INFORMÁTICA PARA INTERNET

Lógica de programação aplicada a projetos C# com banco de dados

Nossos estudos partiram da lógica de algoritmos, passando pelo aprendizado na linguagem C#, até a criação de aplicações *web* utilizando o ASP.NET Core MVC.

Nosso próximo passo, agora que estamos aprendendo banco de dados, é integrar uma aplicação *web* a um banco de dados.

Vamos olhar para esses temas futuros, com confiança em nosso aprendizado até aqui. Por isso, traremos como uma revisão alguns dos principais conceitos da lógica de programação, desta vez aplicados a projetos *web* com banco de dados.

Preparando os estudos

Para os exemplos e os tutoriais deste material, usaremos o VS Code (Visual Studio Code. É necessário ter as extensões C# e C# Extensions. Os comandos principais de terminal são:

- **dotnet new mvc --no-https** (para criar um novo projeto MVC)
- **dotnet build** (para compilar o projeto)
- **dotnet run** (para executar o projeto)

Também precisaremos de um banco de dados MySQL ou MariaDB, que pode ser obtido por meio da instalação de pacotes como o XAMPP, WAMP ou EasyPHP, ou diretamente pelo instalador do servidor MySQL Community.

Ferramentas de *design* e manipulação de banco de dados, como o MySQL Workbench, podem ser necessários.

Integrando banco de dados a um projeto *web*

Agora temos conhecimento para criar um banco de dados e também para criar uma aplicação *web* utilizando ASP.NET Core MVC.

Uma tecnologia não é completa sem a outra. Nesse sentido, é muito mais interessante desenvolver um sistema que cadastre, busque e altere dados em um banco do que um que não use dados ou utilize paliativos, como listas ou arquivos. Além disso, também é muito mais usual utilizar um aplicativo para facilitar a manipulação de informações do que precisar usar o tempo todo comandos SQL ou, ainda, precisar que o usuário construa suas próprias consultas (na verdade, essa última hipótese é inadmissível).

Entre as vantagens de usar um banco de dados em uma aplicação de Internet, podemos destacar as seguintes:

Escalabilidade

Aplicações *web* são de natureza escaláveis, ou seja, podem ser incrementadas e adaptadas constantemente. Sem uso de banco de dados, que manejam milhões de conexões simultâneas e recuperam informações rapidamente, essa característica seria prejudicada. Por isso, redes sociais como o Facebook e o Twitter investem em sistemas de banco de dados poderosos e flexíveis.

Acessibilidade

As informações poderão ser acessadas de praticamente qualquer aparelho que possa acessar a Internet.

Segurança

A evolução dos sistemas *web* também impulsiona a evolução da segurança nos bancos de dados. Hoje, pode-se contar com recursos robustos de proteção a dados nos principais sistemas de gestão de base de dados (SGBDs).

Para conseguir interagir com o código do sistema, independentemente de sua natureza (*web* ou *desktop*), com o banco de dados é necessário implementar uma conexão entre essas duas tecnologias, de maneira que, dinamicamente e via código, seja possível acessar um banco de dados e enviar comandos; o banco de dados, por sua vez, responde com o resultado do comando enviado.

O banco de dados pode estar na própria máquina em que o sistema roda ou em outro computador remoto.

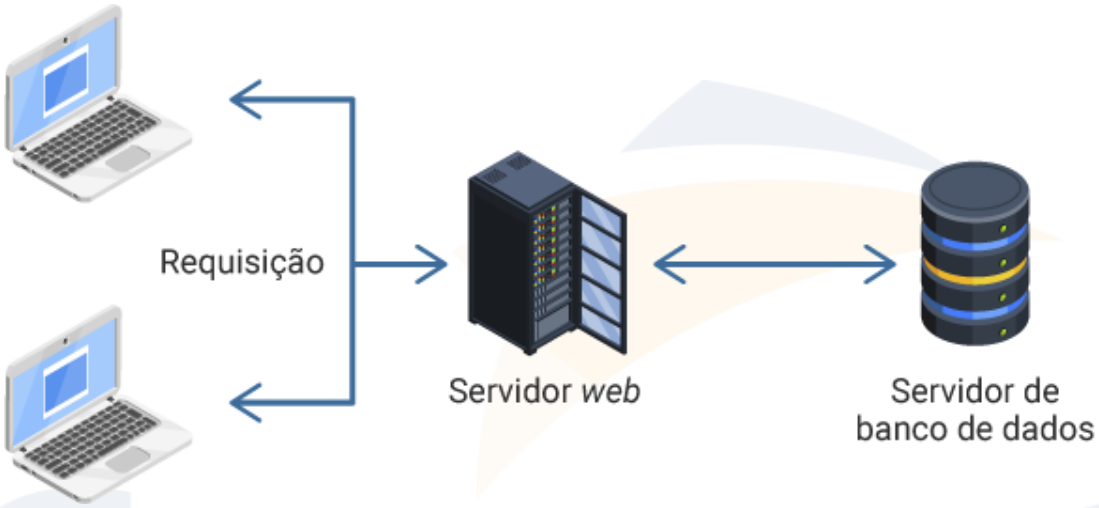


Figura 1 – Arquitetura cliente e servidor com banco de dados, um recurso próprio da camada de *back-end*

Dois computadores de cliente acessam um servidor que, por sua vez, está conectado a um banco de dados.

Trata-se de uma funcionalidade intrinsecamente de *back-end* e que depende tanto da tecnologia usada para a programação do sistema *web* quanto do SGBD utilizado. De maneira geral, o SGBD fornece **bibliotecas** específicas para as diferentes linguagens que ele suporta e que podem ser incluídas no projeto para obter a conexão.

No caso do ASP.NET Core e do MySQL, essa biblioteca se chama **MySqlConnector** e está disponível tanto no *site* do MySQL quanto no repositório de código NuGet da Microsoft.

O NuGet é uma ferramenta muito poderosa, que consiste em um grande repositório de bibliotecas plugáveis em projetos .NET, criada em 2010 para as ferramentas de desenvolvimento da Microsoft. Uma das vantagens de utilizar esta ferramenta é a facilidade para se obter e atualizar uma biblioteca, sendo rapidamente baixada no repositório na Internet, incluída e configurada para funcionar no projeto. Sua popularidade fez com que ela se tornasse parte integrante do VS Code a partir de 2012. O .NET Core tem integração nativa com o NuGet.

A biblioteca é constituída de algumas classes que auxiliam na manipulação do banco de dados, entre elas a classe **MySqlConnection** (para realizar a conexão com um banco de dados) e a **MySqlCommand** (responsável por enviar comandos ao banco e recuperar seus resultados). De maneira análoga, há bibliotecas que conectem a solução .NET Core a bancos de dados como PostGre, Oracle e SQLServer (este último, por ser da Microsoft, garante uma série de funcionalidades próprias).

Agora, vamos ver alguns dos passos comuns na conexão com banco de dados:

Conectar ao banco de dados

Para conexão, devemos informar o endereço do banco de dados (geralmente, um IP – lembre-se de que o servidor de banco de dados pode estar na própria máquina do servidor *web* ou em outro computador) e o nome do banco de dados que se quer acessar (os SGBDs permitem a criação de múltiplos bancos em um mesmo servidor). Também são necessárias credenciais, como *log in* e senha do banco de dados.

Recuperar informações do banco de dados

Uma vez que a conexão está estabelecida, pode-se realizar a busca e a recuperação de informações do banco por meio de comandos SQL que são enviados do sistema para o sistema de banco de dados. O programa (no nosso caso em C#) então itera as informações e as arranja de maneira adequada ao uso na aplicação, tanto para mostrá-la em tela quanto para iniciar outro processamento.

Incluir informações no banco de dados

O C# (ou a linguagem com que o programa está sendo programado) pode também ser usado para incluir registros no banco. Como também se trata de um comando SQL, a operação é análoga à de recuperação de

dados. A diferença é que, neste caso, o banco de dados retornará apenas um *status* de sucesso ou falha, e não registros gravados.

Atualizar e remover informações do banco de dados

O sistema, por meio de sua linguagem de programação, também pode fazer atualização ou remoção das informações gravadas em tabelas no banco (análogo ao incluir). É possível perceber que cláusulas SQL de *update* e *delete* são formadas dinamicamente no código, de maneira a especificar qual registro deve ser afetado.

Desconectar

Ao fim das operações, se realiza a desconexão ao banco de dados. Não é recomendável manter uma conexão ativa durante todo o ciclo de vida de uma aplicação.

Um termo muito usado para designar o conjunto de operações de inclusão, busca, alteração e remoção de dados em um banco por uma aplicação é a sigla CRUD (*create*, *retrieve*, *update*, *delete*, que são as operações básicas de um cadastro).

Conectando uma aplicação C# a um banco de dados MySQL

Antes de usar banco de dados com aplicações *web*, vamos experimentar os conceitos de conexão utilizando uma aplicação console. Dessa maneira, podemos nos concentrar nessas novas ferramentas de maneira mais direta. Adiante, ainda neste material, usaremos esses conceitos para integrar o banco de dados a uma aplicação ASP.NET Core MVC.

Criaremos um projeto simples e associaremos a ele a biblioteca MySQL Connector, responsável por se comunicar com o SGBD. Antes, porém, precisamos criar um banco de dados. Para isso, vamos abrir painel controlador do XAMPP ou servidor equivalente e selecionar **iniciar** o MySQL.

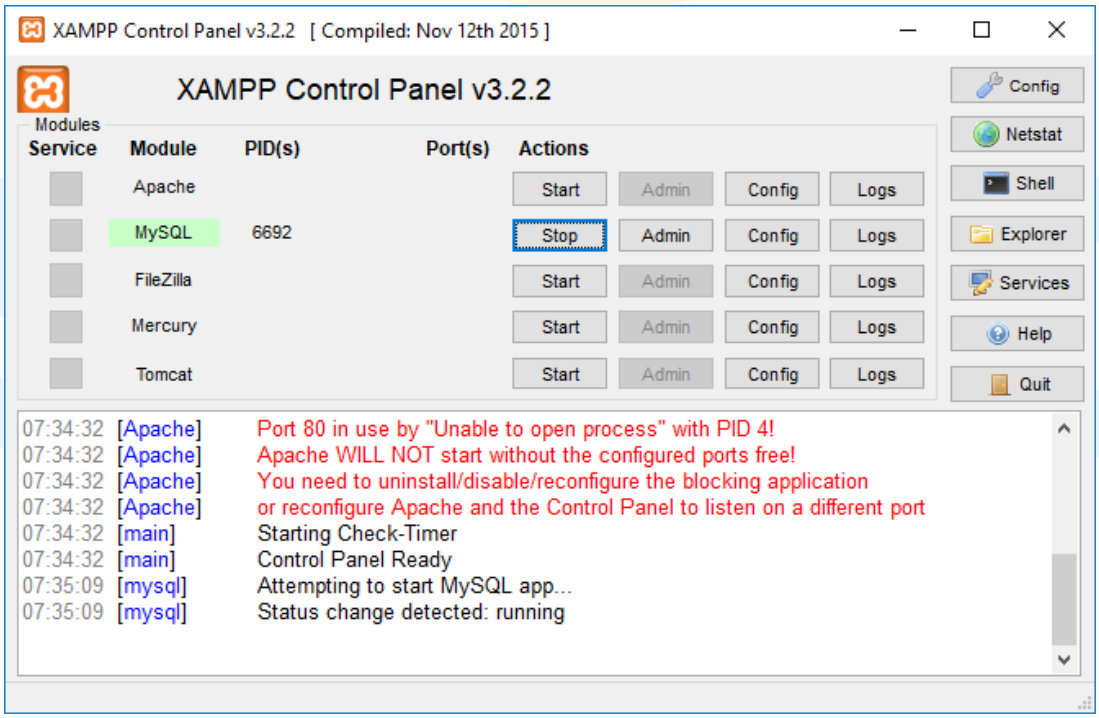


Figura 2 – Painel de controle do XAMPP, no qual podemos iniciar o serviço de banco de dados

janela do painel de controle do XAMPP em que vemos as opções Apache, MySQL, FileZilla, Mercury e Tomcat, cada uma seguida de botões Start, Admin, Config e Logs. O botão Start de MySQL está clicado, tornando-se Stop. A opção MySQL se torna destacada em verde.

Depois disso, abrimos o MySQL Workbench ou o editor de SQL equivalente para criar o banco de dados.

O XAMPP traz consigo um editor baseado em *web* chamado **PhpMyAdmin**, que pode substituir tranquilamente o Workbench. Para acessá-lo, no painel de controle do XAMPP, clique em **Start** para a opção **Apache** e depois clique no botão **Admin** da opção MySQL. O **Apache** é um

servidor *web* muito usado para trabalhar com PHP, alvo real do XAMPP. Toda a operação no phpMyAdmin ocorre pelo navegador. Para mais informações, verifique o material **Modelagem de banco de dados** desta UC (unidade curricular).

No editor, usaremos o seguinte *script* de criação de banco de dados:

```
CREATE DATABASE admprodutos;

USE admprodutos;

CREATE TABLE produto (
  id int(11) NOT NULL AUTO_INCREMENT,
  nome varchar(255),
  fabricante varchar(255),
  preco DECIMAL(11,2),
  disponivel bool,
  dataCadastro DATETIME,
  PRIMARY KEY (id)
);
```

Executamos o código SQL no editor MySQL Workbench ou equivalente.

O próximo passo será a **criação do projeto MVC**. Os passos habituais são:

- No VS Code, abrimos uma nova pasta chamada **CadProdutos**
- Acionamos o **Terminal** e usamos o comando **dotnet new console**, que cria toda a estrutura de projeto console.

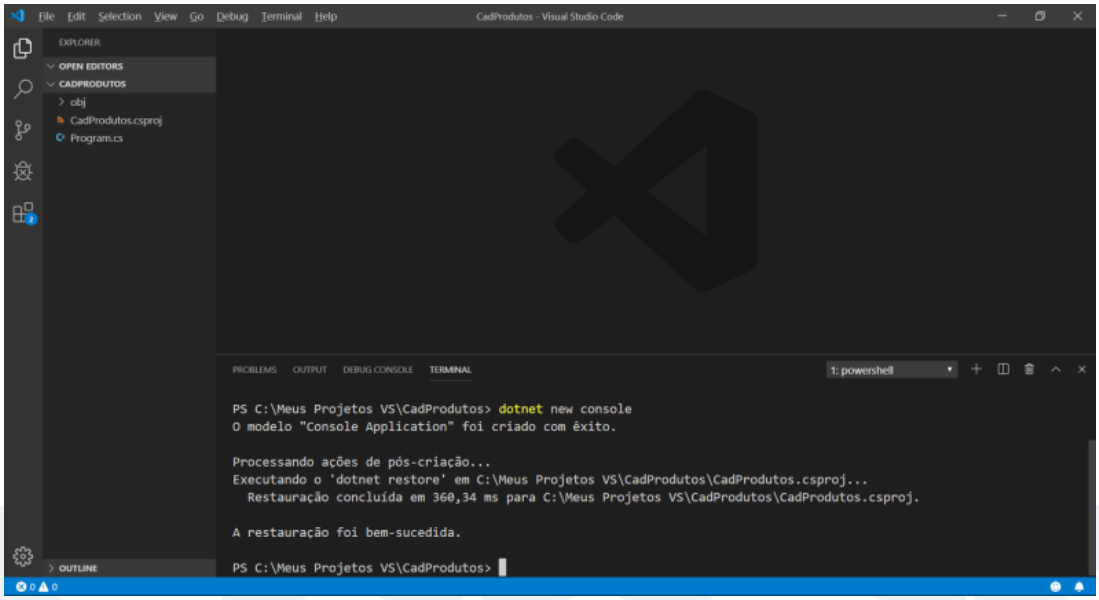


Figura 3 – Projeto Produtos criado no VS Code

janela do VSCode aberta mostra a estrutura das pastas do projeto Produto, que foi criado com o comando dotnet new mvc, cuja execução é mostrada na aba Terminal, na parte inferior da janela.

Problema com a extensão C# Omnisharp?

Dica: caso apareça a mensagem: “some projects have trouble loading. Please review the output for more details. Source: C# (Extension)”, há problemas com a extensão C# OmniSharp.

Uma análise mais atenta pode ser necessária, mas dois passos podem ser tomados para tentar solucionar a situação:

1. Utilizar uma versão anterior da extensão C#. Na guia **extensões** do VS Code, no ícone de engrenagem, selecione **Install another version** e instale a versão 1.18. Desabilite a autoatualização (ícone **...** no topo da aba **Extensions**).
2. Crie um novo arquivo no diretório base do projeto chamado **json**. Nele, inclua o seguinte conteúdo:

{

```
    "MsBuild": {
      "UseLegacySdkResolver": true
    }
}
```

3. Feche e reabra o VS Code.

Caso o problema persista, será necessário um olhar mais detalhado nas mensagens de erro mostradas com o botão **Show Output** daquele *pop up* para investigar a causa do problema. Neste caso, estamos atacando especificamente uma ocorrência em que a extensão está falhando por conta de versão desatualizada do .NET Framework instalado no Windows.

A obtenção da biblioteca de conexão com o banco de dados MySQL acontece por meio da ferramenta NuGet, integrada no .NET Core. Para utilizá-la, vamos ao **Terminal** do VS Code e digitamos o seguinte comando:

```
dotnet add package MySqlConnector
```

O comando **add package** busca no repositório remoto NuGet a última versão do projeto solicitado (nesse caso, o MySQL Connector), baixa essa biblioteca, inclui no projeto e o configura para que ela possa ser usada em nosso código. O resultado, caso tudo dê certo, será algo próximo do seguinte:

```
PS C:\Meus Projetos VS\Produtos> dotnet add package MySqlConnector
  Writing C:\Users\Daltron\AppData\Local\Temp\tmpE17F.tmp
info : Adicionando PackageReference do pacote 'MySqlConnector' ao projeto
'C:\Meus Projetos VS\Produtos\Produtos.csproj'.
info : Restaurando pacotes para C:\Meus Projetos VS\Produtos\Produtos.csp
roj...
info : GET https://api.nuget.org/v3-flatcontainer/mysqlconnector/index.js
on
info : OK https://api.nuget.org/v3-flatcontainer/mysqlconnector/index.jso
n 589ms
info : O pacote 'MySqlConnector' é compatível com todas as estruturas esp
ecificadas no projeto 'C:\Meus Projetos VS\Produtos\Produtos.csproj'.
info : PackageReference do pacote 'MySqlConnector' versão '0.56.0' adicio
nada ao arquivo 'C:\Meus Projetos VS\Produtos\Produtos.csproj'.
info : Confirmando restauração...
info : Writing assets file to disk. Path: C:\Meus Projetos VS\Produtos\ob
j\project.assets.json
log  : Restauração concluída em 4,08 sec para C:\Meus Projetos VS\Produto
s\Produtos.csproj.
```

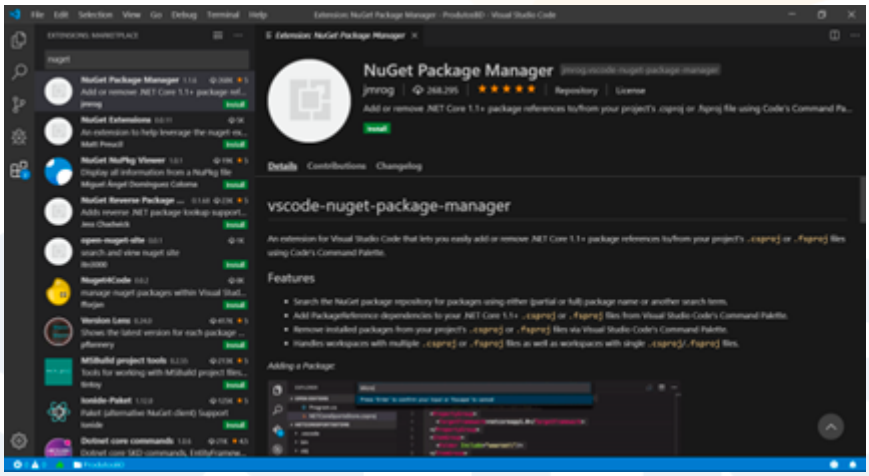
No arquivo **CadProdutos.csproj** do projeto, podemos notar, na altura da linha 12, aproximadamente, que foi incluída a seguinte referência:

```
PackageReference Include="MySqlConnector" Version="0.56.0" />
```

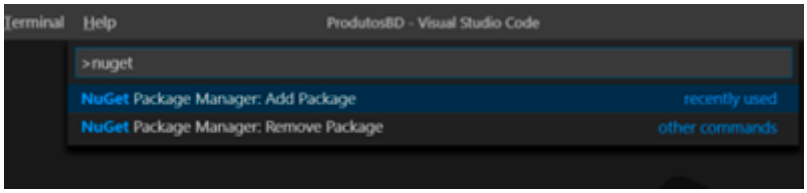
Essa referência indica que podemos usar essa biblioteca em nossos códigos a partir de uma diretiva *using*.

Da mesma maneira, poderíamos incluir outros pacotes disponíveis no NuGet. Para consultar a galeria de pacotes disponíveis, acesse o *site* do Nuget (busque por **NuGet Gallery** no *site* de buscas que você costuma utilizar) e navegue até a opção **Packages**.

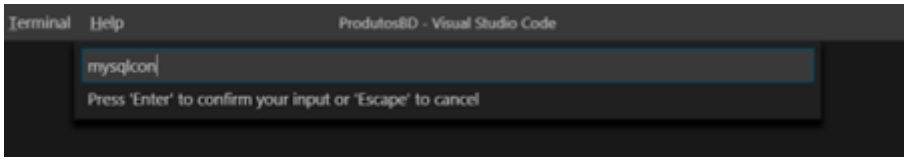
Dica: no VS Code, há ainda a opção de instalar a extensão **NuGet Package Manager**. Na guia **Extensions**, busque por **NuGet** e selecione **NuGet Package Manager**. Clique no botão **Install**. A imagem a seguir mostra o detalhe da extensão.



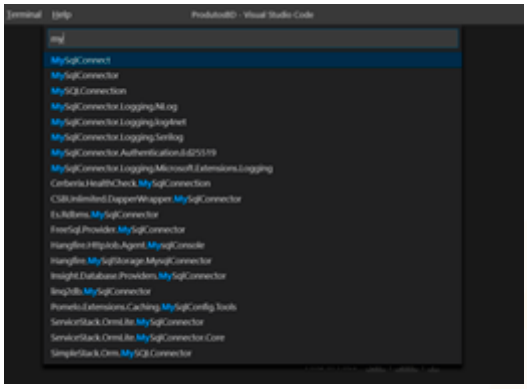
Para usá-lo, acione a paleta de comandos usando o menu **View** > **Command Palette** ou pelo atalho Ctrl + Shift + P. Na caixa suspensa, digite **NuGet** e selecione **NuGet Package Manager: Add Package**.



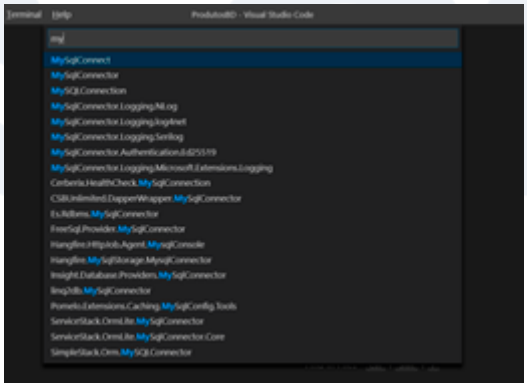
Em seguida, na caixa suspensa, digite um termo de busca para encontrar o pacote que deseja importar. No exemplo, vamos buscar por **mySQLCon** na intenção de encontrar o pacote MySQL Connector, digitando **Enter**, em seguida.



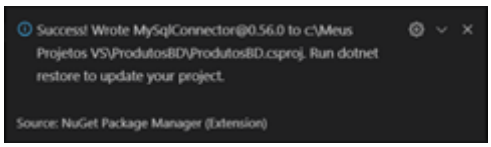
A lista de sugestões é extensa para o termo buscado. Podemos aplicar novo filtro novamente na caixa suspensa da paleta de comando.



Clicando no resultado desejado (ou selecionando com setas no teclado e digitando **Enter**), somos levados a escolher uma versão do pacote.



Após escolher a versão, a instalação é realizada e a seguinte mensagem de sucesso surge em um *pop up* no canto inferior direito.



Uma vez que temos a biblioteca de conexão com o MySQL incluída, podemos programar a conexão. Para isso, vamos criar no diretório base do projeto uma classe que cuidará das operações com o banco de dados. Clique com o botão direito do *mouse* e selecione **New C# Class** (opção proveniente da extensão C# Extensions), nomeando o novo arquivo como **ProdutoRepository.cs**.

```
namespace CadProdutos
{
  public class ProdutoRepository
  {

  }
}
```

Nesta classe, faremos nosso trabalho de conexão com o banco de dados. Para isso, criamos um método chamado **Insert()**, que incluirá um registro na tabela **Produto** do banco de dados que criamos há pouco.

O primeiro passo no código método é definir a conexão usando a classe **MySqlConnection**.

```
public void Insert()
{
  MySqlConnection conexao = new MySqlConnection();
}
```

Será necessário incluir uma referência *using* para o *namespace***MySql.Data.MySqlClient**. A ferramenta da lâmpada do editor ajuda com sugestões.

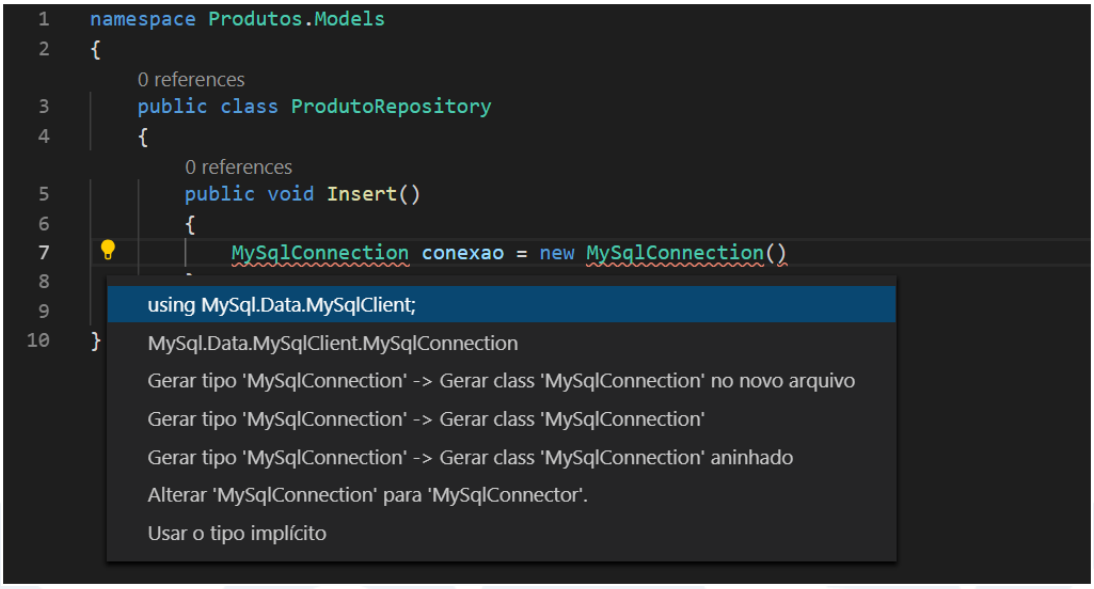


Figura 4 – A lâmpada à esquerda na linha de código ajuda com sugestão de *using* a ser acrescentado

Detalhe do código no VS Code mostrando a linha MySqlConnection conexao = new MySqlConnection() marcada em vermelho, indicando erro. À esquerda, a lâmpada de sugestões aparece. A imagem mostra a lâmpada clicada e o menu suspenso mostrando como primeira opção using MySql.Data.MySqlClient;.

Para realizar a conexão, é necessário informar ao objeto o endereço do servidor de banco de dados e o nome do banco que será usado. Esse tipo de informação é geralmente chamado de *string* de conexão.

```
public void Insert()
{
  string enderecoConexao = "Database=admprodutos;Data Source=localhost;Us
er Id=root;";
  MySqlConnection conexao = new MySqlConnection(enderecoConexao);
}
```

Na variável **enderecoConexao**, definimos a seguinte configuração:

Database=admprodutos;

Indica qual banco de dados vamos utilizar. Lembre-se de que, nos passos anteriores, criamos um banco de dados chamado **admprodutos**.

Data Source=localhost;

No endereço do banco de dados, geralmente informa-se um IP (como 192.168.10.20, por exemplo). *Localhost* indica que o servidor de banco de dados está na mesma máquina em que o servidor *web* (ou seja, nossa aplicação) está executando.

User Id=root;

Na instalação padrão do MySQL pelo XAMPP, o usuário *root* (o mais importante do SGBD) é criado sem senha. Caso tivéssemos uma senha configurada, precisaríamos preencher também a informação **Password** (exemplo: Password=12345;).

Há várias outras opções que podem ser configuradas na *string* de conexão, como, por exemplo, porta de acesso, tempo de expiração da conexão, criptografia e outros.

Algumas das propriedades de *string* de conexão têm sinônimos. **Data Source** pode ser usado também como **DataSource**, **Host**, **Server**, **Address**, **Addr** ou **Network Address**. Em vez de informar o banco de dados com **Database**, podemos usar **Initial Catalog**. **User Id** pode ser abreviado como **Uid** ou expandido como **Username** ou **User name**. O funcionamento da conexão não será afetado pela escolha de um ou outro termo.

Informamos então a *string* de conexão como um parâmetro para o construtor de MySQL Connection.

A seguir, podemos abrir a conexão, usando o método **Open()** do objeto **conexao**.

```
conexao.Open();
```

Ao abrir a conexão com o banco de dados, estamos prontos para enviar-lhe comandos SQL de manipulação de dados. No caso das bibliotecas de MySQL Connector, usamos a classe **MySqlCommand** para formar um desses comandos. Vamos usá-la da seguinte maneira:

```
string sqlInsert =
  "INSERT INTO produto (nome, fabricante, preco, dataCadastro, disponive
l)"+
  "VALUES ('prodTeste', 'fabrTeste', 1.99, NOW(), 1)";
MySqlCommand comando = new MySqlCommand(sqlInsert, conexao);
comando.ExecuteNonQuery();
```

Primeiro, definimos a consulta SQL que precisamos executar. Nesse caso, um comando **Insert**. Note que a cláusula é formada como um *string* e é exatamente como se fossemos digitar em um editor como o MySQL Workbench. Também é importante ver que os valores a serem cadastrados estão fixos.

Não é informado valor para a coluna **id** com autoincremento na tabela, ou seja, o valor para esta coluna é calculado e incluído automaticamente. Para a coluna **dataCadastro**, informamos o valor **NOW()**. Esta é uma função própria do MySQL que retorna data e hora atuais.

Na segunda linha, criamos um objeto do tipo MySQL Command a cujo construtor informamos a cláusula SQL formada e também a conexão aberta com o banco de dados.

Por fim, invocamos o método **ExecuteNonQuery()** do objeto comando. Esse método executa uma instrução que não retorna valores (é o caso do **Insert**). No caso de uma cláusula **Select**, usaríamos outros métodos como **ExecuteScalar()** e **ExecuteReader()**.

Encerrando o método, encerramos a conexão com a seguinte chamada:

```
conexao.Close();
```

O código completo do método **Insert** da classe **ProdutoRepository** fica como o seguinte:

```
public void Insert()
{
  string enderecoConexao = "Database=admprodutos;Data Source=localhost;Us
er Id=root;";
  MySqlConnection conexao = new MySqlConnection(enderecoConexao);
  conexao.Open();

  string sqlInsert =
    "INSERT INTO produto (nome, fabricante, preco, dataCadastro, disponiv
el)"+
    "VALUES ('prodTeste', 'fabrTeste', 1.99, NOW(), 1)";
  MySqlCommand comando = new MySqlCommand(sqlInsert, conexao);
  comando.ExecuteNonQuery();

  conexao.Close();
}
```

Para testar esse código, precisamos fazer a invocação do método **Insert()** que criamos. Como estamos em um projeto console, podemos fazer essa chamada diretamente na classe **Program**, no método **Main()**.

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("Cadastrando Produto");
    ProdutoRepository pr = new ProdutoRepository();
    pr.Insert();
  }
}
```

Para testar, usamos no terminal no VS Code o comando dotnet build para compilar a aplicação e, em seguida, dotnet run para executá-la. O programa em si nos traz como resposta apenas a frase **Cadastrando Produto**, nada especial, mas podemos utilizar o MySQL Workbench ou o editor de SQL escolhido para rodar a seguinte consulta:

SELECT * FROM produto;

O resultado não deverá ser muito diferente do seguinte:

	id	nome	fabricante	preco	disponivel	dataCadastro
	1	prodTeste	fabrTeste	1.99	1	2019-08-20 10:03:22
▶*	NULL	NULL	NULL	NULL	NULL	NULL

Figura 5 – Resultado da consulta SELECT * FROM produto executada no MySQL Workbench

grid com os títulos de colunas: id, nome, fabricante, preco, disponivel, dataCadastro; tabela preenchida com valores '1', 'prodTeste', 'fabrTeste', '1.99', '1', '2019-08-26 10:03:22'.

Foi incluído um registro com o nome **prodTeste**, fabricante **fabrTeste**, preço disponibilidade e data de cadastro também preenchidas. Sinal de sucesso, nosso código de conexão e manipulação do banco de dados está

correto. Se rodarmos novamente o programa, poderemos ver no banco de dados que um novo registro (com os mesmos valores) é incluído. Isso é totalmente esperado, já que nosso código faz a inclusão de valores fixos na cláusula **insert**.

Para tornar esse cadastro um pouco mais interessante, vamos permitir entradas de dados para nome e fabricante. Para isso, alteramos o método **Main** da classe **Program** da seguinte maneira:

```
static void Main(string[] args)
{
  Console.WriteLine("Cadastrando Produto");

  Console.Write("Digite o nome do produto: ");
  string nome = Console.ReadLine();

  Console.Write("Digite o fabricante do produto: ");
  string fabricante = Console.ReadLine();

  ProdutoRepository pr = new ProdutoRepository();
  pr.Insert(nome, fabricante);
}
```

A seguir, vamos adaptar o método **Insert()** para que não cadastre apenas dados fixos. Para isso, voltamos a **ProdutoRepository** e incluímos parâmetros ao método, um para transmitir o nome do produto e outro para o fabricante. As outras colunas podem ficar, por ora, com valores fixos.

```
public void Insert(string nome, string fabricante)
{
  string enderecoConexao = "Database=admprodutos;Data Source=localhost;Us
er Id=root;";
  MySqlConnection conexao = new MySqlConnection(enderecoConexao);
  conexao.Open();

  string sqlInsert =
    "INSERT INTO produto (nome, fabricante, preco, dataCadastro, disponiv
el)" +
    "VALUES ('"+ nome +"', '"+ fabricante +"', 1.99, NOW(), 1)";
  MySqlCommand comando = new MySqlCommand(sqlInsert, conexao);
  comando.ExecuteNonQuery();
  conexao.Close();
}
```

Ao rodar novamente o programa no terminal do VS Code, podemos informar os valores e, ao encerrar o programa, podemos verificar o resultado de nossa aplicação consultando o banco de dados no editor MySQL Workbench, ou equivalente, para verificar se os dados foram incluídos.

	id	nome	fabricante	preco	disponivel	dataCadastro
	1	prodTeste	fabrTeste	1.99	1	2019-08-20 10:03:22
	2	produto 1	fabricante 1	1.99	1	2019-08-20 10:32:50
▶*	NULL	NULL	NULL	NULL	NULL	

Figura 6 – Resultado da consulta por produtos, com registro “produto 1” cadastrado a partir do formulário de home/cadastro.

a imagem mostra o resultado de uma consulta realizada no MySQL Workbench com alguns registros, entre eles id = 2, nome = produto 1 e fabricante = fabricante 1.

Essa foi nossa primeira experiência com a conexão de banco de dados, um projeto simples, mas que realiza com sucesso a operação de conexão e inclusão de registro no banco de dados. Na próxima seção, retomaremos alguns conceitos básicos de lógica e os aplicaremos às rotinas de buscas no banco.

Realizando buscas no banco de dados e aplicando conceitos básicos de lógica

A operação de **insert** é muito básica e, por não retornar valores, não envolve informações resultantes do banco de dados. A partir de agora, vamos usar **select** no banco de dados e manipular os registros que vêm dessas consultas. Esta é a oportunidade de rever alguns conceitos que trabalhamos em algumas UCs, então este conteúdo também serve como uma pequena revisão, mas orientado para uma aplicação específica.

Informações retornadas pelo banco de dados: exercitando tipos de dados, variáveis e constantes

Vamos usar o projeto iniciado na seção anterior e expandi-lo com consultas. Para isso, usaremos a seguinte cláusula SQL na aplicação:

```
SELECT * FROM produto;
```

Ela traz todos os registros, com valores para todas as colunas da tabela **produto**.

Revisão de conceito

- Tipos de dados: cada variável, cada informação, tem um tipo próprio (numérico, textual, lógico, entre outros). Em C#, dizemos que a linguagem é **fortemente tipada**, porque as variáveis precisam especificar claramente que tipo de informação que ela pode carregar. Outras linguagens, como o PHP, são **fracamente tipadas**, pois uma variável pode carregar informação de qualquer tipo.

- Entre os tipos do C#, temos: int (números inteiros), string (texto), bool (lógico – verdadeiro ou falso), float ou double (números fracionários), que também podem ser considerados como tipos as classes.

- Variáveis são contêineres de informação em que o dado é mutável, como na seguinte declaração:

```
int valor;

valor = 10;
```

- Constantes são contêineres de dados em que não se pode fazer alteração:

```
const int valor = 10;

valor = 15; //erro
```

- Constantes são usadas para quando se quer representar um valor imutável. Um exemplo recorrente é definir uma constante para o valor PI (3,14159...)

- O SQL usa tipos de dados para suas colunas. Variáveis e constantes podem ser usadas apenas em um contexto de um *script* com várias instruções ou construções, como **Trigger**, **Procedure** e **Function**. Curiosamente, as variáveis no MySQL são fracamente tipadas.

Cada coluna na tabela **produto** tem um tipo próprio:

- **Id** é do tipo **Int**, correspondente ao tipo int no C#.
- **Nome** e **Fabricante** são do tipo **Varchar** (255), ou seja, uma cadeia de caracteres de no máximo 255 letras, correspondente ao tipo string no C#.
- Preço é do tipo **Decimal** (11, 2), indicando que é um número de ponto flutuante que tem duas casas decimais. Em C#, podemos usar double.
- **Disponível** é do tipo **Bool**, que, no SQL, pode carregar valores 0 ou 1. Corresponde ao bool no C#.

Você pode testar no MySQL Workbench ou no editor de SQL que estiver usando para verificar o resultado daquela consulta e comparar com o que obteremos com o experimento a seguir.

No VS Code, no projeto **CadProdutos**, vamos alterar a classe **ProdutoRepository**, para que ela realize a consulta. Criamos um método chamado **Query()**:

```
public void Query()
{

}
```

Como fizemos anteriormente, precisaremos realizar a conexão com banco de dados.

```
public void Query()
{
  string enderecoConexao = "Database=admprodutos;Data Source=localhost;Us
er Id=root;";
  MySqlConnection conexao = new MySqlConnection(enderecoConexao);
  conexao.Open();
}
```

Note que estamos repetindo aqui a *string* de conexão usada no **insert**. É boa ideia deixá-la como uma constante na classe **ProdutoRepository**; podemos considerar isso como uma boa prática, já que, se precisarmos alterar alguma coisa na conexão, como o endereço ou o usuário, em vez de modificar em dois pontos (como está agora), mudaremos apenas um (a constante).

Após a alteração, a classe completa ficará desta maneira (omitindo-se o *namespace*):

```
public class ProdutoRepository
{
  private const string enderecoConexao = "Database=admprodutos;Data Sourc
e=localhost;User Id=root;";

  public void Insert(string nome, string fabricante)
  {
    MySqlConnection conexao = new MySqlConnection(enderecoConexao);
    conexao.Open();

    string sqlInsert =
      "INSERT INTO produto (nome, fabricante, preco, dataCadastro, dispon
ivel)" +

      "VALUES ('" + nome + "', '" + fabricante + "', 1.99, NOW(), 1)";
    MySqlCommand comando = new MySqlCommand(sqlInsert, conexao);
    comando.ExecuteNonQuery();

    conexao.Close();
  }
  public void Query()
  {
    MySqlConnection conexao = new MySqlConnection(enderecoConexao);
    conexao.Open();
  }
}
```

Como enderecoConexao é constante, não poderá ser alterado via código por nenhuma outra classe do programa, o que é completamente coerente com o que precisamos.

Seguindo com o trabalho no método **Query**, incluímos a consulta utilizando a classe **MySQL Command**.

```
public void Query()
{
  MySqlConnection conexao = new MySqlConnection(enderecoConexao);
  conexao.Open();

  string sqlSelect = "SELECT * FROM produto";
  MySqlCommand comandoQuery = new MySqlCommand(sqlSelect, conexao);
  MySqlDataReader resultado = comandoQuery.ExecuteReader();
}
```

Esse processo é semelhante ao que tínhamos no método **Insert**. No entanto, a diferença principal, além da consulta, é que, em vez de chamar **ExecuteNonQuery()**, precisamos invocar o método **ExecuteReader()**, que retorna um objeto **MySqlDataReader**. Esse método é adequado para quando prevemos que a consulta retornará elementos e servirá para nos fornecer os registros recuperados pela consulta SQL.

O próximo passo é justamente usar as informações do *reader* (objeto **resultado**) para recuperar os dados do produto. O método **Read()** é necessário para recuperarmos o próximo registro trazido pela consulta.

```
resultado.Read();
```

Em seguida, podemos consultar os valores de cada coluna. Veja como obtemos o valor de id:

```
int id=resultado.GetInt32("Id");
```

Métodos como **GetInt32()** são necessários para converter o tipo de coluna no banco de dados para o tipo de dados no C#. O texto passado por parâmetro é o nome da tabela da maneira que vem pelo **select** do banco de dados (nesse caso, corresponde à coluna Id). Analogamente, obtemos os valores de nome, fabricante e outras colunas da tabela usando o método **Get** correspondente:

```
string nome = resultado.GetString("Nome");
string fabricante = resultado.GetString("Fabricante");
decimal preco = resultado.GetDecimal("preco");
bool disponivel = resultado.GetBoolean("disponivel");
DateTime dataCadastro = resultado.GetDateTime("dataCadastro");
```

No C# decimal, é um tipo equivalente a double e float. Entre os três tipos, é o que comporta os maiores valores.

Com as variáveis todas criadas, podemos mostrar na tala do terminal as informações do produto:

```
Console.WriteLine($"Produto: {nome}; Fabricante: {fabricante}; "+
$"Preco: {preco}; Disponível: {disponivel}; Data Cadastro:{dataCadastro}"
);
```

Após o código anterior, será necessário incluir uma cláusula **using System;**, por causa dos tipos **DetaTime** e **Console**. Isso pode ser feito manualmente ou com ajuda do assistente, que aparece no ícone da lâmpada na parte esquerda do código.

Por último, encerramos o *reader* e a conexão:

```
resultado.Close();
conexao.Close();
```

O método **Query** completo ficará da seguinte forma:

```
public void Query()
{
  MySqlConnection conexao = new MySqlConnection(enderecoConexao);
  conexao.Open();

  string sqlSelect = "SELECT * FROM produto";
  MySqlCommand comandoQuery = new MySqlCommand(sqlSelect, conexao);
  MySqlDataReader resultado = comandoQuery.ExecuteReader();

  resultado.Read();

  int id=resultado.GetInt32("Id");
  string nome = resultado.GetString("Nome");
  string fabricante = resultado.GetString("Fabricante");
  decimal preco = resultado.GetDecimal("preco");
  bool disponivel = resultado.GetBoolean("disponivel");
  DateTime dataCadastro = resultado.GetDateTime("dataCadastro");

  Console.WriteLine($"Produto: {nome}; Fabricante: {fabricante}; "+
  $"Preco: {preco}; Disponível: {disponivel}; Data Cadastro:{dataCadastr
o}");

  resultado.Close();
  conexao.Close();
}
```

Lembre-se de eventualmente realizar a compilação do projeto usando dotnet build para corrigir possíveis erros antes que eles se acumulem.

Para botar o método **Query()** em ação, vamos incluir uma chamada a ele no método **Main()** da classe **Program**.

```
static void Main(string[] args)
{
  Console.WriteLine("Cadastrando Produto");

  Console.Write("Digite o nome do produto: ");
  string nome = Console.ReadLine();

  Console.Write("Digite o fabricante do produto: ");
  string fabricante = Console.ReadLine();

  ProdutoRepository pr = new ProdutoRepository();
  pr.Insert(nome, fabricante);
  pr.Query();
}
```

Para testar, executamos dotnet run. Evidentemente, o programa primeiro solicitará informações para o **Insert()** que mantivemos ali e, após isso, realizará a consulta. O resultado deverá ser algo próximo ao seguinte:

```
Produto: prodTeste; Fabricante: fabrTeste; Preco: 1,99; Disponível: True;
Data Cadastro:20/08/2019 10:03:22
```

A nossa consulta retorna apenas o primeiro registro recuperado no banco de dados. A seguir, expandiremos a aplicação para que ela nos mostre todos os registros da tabela.

Erros de digitação no código podem acontecer (e sem dúvidas acontecerão). Um inconveniente gerado pelo uso de SQL no código é a consulta não ser analisada via compilador. Então, erros podem ser descobertos apenas durante a execução da aplicação. Alguns exemplos mais comuns são:

- *System InvalidCastException: Unable to cast object of type '<Type>' to type '<Other Type>'*, indicando que não é possível converter de um tipo a outro tipo. Este erro aconteceria se, por exemplo, optássemos por usar **resultado.GetInt32("Nome");**. Como **Nome** é texto, a conversão não seria possível.
- *MySqlException: Column count doesn't match value count at row 1*. Este erro indica que a contagem de coluna não bate com o número de valores informados e poderia ocorrer na seguinte cláusula: **insert tabela (col1, col2) values ('col1')**, pois estamos especificando duas colunas, mas informando apenas um valor.
- *MySqlException: Unknown database 'banco'*, que indica que o banco de dados informado não foi encontrado. Nesse caso, é provável que a *string* de conexão esteja incorreta.
- *MySqlException: Unable to connect to any of the specified MySQL hosts* indica que não foi possível conectar ao servidor MySQL. Isso ocorre quando o banco de dados MySQL não está iniciado ou está inacessível por algum motivo (como por causa de algum problema na rede, por exemplo).

Os erros aparecerão no **Terminal** do VS Code no formato de **Stack trace**, ou seja, um encadeamento de falhas que aconteceram devido a um problema original. Por isso, é necessário analisar as mensagens surgidas no **Terminal**. Veja um exemplo de *stack trace*.

```
Unhandled Exception: MySql.Data.MySqlClient.MySqlException: Unable to con
nect to any of the specified MySQL hosts.

at MySqlConnector.Core.ServerSession.ConnectAsync(ConnectionSettings cs,
 ILoadBalancer loadBalancer, IOBehavior ioBehavior, CancellationToken can
cellationToken) in C:\projects\mysqlconnector\src\MySqlConnector\Core\Ser
verSession.cs:line 328

at CadProdutos.ProdutoRepository.Insert(String nome, String fabricante) i
n C:\Meus Projetos VS\CadProdutos\ProdutoRepository.cs:line 13

at CadProdutos.Program.Main(String[] args) in C:\Meus Projetos VS\CadProd
utos\Program.cs:line 18
```

A primeira mensagem (destacada em verde no exemplo anterior) geralmente é a mais significativa. A última (destacada em laranja no exemplo anterior) é a que nos indica em que momento de nosso código ela se originou.

Iterando sobre registros: exercitando decisões, repetições, vetores e estruturas de dados

Um inconveniente da nossa última experiência é o fato de que, ao testar, precisamos primeiro passar pela inserção, para só depois experimentar a busca com o método **Query()**. Uma limitação é o fato de que este método nos retorna apenas informação de um registro. Por isso, vamos usar decisões e repetições para melhorar esses aspectos.

Revisão de conceito

- Decisões (ou condicionais) são estruturas básicas de lógica que permitem que o fluxo de um código seja desviado de acordo com uma condição lógica. No C#, usamos **if**.

```
if(valor > 0)

{

    Console.WriteLine("Valor positivo");

}

else

{

    Console.WriteLine("Valor negativo");

}
```

- Outra estrutura para condição é o *switch*, que verifica vários valores possíveis para uma variável.
- Laços de repetição são também estruturas básicas de lógica que permitem que a execução de um bloco de instruções aconteça repetidas vezes até que uma condição seja satisfeita. No C#, usamos **while**, **for** e **do-while**.

```
while(comando != "X")

{

    valor++;

    comando = Console.ReadLine();

}
```

- O C# fornece ainda o **foreach**, que permite a iteração sobre listas ou vetores.
- Estruturas de dados são organizações de dados na memória do computador de maneira que eles sejam acessados eficientemente. Entre as principais, estão vetores (*arrays*), listas, dicionários, pilhas e filas.

```
int[] vetorInteiros = new int[] {1, 1, 2, 3, 5, 8};
```

- O C# fornece classes que implementam o comportamento de lista infinita (**List<>**), dicionário (**Dictionary<,>**), pilhas (**Stack<>**) e filas (**Queue<>**).

Exercitando o condicional e melhorando nosso teste, vamos usar uma estrutura *switch* para implementar uma espécie de menu em nosso programa, para inserir produto ou consultar. Essa alteração ocorre no método **Main()** de **Program**.

```
static void Main(string[] args)
{
  Console.WriteLine("Cadastrando Produto");
  Console.Write("Digite C para cadastrar ou L para listar:");
  string comando = Console.ReadLine();

  ProdutoRepository pr = new ProdutoRepository();
  string nome, fabricante;

  switch(comando)
  {
    case "C":
    Console.Write("Digite o nome do produto: ");
    nome = Console.ReadLine();

    Console.Write("Digite o fabricante do produto: ");
    fabricante = Console.ReadLine();

    pr.Insert(nome, fabricante);
    break;

  case "L":
    pr.Query();
    break;

  default:
    Console.WriteLine("Comando inválido");
    break;
  }
}
```

O usuário digitará **C** ou **L** e, de acordo com o que informou, usando a estrutura *switch*, o programa executará a rotina de cadastro ou de listagem. Algumas adaptações acessórias foram necessárias, como mudança de local da declaração das variáveis **nome** e **fabricante**, que agora precisa acontecer antes do *switch*.

Note que, se o usuário digitar **c** ou **l** minúsculos, o programa não reconhecerá o comando como válido, pois o C# diferencia maiúsculo de minúsculo nas comparações. Para solucionar isso, temos a opção de tornar

o valor informado em maiúsculo ou minúsculo, para então comparar. No exemplo, poderíamos empregar o método **ToUpper()** de *string* da seguinte maneira.

```
switch(comando.ToUpper())
```

ToUpper() torna todos os caracteres do *string* maiúsculos. Como em *case*, estamos comparando com maiúsculos, então na prática, se o usuário digitar em caixa alta ou baixa, o programa entenderá o comando.

Analogamente, poderíamos usar o método **ToLower()**, que torna os caracteres minúsculos.

O teste está rodando o programa com o comando dotnet run, então, experimente os comandos separadamente.

Outro incômodo é o fato de a listagem nos mostrar apenas o primeiro registro gravado no banco de dados na tabela **produto**. Para isso, precisaremos de uma estrutura de repetição: vamos mostrar o valor de cada um dos registros recuperados pelo objeto de tipo **MySqlDataReader** no método **Query()** de **ProdutoRepository**. A seguir, veja o código após a mudança:

```
public void Query()
{
  MySqlConnection conexao = new MySqlConnection(enderecoConexao);
  conexao.Open();

  string sqlSelect = "SELECT * FROM produto";
  MySqlCommand comandoQuery = new MySqlCommand(sqlSelect, conexao);
  MySqlDataReader resultado = comandoQuery.ExecuteReader();

  while(resultado.Read())
  {
    int id=resultado.GetInt32("Id");
    string nome = resultado.GetString("Nome");
    string fabricante = resultado.GetString("Fabricante");
    decimal preco = resultado.GetDecimal("preco");
    bool disponivel = resultado.GetBoolean("disponivel");
    DateTime dataCadastro = resultado.GetDateTime("dataCadastro");

    Console.WriteLine($"Produto: {nome}; Fabricante: {fabricante}; " +
    $"Preco: {preco}; Disponível: {disponivel}; Data Cadastro:{dataCadast
ro}");
  }
  resultado.Close();
  conexao.Close();
}
```

A alteração foi simples, em razão do tipo de retorno do método **Read()** de **MySqlDataReader**, que retorna verdadeiro, caso ainda haja registros a serem lidos; ou falso, caso ocorra o contrário. Assim, basta um while condicionado por resultado.Read(); o escopo do laço envolve todo o código de recuperação dos dados dos campos e escrita na tela com **WriteLine()**.

Execute novamente o programa e verifique a opção de listagem. O resultado, caso tudo ocorra como o esperado, será todos os registros cadastrados no banco de dados sendo mostrados na aba de **Terminal**, como o exemplo:

```
Cadastrando Produto
Digite C para cadastrar ou L para listar:L
Produto: prodTeste; Fabricante: fabrTeste; Preco: 1,99; Disponível: True;
Data Cadastro:20/08/2019 10:03:22
Produto: produto 1; Fabricante: fabricante 1; Preco: 1,99; Disponível: Tr
ue; Data Cadastro:20/08/2019 10:32:50
```

Apesar de termos um resultado apropriado, há algo conceitualmente incorreto em nossa implementação do método **Query()**. Nele, fazemos toda a consulta e mostramos na tela diretamente. Estamos misturando duas responsabilidades: a de consultar o banco de dados e a de apresentar as informações ao usuário.

Lembra-se do MVC?

É como se estivéssemos usando *view* dentro de um *model*, o que não é recomendável.

Como podemos fazer essa separação? Podemos usar uma estrutura de dados de lista e retorná-la ao código que invocar **Query()**. Este, por sua vez, usa e manipula essa lista como quiser.

A primeira coisa a fazer é criar uma classe **Produto**, que conterá toda a informação desse tipo de produto. Para isso, criamos uma classe no diretório base do programa, chamando o arquivo de **Produto.cs**. Então, preenchemos a classe com propriedades para cada uma das colunas definidas na tabela **produto** do banco de dados. Além disso, também vamos incluir um método acessório para mostrar os valores do objeto como um texto.

```
using System;

namespace CadProdutos
{
  public class Produto
  {
    public int Id {get; set;}
    public string Nome {get; set;}
    public string Fabricante {get; set;}
    public decimal Preco {get; set;}
    public bool Disponivel {get; set;}
    public DateTime DataCadastro {get; set;}

    public override string ToString()
    {
      return $"Id {Id}; Produto: {Nome}; Fabricante: {Fabricante}; Preco:
{Preco}; Disponível: {Disponivel}; Data Cadastro:{DataCadastro}";
    }
  }
}
```

Usamos *override* no método **ToString()**, porque ele é próprio da classe **Object**, da qual qualquer classe deriva. Consulte o material **Fundamentos de programação** desta UC para mais informações sobre sobrecarga de métodos.

Agora, no método **Query()** de **ProdutoRepository**, podemos montar uma lista de objetos do tipo **Produto**. Com isso, não é mais necessário utilizar tantas variáveis e podemos concentrar tudo em um objeto.

```
public List<Produto> Query()
{
  MySqlConnection conexao = new MySqlConnection(enderecoConexao);
  conexao.Open();

  string sqlSelect = "SELECT * FROM produto";
  MySqlCommand comandoQuery = new MySqlCommand(sqlSelect, conexao);
  MySqlDataReader resultado = comandoQuery.ExecuteReader();

  List<Produto> listaProdutos = new List<Produto>();

  while(resultado.Read())
  {
    Produto item = new Produto();
    item.Id = resultado.GetInt32("Id");
    item.Nome = resultado.GetString("Nome");
    item.Fabricante = resultado.GetString("Fabricante");
    item.Preco = resultado.GetDecimal("preco");
    item.Disponivel = resultado.GetBoolean("disponivel");
    item.DataCadastro = resultado.GetDateTime("dataCadastro");

    listaProdutos.Add(item);
  }
  resultado.Close();
  conexao.Close();

  return listaProdutos;
}
```

Pontos importantes dessa alteração:

- Na assinatura do método, em vez de void, trocamos o tipo de retorno para List<Produto>
- Antes do while, declaramos o objeto listaProduto que armazenará todos os objetos formados a partir da pesquisa feita no banco de dados
- A cada registro recuperado pelo *reader* resultado, montamos um novo objeto **Produto** e o adicionamos a listaProduto
- Antes de fechar o método, incluímos a cláusula de retorno com o objeto listaProduto
- Por conta do tipo List<>, precisamos incluir a cláusula using Collections.Generic; no tipo do arquivo. Lembre-se de que você pode usar também a ferramenta de sugestões da lâmpada amarela, que aparece à esquerda do código.

Podemos dizer que, de maneira geral, as consultas em banco de dados obedecerão a essa estrutura.

É hora de adaptar a chamada do método **Query()** para essa nova realidade. Veja no código abaixo o detalhe do método **Main()** da classe **Program**.

```
case "L":
  List<Produto> lista = pr.Query();
  foreach(Produto p in lista)
  {
    Console.WriteLine(p.ToString());
  }
  break;
```

Usamos um laço do tipo foreach, que passa por cada elemento presente na lista. Também usamos aqui o método **ToString()**, que definimos na classe **Produto**.

Porque refatoramos nosso código, ele está mais completo e mais correto. Ao testar utilizando o comando **dotnet run**, a resposta deve ser similar à obtida anteriormente. A vantagem que temos com essa alteração é a de que podemos usar essas classes (**Produto** e **ProdutoRepository**) da maneira como está escrita em um projeto *web*, por exemplo. Isso não seria recomendável da maneira como estava, com chamadas a **Console.WriteLine()** dentro da classe **ProdutoRepository**.

Se você sente confiante para expandir o projeto e exercitar seus conhecimentos, tente implementar o desafio a seguir.

Modifique o método **Insert()** da classe **ProdutoRepository** para que, em vez de receber dois parâmetros separados, receba um parâmetro do tipo **Produto**. Use as propriedades desse objeto para montar a cláusula SQL. Em **Produto.Main()**, monte um objeto **Produto** a partir de valores informados pelo usuário e informe esse objeto como argumento na chamada a **Insert()**.

Organizando os registros: pesquisa e ordenação

Um dos assuntos mais clássicos em algoritmos é a ordenação de dados, definido formalmente como: *dada uma sequência de elementos* $<a_1, a_2, ..., a_n>$*, uma ordenação resulta na permutação (ou reordenação)* $<a'_1, a'_2, ..., a'_n>$ *dos elementos de entrada de maneira que* $a'_1 \leq a'_2 \leq ... \leq a'_n$.

Geralmente, uma ordenação se aplica a elementos de uma lista ou vetor e pode se aplicar a números, caracteres ou outros tipos de dados. Trata-se de uma operação necessária em diversos processamentos. No nosso caso em específico, gostaríamos de mostrar os registros de **produto** em ordem alfabética de nome.

Para fazer isso usando a lista que recuperamos do método **Query()**, poderíamos usar uma estratégia de percorrer os itens da lista e trocar posições entre eles de acordo com sua ordem. Há alguns algoritmos clássicos que cuidam desse assunto e que poderíamos implementar em nossa lista.

Bubble sort

Esta é uma implementação pouco eficiente, em que cada elemento da lista é comparado com cada um dos elementos anteriores na lista. Caso o elemento anterior seja maior, os dois elementos trocam seus valores entre si. Leva este nome em razão de sua característica de fazer o maior valor “subir” como bolha.

Figura 7 – Esquematizando a execução do Bubble sort

Ordenação em três passos. No primeiro, há o vetor {4,9,5,1}, em que na primeira iteração compara-se 4 com 9 e não troca; na segunda, compara-se 5 com 9 e troca, pois 5 é menor; na terceira, compara 1 com 9 e troca, resultando em {4,5,1,9}. No passo 2, reinicia o processo: compara 5 com 4 e não troca; compara 1 com 5 e troca. Consolida-se assim o 5 na penúltima posição, resultando em {4, 1, 5, 9}. Por fim, compara-se 1 com 4 e troca, pois 1 é menor. Resultado final {1, 4, 5, 9}.

Merge sort

A técnica de ordenação divide o vetor em pedaços menores, até que restem apenas dois elementos que são ordenados. Depois, os fragmentos são reunidos em um único vetor ordenado.

Figura 8 – Esquematização do Merge sort.

inicia com vetor {38,27,43,3,9,82,10}. Quebra em {38,27,43,3} e {9,82,10}; que são quebrados em {38,27}, {43,3}, {9,82} e {10}; que são reordenados como {27,38}, {3,43}, {9,82}, {10}; que são combinados e reordenados como {3, 27, 38, 43} e {9, 10, 82}; que são combinados e reordenados como {3, 9, 10, 27, 38, 43, 82}.

Quick sort

É uma solução que também subdivide o problema, como o Merge sort. Escolhe-se um item da lista como um pivô. Depois, se divide a lista entre os menores que o pivô e os maiores que o pivô, em uma operação que se chama partição. Depois, realiza-se o processo de partição em cada subgrupo menor, finalizando com a lista ordenada. É o método mais rápido computacionalmente.

Figura 9 – Quick sort em ação

lista {9, -3, 5, 2, 6, 8, -6, 1, 3}. Pivô: -3, divide em menores que 3 {-3, 2, -6, 1} e maiores que 3 {8, 5, 9 e 6}. Aplica-se a mesma técnica nas subdivisões usando como pivô da primeira sublista 1 (resulta em {-3, -6} e {2}) e na seguda o valor 6 (resulta em {5} e {9, 8}). As sublistas são reordenadas terminando o algoritmo com {-6, -3, 1, 2, 3, 5, 6, 8, 9}.

Na realidade, trata-se de uma área de estudo muito desenvolvida em algoritmos. No entanto, não precisamos implementar todo um código específico para esses processos pois dois motivos, basicamente: pois temos disponíveis métodos e classes em C# que já fazem isso por nós; e (principalmente) porque estamos trabalhando com SQL.

Uma das vantagens de se usar banco de dados é que podemos ordenar, reordenar e combinar informações de maneira bem rápida e flexível. No caso da ordenação, tudo o que precisamos é de uma expressão Order By.

SELECT * FROM produto ORDER BY nome;

Usando essa consulta, conseguimos trazer do banco de dados as informações de produtos já ordenadas de acordo com a coluna **nome**, e não precisamos de mais nenhum trabalho no código C# para reordenar.

Vamos aplicar essa alteração em **ProdutoRepository**, no método **Query()**.

```
string sqlSelect = "SELECT * FROM produto ORDER BY nome";
```

Mostramos acima apenas o trecho da variável **SQL Select** no método **Query()** (está aproximadamente na linha 30 no código), porque realmente é a única coisa que precisamos fazer. Este é um processo bem simples, como foi possível perceber.

Ao testar, o resultado deve ser algo similar ao seguinte:

```
Id 4; Produto: apontador; Fabricante: labra; Preco: 1,99; Disponível: Tru
e; Data Cadastro:26/08/2019 16:12:14
Id 3; Produto: caneta; Fabricante: bic; Preco: 1,99; Disponível: True; Da
ta Cadastro:26/08/2019 14:49:03
Id 1; Produto: prodTeste; Fabricante: fabrTeste; Preco: 1,99; Disponível:
True; Data Cadastro:20/08/2019 10:03:22
Id 2; Produto: produto 1; Fabricante: fabricante 1; Preco: 1,99; Disponív
el: True; Data Cadastro:20/08/2019 10:32:50
```

Provavelmente, os seus dados estão diferentes desses do exemplo, mas é importante notar que os registros vieram em ordem crescente no que se refere ao nome do produto.

Também podemos utilizar o SQL para flexibilizar nossa aplicação. Por exemplo, se quisermos pesquisar pelas iniciais de um produto, podemos usar uma cláusula where e uma comparação Like.

SELECT * FROM produto WHERE nome LIKE 'prod%' ORDER BY nome

A consulta anterior traz como resultado todos os produtos com nomes que iniciam com “prod”. Podemos implementar isso no nosso código e flexibilizar nosso método **Query()**. Para isso, vamos incluir um parâmetro chamado de **filtroNome**.

```
public List<Produto> Query(string filtroNome)
```

Em seguida, ao montar a cláusula de **select** (variável **SLQ Select**), vamos quebrá-la em etapas. Queremos verificar se o usuário informou um filtro. Em caso positivo, vamos implementar o like; em caso negativo, vamos incluir filtragem no SQL.

```
string sqlSelect = "SELECT * FROM produto";

if(!String.IsNullOrEmpty(filtroNome))
  sqlSelect = sqlSelect + $" WHERE nome like '{filtroNome}%'";

sqlSelect = sqlSelect + " ORDER BY nome";
```

Usamos o método **IsNullOrEmpty()** da classe **String** para verificar se há valor no parâmetro **filtroNome**. Se houver, incluímos um like, concatenando o valor do filtro informado no texto da consulta.

Cuidado com espaços ao concatenar *strings* em uma cláusula SQL. A falta de espaço antes de **ORDER BY nome**, por exemplo, pode resultar na cláusula “SELECT * FROM produtoORDER BY nome”, que é incorreta e gerará falha no programa.

O método **Query()** completo, neste momento, está assim:

```
public List<Produto> Query(string filtroNome)
{
  MySqlConnection conexao = new MySqlConnection(enderecoConexao);
  conexao.Open();

  string sqlSelect = "SELECT * FROM produto";

  if(!String.IsNullOrEmpty(filtroNome))
    sqlSelect = sqlSelect + $" WHERE nome like '{filtroNome}%'";

  sqlSelect = sqlSelect + " ORDER BY nome";

  MySqlCommand comandoQuery = new MySqlCommand(sqlSelect, conexao);
  MySqlDataReader resultado = comandoQuery.ExecuteReader();

  List<Produto> listaProdutos = new List<Produto>();

  while(resultado.Read())
  {
    Produto item = new Produto();
    item.Id = resultado.GetInt32("Id");
    item.Nome = resultado.GetString("Nome");
    item.Fabricante = resultado.GetString("Fabricante");
    item.Preco = resultado.GetDecimal("preco");
    item.Disponivel = resultado.GetBoolean("disponivel");
    item.DataCadastro = resultado.GetDateTime("dataCadastro");

    listaProdutos.Add(item);
  }

  resultado.Close();
  conexao.Close();

  return listaProdutos;
}
```

Para testar, voltamos ao método **Main()** de **Program** e podemos notar que o editor acusa erro na chamada a **pr.Query()**, pois não informamos o parâmetro agora obrigatório. Vamos corrigir isso modificando essa chamada da seguinte maneira:

```
List<Produto> lista = pr.Query("prod");
```

Nesse momento, é a única alteração necessária para essa classe. Podemos executar o comando dotnetdotnet run e rodar a aplicação. O resultado deve ser a lista de produtos que iniciam com “prod”.

Vamos deixar um pouco mais útil essa filtragem? Se estiver confiante, realize o desafio a seguir:

Modifique o programa para que o usuário informe um valor para o filtro, que será repassado como argumento para **pr.Query()**, tornando dinâmica a filtragem por nome.

Atualizando registros

A operação de *update* no SQL segue a linha da operação de **Insert**, pois também não retorna valores. A seguir, vamos implementar rapidamente em nossa aplicação um *update* simples para produto.

A cláusula que precisamos é:

UPDATE produto

SET

nome = <{nome: }>,

fabricante = <{fabricante: }>,

preco = <{preco: }>,

disponivel = <{disponivel: }>,

dataCadastro = <{dataCadastro: }>

WHERE id=<{expr}>;

Nela, precisamos informar valores para os campos sinalizados com os símbolos **<** e **>**.

Em **ProdutoRepository**, vamos incluir um novo método chamado **Update()** com um parâmetro do tipo **Produto**. O corpo do método fica da seguinte forma:

```
public void Update(Produto p)
{
  MySqlConnection conexao = new MySqlConnection(enderecoConexao);
  conexao.Open();

  string sqlUpdate =
    "UPDATE produto " +
    " SET nome = '"+ p.Nome +"', fabricante = '"+ p.Fabricante + "', " +
      " preco = "+ p.Preco.ToString("0.00", System.Globalization.CultureI
nfo.InvariantCulture) +", "+
      " disponivel = " + (p.Disponivel ? 1 : 0) +
    " WHERE id="+ p.Id ;

  MySqlCommand comando = new MySqlCommand(sqlUpdate, conexao);
  comando.ExecuteNonQuery();

  conexao.Close();
}
```

Note que na cláusula SQL, omitimos **id** e **dataCadastro** da porção **SET**. **Id**, por ser chave primária, não deve ser alterado; e **dataCadastro**, por seu significado coluna, também não. Para a coluna **disponivel**, utilizamos um operador ternário (se **p.Disponivel** é verdadeiro, então o valor no SQL será 1; caso contrário, o valor será 0).

O método **ToString()** aplicado na propriedade **Preco** traz os argumentos **0.00**, que indicam o formato numérico desejado e o valor **System.Globalization.CultureInfo.InvariantCulture**. Usamos isso para que o valor decimal seja escrito sem vírgula na cláusula SQL – a vírgula gerará erro.

Vamos testar e, para isso, precisamos incluir um novo *case* dentro do *switch* do método **Main()** de **Program**. Antes do item **default**, inclua o seguinte:

```
case "A":
  prod = new Produto();
  Console.Write("Digite o id do produto que será atulizado: ");
  prod.Id = int.Parse(Console.ReadLine());

  Console.Write("Digite o novo nome do produto: ");
  prod.Nome = Console.ReadLine();

  Console.Write("Digite o novo fabricante do produto: ");
  prod.Fabricante = Console.ReadLine();

  Console.Write("Digite o novo preco do produto: ");
  prod.Preco = Decimal.Parse(Console.ReadLine());

  Console.Write("O produto está disponível [s/n]: ");
  prod.Disponivel = (Console.ReadLine().ToLower() == "s");

  pr.Update(prod);
  break;
```

O objeto **prod** foi declarado antes do *switch*.

```
Produto prod;
```

Para declarar variáveis ou objetos dentro de um trecho *case*, precisamos usar chaves: uma **{** após o *case* e uma **}** antes do *break*.

Também trocamos a instrução ao usuário:

```
Console.Write("Digite C para cadastrar, L para listar, A para atualizar:"
);
```

O código final de **Program.cs** fica assim:

```
using System;
using System.Collections.Generic;

namespace CadProdutos
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Cadastrando Produto");
      Console.Write("Digite C para cadastrar, L para listar, A para atual
izar:");
      string comando = Console.ReadLine();
      ProdutoRepository pr = new ProdutoRepository();
      string nome, fabricante;
      Produto prod;

      switch (comando.ToUpper())
      {
        case "C": //código insert sem completar o desafio proposto
          Console.Write("Digite o nome do produto: ");
          nome = Console.ReadLine();

          Console.Write("Digite o fabricante do produto: ");
          fabricante = Console.ReadLine();

          pr.Insert(nome, fabricante);
          break;

        case "L":
          List<Produto> lista = pr.Query("prod");
          foreach (Produto p in lista)
          {
            Console.WriteLine(p.ToString());
          }
          break;

        case "A":
          prod = new Produto();
          Console.Write("Digite o id do produto que será atulizado: ");

          prod.Id = int.Parse(Console.ReadLine());

          Console.Write("Digite o novo nome do produto: ");
          prod.Nome = Console.ReadLine();

          Console.Write("Digite o novo fabricante do produto: ");
          prod.Fabricante = Console.ReadLine();
```

```
            Console.Write("Digite o novo preco do produto: ");
            prod.Preco = Decimal.Parse(Console.ReadLine());

            Console.Write("O produto está disponível [s/n]: ");
            prod.Disponivel = (Console.ReadLine().ToLower() == "s");

            pr.Update(prod);
            break;

          default:
            Console.WriteLine("Comando inválido");
            break;
        }
      }
    }
}
```

O código de **ProdutoRepository** ficou assim:

```
using System;
using System.Collections.Generic;
using MySql.Data.MySqlClient;

namespace CadProdutos
{
  public class ProdutoRepository
  {
    private const string enderecoConexao = "Database=admprodutos;Data Sou
rce=localhost;User Id=root;";
    public void Insert(string nome, string fabricante)
    {
      MySqlConnection conexao = new MySqlConnection(enderecoConexao);
      conexao.Open();

      string sqlInsert =
        "INSERT INTO produto (nome, fabricante, preco, dataCadastro, disp
onivel)" +
        "VALUES ('" + nome + "', '" + fabricante + "', 1.99, NOW(), 1)";
      MySqlCommand comando = new MySqlCommand(sqlInsert, conexao);
      comando.ExecuteNonQuery();

      conexao.Close();
    }

    public List<Produto> Query(string filtroNome)
    {

      MySqlConnection conexao = new MySqlConnection(enderecoConexao);
      conexao.Open();

      string sqlSelect = "SELECT * FROM produto";

      if (!String.IsNullOrEmpty(filtroNome))
        sqlSelect = sqlSelect + $" WHERE nome like '{filtroNome}%'";

      sqlSelect = sqlSelect + " ORDER BY nome";
      MySqlCommand comandoQuery = new MySqlCommand(sqlSelect, conexao);
      MySqlDataReader resultado = comandoQuery.ExecuteReader();

      List<Produto> listaProdutos = new List<Produto>();

      while (resultado.Read())
      {
        Produto item = new Produto();
        item.Id = resultado.GetInt32("Id");
        item.Nome = resultado.GetString("Nome");
        item.Fabricante = resultado.GetString("Fabricante");
```

```
        item.Preco = resultado.GetDecimal("preco");
        item.Disponivel = resultado.GetBoolean("disponivel");
        item.DataCadastro = resultado.GetDateTime("dataCadastro");
        listaProdutos.Add(item);
      }

      resultado.Close();
      conexao.Close();

      return listaProdutos;
    }

    public void Update(Produto p)
    {
      MySqlConnection conexao = new MySqlConnection(enderecoConexao);
      conexao.Open();

      string sqlUpdate =
        "UPDATE produto " +
        " SET nome = '" + p.Nome + "', fabricante = '" + p.Fabricante +
"', " +
          " preco = " + p.Preco.ToString("0.00", System.Globalization.Cul
tureInfo.InvariantCulture) + ", " +
          " disponivel = " + (p.Disponivel ? 1 : 0) +
        " WHERE id=" + p.Id;
      MySqlCommand comando = new MySqlCommand(sqlUpdate, conexao);
      comando.ExecuteNonQuery();

      conexao.Close();
    }
  }
}
```

Testando com dotnet run, podemos informar valores e, se tudo der certo, os valores das colunas serão atualizados. A seguir, veja um exemplo de suas execuções consecutivas da aplicação.

```
PS C:\Meus Projetos VS\CadProdutos> dotnet run
Cadastrando Produto
Digite C para cadastrar, L para listar, A para atualizar:A
Digite o id do produto que será atulizado: 1
Digite o novo nome do produto: prod Updated
Digite o novo fabricante do produto: fabr Updated
Digite o novo preco do produto: 190,50
O produto está disponível [s/n]: n
PS C:\Meus Projetos VS\CadProdutos> dotnet run
Cadastrando Produto
Digite C para cadastrar, L para listar, A para atualizar:L
Id 1; Produto: prod Updated; Fabricante: fabr Updated; Preco: 190,50; Dis
ponível: False; Data Cadastro:20/08/2019
10:03:22
Id 2; Produto: produto 1; Fabricante: fabricante 1; Preco: 1,99; Disponív
el: True; Data Cadastro:20/08/2019 10:32:50
```

Vemos pela listagem que nosso produto de **Id 1** está atualizado com os valores informados.

Você sabe qual operação está faltado? Se estiver disposto a completar o projeto, faça o exercício a seguir:

Implemente a operação de **Delete** no sistema **CadProdutos**. A sintaxe da cláusula é DELETE FROM produto WHERE id=<{id}>;.

Em **ProdutoRepository**, será necessário criar um novo método, que ficará semelhante ao **Insert()** e ao **Update()**. Na classe **Program**, no método **Main()**, inclua uma nova opção ao usuário para que ele possa informar um **id** e excluir o registro correspondente.

Funções, procedimentos e recursividade

Vamos aproveitar agora nossas experiências com projetos do tipo console para exercitar alguns conceitos novos e alguns já conhecidos: funções, procedimentos e recursividade.

Revisão de conceito

- Funções são sub-rotinas, algoritmos autocontidos e que podem ser invocados por outros trechos de algoritmo, que, ao finalizar sua execução, retornam um valor.
- Procedimentos são sub-rotinas que não retornam valor.
- Ambos, em orientação a objetos, são chamados de métodos e são definidos dentro de classes.
- Funções e procedimentos podem receber entradas de dados na forma de parâmetros.

Até aqui, já utilizamos métodos de várias maneiras. Alguns retornaram valores (como o método **Query()** de **ProdutoRepository**), mas outros, não (como **Insert()**). Métodos são uma sequência de códigos que, convenientemente, podemos invocar em outro trecho de código. Também podemos dizer que são uma das ferramentas essenciais da orientação a objetos, pois é por meio deles que os objetos se comunicam.

Um conceito que ainda não experimentamos é a recursividade. Diz-se que um método é recursivo quando ele invoca a si mesmo. Esse tipo de chamada é muito útil em algumas tarefas repetitivas e, geralmente, substituem um laço de repetição mais complexo. No exemplo, veja um método recursivo que calcula o fatorial de um número:

```
public static float CalculaFatorial(int n)
{
  if(n == 0){
    return 1;
  }
  else
  {
    return n * CalculaFatorial(n - 1);
  }
}
```

Crie um projeto console e faça o teste incluindo o método anterior, invocando-o em **Main()**, como a seguir:

```
Console.WriteLine("Fatorial: " + CalculaFatorial(4));
```

Vamos ver como funciona:

- O método **CalculaFatorial** é invocado com o valor 4. Como o valor é diferente de zero, o código vai para o **else** e executa quatro vezes o resultado de **CalculaFatorial(3)**
- **CalculaFatorial(3)** inicia e também cai no **else**, executando três vezes **CalculaFatorial(2)**
- **CalculaFatorial(2)** inicia e também cai no **else**, executando duas vezes **CalculaFatorial(1)**
- Analogamente, **CalculaFatorial(1)** é invocado e também cai no **else**, já que é diferente de zero, executando 1uma vez **CalculaFatorial(0)**
- **CalculaFatorial(0)** executa, mas cai no **if**, retornando o valor 1
- A chamada **CalculaFatorial(1)** aguardava o valor de **CalculaFatorial(0)**. Agora, ela conclui a operação 1 vezes 1 e retorna
- **CalculaFatorial(2)**, com o valor retornado do cálculo de 1, retorna o valor de 2 vezes 1
- **CalculaFatorial(3)**, que aguardava a conclusão de **CalculaFatorial(2)**, agora consegue concluir sua operação: 3 vezes 2, retornando o valor 6
- Por fim, **CalculaFatorial(4)**, com o valor retornado de **CalculaFatorial(3)**, consegue concluir a operação fazendo 4 vezes 6: o valor final é 24

A execução é estranha à primeira vista, mas obedece a uma ordem de empilhamento: as primeiras chamadas à função aguardam a execução da próxima, que aguarda a próxima e assim por diante. É de extrema importância implementar uma condição de parada, senão o código executará indefinidamente. No exemplo, a condição de parada é if(n==0), pois nesse caso não ocorre nova chamada à função.

Os algoritmos de ordenação **Merge sort** e **Quick sort**, vistos anteriormente, são usualmente implementados utilizando recursividade.

Tente fixar seu conhecimento com o exercício proposto.

Faça um método recursivo para somar todos os elementos de um vetor. O vetor é passado por parâmetro, assim como o índice. A soma se dá por **vetor[n-1] + vetor[n-2]+...+vetor[0]**, em que **n** é o tamanho do vetor.

Até aqui, foi possível relembrar e reforçar alguns conceitos de programação que permeiam nosso trabalho desde as UCs iniciais. Aplicamos a isso a conectividade com banco de dados, utilizando a biblioteca de conexão do MySQL para .NET Core.

Podemos (e vamos) aplicar esses conceitos a projetos *web*. A ideia de uma classe de repositório e outra de modelo é bastante conveniente para arquitetura MVC. É claro que você já pode se adiantar e adaptar o que aprendeu e desenvolveu aqui a um projeto ASP.NET Core MVC.