por parâmetro, o método **executeQuery** deve receber a *string* de consulta, que será “SELECT COUNT(\*) FROM usuário”.

Para obter o resultado da consulta e usar esse dado na aplicação, você precisará navegar até a primeira entrada utilizando a variável **rs** com o método **next()**. Por fim, o resultado da consulta será exibido no console por meio do método **getInt()** da variável **rs**, com o parâmetro **COUNT(\*)**.

Você pode testar sua aplicação e, caso apareça na aba *output* a frase “Conexão realizada com sucesso” e o número “1”, tudo estará correto.

Observe que o método **conectar** agora tem duas responsabilidades: realizar a conexão e realizar uma busca. Idealmente, a busca não deve ficar nesse método. Assim, apenas como exercício da classe **Conexao**, transporte a consulta para o método **main()**. Primeiro, retire o código de criação de **Statement** e de consulta de **conectar()**:

```
public boolean conectar(){  
  
    try {  
  
        Class.forName("com.mysql.cj.jdbc.Driver");  
        conn = DriverManager.getConnection(url, user, password);  
        System.out.println("Conexão realizada com sucesso");  
  
        return true;  
    } catch (ClassNotFoundException | SQLException ex) {  
        System.out.println("Falha na conexão com o banco" + " " + ex.getMessage());  
        return false;  
    }  
}
```

Em seguida, mova o código específico de consulta para o método **main()** da classe principal do projeto:

```
import java.sql.Statement;
import java.sql.SQLException;
import java.sql.ResultSet;
import java.util.logging.Level;
import java.util.logging.Logger;

public class AplicacaoAcessoBd {

    public static void main(String[] args) {
        try {
            Conexao conector = new Conexao();
            conector.conectar();

            Statement st = conector.conn.createStatement();
            ResultSet rs = st.executeQuery("SELECT COUNT(*) FROM usuario"); //variavel da classe Res
ultSet para receber o valor da consulta
            rs.next();
            System.out.println(rs.getInt("COUNT(*)"));
        } catch (SQLException ex) {
            Logger.getLogger(AplicacaoAcessoBd.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

Com isso, haverá melhor separação de responsabilidades e a classe **Conexao** poderá ser reutilizada por outras classes do projeto tranquilamente.

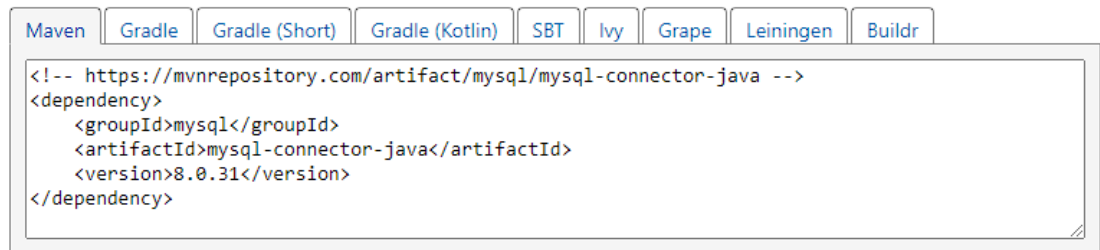
## Aplicação Java Maven usando *driver* JDBC

Uma alternativa ao projeto Java Ant é o uso da ferramenta Maven, que é capaz de controlar as dependências de um projeto sem incluir manualmente arquivos JAR, como foi feito no projeto anterior. Essa alternativa será testada a seguir (e você poderá optar pela abordagem que for mais conveniente em seus próximos projetos).

No NetBeans, crie um projeto Java Maven, que será chamado de “**exemploConexao**”.

Depois de criado o projeto, é preciso adicionar uma dependência do *driver* no projeto Java, o que é feito da seguinte forma:

- ◆ Pesquise “maven mysql driver” no Google ou acesse o *site* MVN Repository e busque “mysql conector java”.
- ◆ No *site* estarão disponíveis todas as versões do *driver*. Opte sempre pela última versão.
- ◆ Clicando na última versão do *driver* de conexão JDBC, você terá os códigos do *driver* JDBC. Agora copie o conteúdo representado na imagem a seguir e cole-o no projeto.



```

Maven  Gradle  Gradle (Short)  Gradle (Kotlin)  SBT  Ivy  Grape  Leiningen  Buildr

<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.31</version>
</dependency>

```

Figura 13 – Trecho de código para inclusão do *driver* JDBC no projeto Java Maven  
Fonte: MvnRepository (c2006-2022)

- ◆ Volte ao NetBeans, abra a árvore do projeto em **Project File** e clique no arquivo “pom.xml”.

É preciso criar o local para adicionar essa dependência. Então, em **<packaging>jar</packaging>**, crie **<dependencies> </dependencies>** e cole o código copiado do *site* do Maven, e o resultado será semelhante ao da imagem a seguir:



```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany</groupId>
  <artifactId>exemploConexaoMAVEN</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <dependencies>
    <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.30</version>
    </dependency>
  </dependencies>

```

Figura 14 – Arquivo “pom.xml” com o trecho de código copiado do *site* MVN Repository  
Fonte: NetBeans (2022)

Essas linhas servem para adicionar a biblioteca do conector JDBC.

Agora, é possível implementar uma conexão no método **main()** da classe principal do projeto, conforme a seguir:



```
Connection conexao = null;
try {
    Class.forName("com.mysql.cj.jdbc.Driver");
    conexao = DriverManager.getConnection("jdbc:mysql://localhost/banco","root","");
    System.out.println("Conexão estabelecida com o MySQL e com o banco de dados!");
} catch (ClassNotFoundException ex) {
    System.out.println("Driver do banco de dados não localizado!");
} catch (SQLException ex) {
    System.out.println("Ocorreu um erro ao acessar o banco: " + ex.getMessage());
} finally {
    if (conexao != null) {
        conexao.close();
    }
}
```

O procedimento de conexão é semelhante ao implementado no projeto Java Ant.

Execute o projeto – por conta de o Maven resolver as dependências na construção do aplicativo, as primeiras compilações podem demorar um pouco, pois será feito *download* dos arquivos das bibliotecas referenciadas (neste caso, da MySQL Connector J). A aplicação rodará com sucesso se a mensagem “**Conexão estabelecida com o MySQL e com o banco de dados!**” surgir no console.

## Principais causas de problemas de conexão

Ao tentar estabelecer uma conexão entre o banco de dados e o código-fonte criado em Java, alguns problemas pontuais podem ocorrer. Tenha cuidado ao definir o nome da base de dados para referenciá-la corretamente no caminho de conexão.

Atente-se também à porta de conexão utilizada: caso a porta de acesso seja alterada na configuração de instalação do MySQL, quando for criada a *string* de conexão no Java, a nova porta deve ser configurada.

Lembre-se de que a porta padrão do MySQL é a 3306.

Também é importante ter cuidado ao informar o nome do usuário. Como padrão, geralmente utiliza-se o *root*, que contém todos os privilégios de acesso. Durante a instalação do MySQL, esse usuário pode ser criado sem ter uma senha habilitada para ele. Portanto, tenha cuidado ao preencher essa informação na *string* de conexão, pois é preciso informar a mesma senha configurada durante a configuração da interface de banco de dados.

Lembre-se sempre de tratar as exceções, pois, como mencionado anteriormente, uma *exception* que não tem um tratamento pode interromper a operação do projeto. Além disso, mantenha as importações de classes externas que serão utilizadas sempre atualizadas e lembre-se de adicionar o *driver* ao projeto para não provocar falhas de execução, portanto mantenha a biblioteca JDBC junto à pasta do projeto.

## Encerramento

Neste material, você revisou conceitos básicos de banco de dados e aprendeu conceitos de conexão de um projeto Java com uma base dados, utilizando o SGBD MySQL. Você executou passo a passo a criação do banco e dos métodos em Java. A partir desse ponto, você está apto a exercitar, criando uma integração entre a base de dados e o projeto Java.