Aula 12 - Design Avançado de Objetos

Docupedia Export

Author:Balem Luis (CtP/ETS)
Date:14-Apr-2023 17:43

Table of Contents

- Namespaces, Projetos, Assemblies, Arquivos e Organização
- Usings Globais
- Classes Estáticas
- Enums
- Tratamento de Erros
- Bibliotecas de Classe
- Interfaces
- Exemplo 3
- Exercícios Propostos

1 Namespaces, Projetos, Assemblies, Arquivos e Organização

Você pode organizar seu código C# de muitas formas e subdividi-lo como quiser, mas alguns padrões podem ser respeitados para melhorar a sua experiência e deixar seu trabalho mais produtivo. Nesta secção aprenderemos um pouco sobre como fazer isso. Como background iremos fazer uma aplicação para um sistema escolar. Ou seja, um sistema onde o usuário poderá cadastrar alunos, professores, disciplinas e turmas(que são vários alunos cursando uma disciplina dada por um professor). Como não queremos perder os dados quando a aplicação fecha, salvaremos os dados em arquivos e assim produziremos uma biblioteca que faz isso.
No C# podemos separar nossas implementações em vários **Projetos**. Um projeto é basicamente uma pasta com um arquivo de extensão '.csproj' em estrutura XML com tags como pode ser visto a seguir:

nomedoprojeto.csproj

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project
```

Dentro de 'PropertyGroup' temos algumas configurações importantes:

- OutputType: Diz qual a saída do projeto. Exe é um executável, ou seja, um projeto que pode abrir e executar no seu projeto. Existem projetos do tipo DLL, que são bibliotecas que podem ser usadas em outros projetos, mas não podem ser executadas. Nesse caso, essa configuração não é necessária.
- TargetFramework: Diz a respeito de qual versão do .Net Framework você está utilizando.
- Nullable: Se habilitada, várias operações que podem resultar em um NullPointException resultam em uma Warning, um aviso de que um erro pode ocorrer ali.

Existem muitas outras opções que veremos no futuro.
Um projeto quando executado gera um **Assembly** que é um objeto de código que pode ser utilizado por outros ou executado. Tudo que você fizer em um projeto estará incluso no mesmo assembly e você pode importar vários assemblies em um mesmo projeto como veremos mais tarde nesta aula.
Como você já sabe, quando executado um projeto buscamos a função Main/Arquivo Top-Level e o executamos. Porém, é completamente possível termos muitos arquivos no mesmo projeto. Para usar um arquivo no outro você não precisa fazer nada, só usar os elementos de um arquivo no outro. Observe:

Aluno.cs

```
public class Aluno
```

```
{
    public int Matricula { get; set; }
    public string Nome { get; set; }
}
```

Program.cs

```
Aluno aluno = new Aluno();

aluno.Matricula = 4;
aluno.Nome = "Gilmar";
```

Em geral, toda vez que você cria uma classe diferente você cria em um diferente arquivo. Isso ajuda a você encontrar as implementações mais rápido, reduz os conflitos ao usar o Github e deixa o projeto mais organizado. Algumas vezes você pode usar pastas para organizar ainda mais o projeto. Ao adicionar uma pasta, ela não muda em nada a forma de utilizar as classes. Por exemplo, se Aluno estivesse em uma pasta Modelos, nada impediria de utilizar no arquivo 'Program' sem nenhuma ação adicional necessária. Porém, ao separar os documentos em pastas, em geral, é comum o uso de **Namespaces**. Um Namespace é uma estrutura de código que permite você esconder classes que só poderão ser utilizadas se importadas em outros arquivos. Usar um namespace é bem fácil:

```
public class Classe1 { }

namespace Namespace1
{
    public class Classe2 { }

    namespace Namespace2
    {
        public class Classe3 { }
    }

    namespace Namespace3.Namespace4
    {
        public class Classe4 { }
    }
}

namespace Namespace1.Namespace2
{
    public class Classe5 { }
```

```
21  }
```

Usamos a palavra reservada 'using' para acessar os conteúdos fora do Namespace onde ela foi criada. Por exemplo, se estamos fora do Namespace1 você precisa importá-lo para usar a Classe2. Abaixo você verá 5 exemplos de uso da classe Program usando o código acima, e como se trabalha corretamente com os Namespaces. Relembrando, apesar do exemplo caótico, em geral você só usa um Namespace específico dentro de pastas em um projeto. Veremos como isso se comporta no futuro.

```
1  using Namespace1;
2
3  Classe1 obj1 = new Classe1();
4  Classe2 obj2 = new Classe2();
```

```
1  using Namespace1.Namespace2;
2
3  Classe1 obj1 = new Classe1(); // OK
4  Classe2 obj2 = new Classe2(); // Error
5  Classe3 obj3 = new Classe3(); // OK
```

```
1  using Namespace1.Namespace2;
2  using Namespace1.Namespace3.Namespace4;
3
4  Classe3 obj3 = new Classe3();
5  Classe4 obj4 = new Classe4();
6  Classe5 obj5 = new Classe5();
```

```
1  Classe2 test = new Classe2(); // Error
2
3  namespace Namespace1
4  {
5      public class Test
6      {
7          public static void Main()
8          {
9              Classe2 obj2 = new Classe2(); // OK, Classe2 e este código estão dentro do Namespace1
10         }
11     }
```

```
}
```

```
using Namespace1.Namespace3.Namespace4;

namespace Namespace1; // Todo código abaixo está no Namespace1

using Namespace2; // Não precisa importar Namespace1.Namespace2 pois já estamos no Namespace1

public class Test
{
    public static void Main()
    {
        Classe1 obj1 = new Classe1();
        Classe2 obj2 = new Classe2();
        Classe3 obj3 = new Classe3();
        Classe4 obj4 = new Classe4();
        Classe5 obj5 = new Classe5();
    }
}
```

2 Usings Globais

Você também pode utilizar usings globais. Ao usar uma using global em qualquer arquivo a using será válida para todos os arquivos. Por exemplo, se você tivesse o seguinte arquivo em sua pasta de projeto:

Usings.cs

```
1  global using Namespace1;
2  global using Namespace1.Namespace2;
3  global using Namespace1.Namespace3.Namespace4;
```

Você poderia acessar as classes Classe1 a Classe5 sem problema algum em qualquer arquivo, mesmo que não fosse o Usings.cs. Em geral nos arquivos de configuração .csproj, uma configuração vem comumente adicinoada:

nomedoprojeto.csproj

```
1  <Project Sdk="Microsoft.NET.Sdk">
2
3    <PropertyGroup>
4      <OutputType>Exe</OutputType>
5      <TargetFramework>net7.0</TargetFramework>
6      <ImplicitUsings>enable</ImplicitUsings>
7      <Nullable>enable</Nullable>
8    </PropertyGroup>
9
10 </Project
```

Este 'ImplictUsings' que vem como 'enable' pede a criação de um arquivo de usings globais na pasta 'obj'. Como ela está no seu projeto será válido para todo ele. Este arquivo comumente vem assim:

obj/NomeDoProjeto.GlobalUsings.g.cs

```
1  // <auto-generated/>
2  global using global::System;
3  global using global::System.Collections.Generic;
4  global using global::System.IO;
5  global using global::System.Linq;
```

```
6  global using global::System.Net.Http;
7  global using global::System.Threading;
8  global using global::System.Threading.Tasks;
```

O '.g' na extensão significa a mesma coisa que o comentário no inicio do arquivo, que aquele documento foi gerado automaticamente. Além disso, como você pode ver, a palavra global tem duas funcionalidades. Ela é reutilizada antes do System nos exemplos. Ela significa que você deve importar o System do namespace global, ou seja, não confundir com um declarado por um usuário. Veja:

```
1  namespace System
2  {
3      public class ClasseA
4      {
5          public void Test()
6          {
7              Console.WriteLine("Classe A");
8          }
9      }
10 }
11
12 namespace MyProduct
13 {
14     public class ClasseB
15     {
16         public void Test()
17         {
18             // Console.WriteLine("Classe B"); Erro
19             global::System.Console.WriteLine("Classe B");
20         }
21     }
22
23     namespace System
24     {
25         public class ClasseC
26         {
27             public void Test()
28             {
29                 // Console.WriteLine("Classe C"); Erro
30                 global::System.Console.WriteLine("Classe C");
31             }
```

```
32              }
33          }
34      }
```

Isso é especialmente útil quando você quer gerar código e não quer ter problemas com código feito pelo usuário. Além disso, você pode ver que a Classe A não teve problemas com isso já que ela está dentro do System.

3 Classes Estáticas

Em algum momento deste curso você pode ter se perguntado o porquê da palavra static usada para importar o namespace System.Console. Ela só foi usada neste caso. Outra pergunta é como WriteLine é usado sem que precisemos de um objeto, afinal de contas C# é orientado a objetos e não deveria ter funções perdidas por ai. Todas essas perguntas serão respondidas agora. O que estamos presenciando não é o namespace System.Console mas sim a classe estática Console dentro do namespace System. Você poderia ter usado, em qualquer momento, está versão:

```
1  using System;
2
3  Console.WriteLine("Olá mundo");
```

Ou seja, você utilizou de uma classe sem instanciá-la, sem criar um objeto. Isso acontece por que Console sendo uma classe estática não pode ser estanciada, mas tudo que tem nela está livre para ser acessada como se Console fosse um único objeto definido globalmente. Você já deve ter visto que a função Main também é estática. Funções, campos e propriedades, todos esses podem ser estáticos e pertencer a classes estáticas ou não. Vamos ver alguns exemplos úteis de Design usando classes, propriedades, campos, construtores e métodos estáticos.

```
1  using System;
2
3  // Sua própria classe de Console personalizada
4  public static class MyConsole
5  {
6      public int? ReadLineInt()
7      {
8          var str = Console.ReadLine();
9
10         // Um código novo para os aventureiros:
11         // Se a conversão é bem sucedida, cria um int i e joga o valor convertido para fora da função
12         // TryParse usando a palavar-reservada out. A palavra ainda retorna verdadeiro. Caso contrário
13         // nenhum erro estoura e retorna falso
14         if (int.TryParse(str, out int i))
15             return i;
16
17         return null;
18     }
19
20     public void Print(object obj)
21     {
22         // Usando a conversão para string que todo objeto tem
23         var str = obj.ToString();
```

```
24          Console.WriteLine(str);
25      }
26  }
```

Usando a classe MyConsole sem using estática:

```
1   MyConsole.Print("Digite um número");
2   int? a = MyConsole.ReadLineInt();
3
4   MyConsole.Print("Digite outro número");
5   int? b = MyConsole.ReadLineInt();
6
7   if (a is null || b is null)
8       MyConsole.Print("Números inválidos.");
9   else MyConsole.Print(a + b);
```

Usando a classe MyConsole com using estática:

```
1   using static MyConsole;
2
3   Print("Digite um número");
4   int? a = ReadLineInt();
5
6   Print("Digite outro número");
7   int? b = ReadLineInt();
8
9   if (a is null || b is null)
10      Print("Números inválidos.");
11  else Print(a + b);
```

Outro uso interessante para classes estáticas é tornar certos valores globais:

```
1   public static class GameConfiguration
2   {
3       public static bool AutoSave { get; set; }
4       public static bool SoundOn { get; set; }
5       public static string MenuButton { get; set; }
6
7       // Construtor estático, usando uma vez, quando você usa a classe GameConfiguration pela primeira vez
8       static GameConfiguration()
```

```
    {
        // Procura em algum arquivo a configuração salva
        // Caso não achar, inicializa normalmente
        AutoSave = true;
        SoundOn = true;
        MenuButton = "Escape";
    }
}
```

Você ainda pode ter classes instanciáveis com membros estáticos:

```
using static System.Random;

public class NPC
{
    public string Nome { get; set; }
    public int Vida { get; set; }
    public int Dinheiro { get; set; }

    public NPC()
    {
        Count++;
    }

    public static int Count { get; private set; } = 0;

    public static NPC CreateRandom()
    {
        NPC npc = new NPC();

        // A partir do .NET 6 a classe Random tem uma propriedade estática chamada Shared. Ela cria um objeto
        // da classe Random que você reutiliza em toda aplicação. Assim importando com uma using estática a
        // classe Random, nós podemos usar 'Shared' como se fosse uma variável local.
        npc.Vida = Shared.Next(0, 100);
        npc.Dinheiro = Shared.Next(0, 1000);
        var nomes = new string[] { "Gilmar", "Pamella", "Xispita" };
        npc.Nome = nomes[Shared.Next(0, 3)];

        return npc;
```

```
29          }
30      }
```

4 Enums

Antigamente todos os parâmetros de funções C eram números quando se tratava de configurações. Por exemplo, caso você utilizasse uma função deveria mandar 0 para configuração X e 1 para configuração Y. Até existiam formas de contornar esse uso escondido das coisas, mas tinham seus problemas. Para não cair no mesmo problema, o C# utiliza-se do Enum, uma estrutura para mascarar opções. Observe:

```
1  public enum DiasDaSemana
2  {
3      Domingo,
4      Segunda,
5      Terça,
6      Quarta,
7      Quinta,
8      Sexta,
9      Sábado
10 }
```

Sua utilização é fácil:

```
1  var dia = DiasDaSemana.Quarta;
2
3  Console.WriteLine("Que semana!");
4  if (dia == DiasDaSemana.Quarta)
5      Console.WriteLine("Mas ainda é quarta-feira!");
```

Você ainda pode atribuir valores e definir tipos para um enum:

```
1  using System;
2
3  Key key1 = Key.Ctrl;
4  Key key2 = (Key)(2); // C
5  Key key = key1 | key2; // União de Ctrl e C
6
7  // Código complexo abaixo, analisar com cuidado
8  if ((key & Key.C) > 0 && (key & Key.Ctrl) > 0)
9      Console.WriteLine("Copiar");
10 else if ((key & Key.V) > 0 && (key & Key.Ctrl) > 0)
11     Console.WriteLine("Colar");
12
13 public enum Key : byte
```

```
14  {
15      Ctrl = 1,
16      C = 2,
17      V = 4
18  }
```

5 Tratamento de Erros

No C# podemos tratar erros de forma bem interessante. Para isso usamos os blocos try, catch e finally:

```
try
{
    // Se um erro acontecer aqui
}
catch (Exception ex)
{
    // Esse código é executado, mas a aplicação não para ou simplesmente 'morre'
}
finally
{
    // Mas isso é sempre executado, dando erro ou não.
}
```

Para lançar uma exceção basta usar a palavra reservada throw seguido de um objeto de uma classe que herde ou seja a System.Exception. Isso significa que você pode fazer suas próprias exceções:

```
using System;
using static System.Console;

string nome = "Nome não encontrado...";

try
{
    string[] nomes = new string[] { "Gilmar", "Pamella", "Erro", "Erro", "Outro Erro", "Outro Erro" };
    int index = Random.Shared.Next(8);
    nome = nomes[index]; // Podemos ter um IndexOutOfRangeException aqui

    if (nome == "Erro")
        throw new MyException();

    if (nome == "Outro Erro")
        throw new MyOtherException(index.ToString());
}
catch (IndexOutOfRangeException ex) // Trata apenas IndexOutOfRangeException
{
```

```
20      WriteLine("O número aleatório foi muito grande!");
21  }
22  catch (MyException ex) // Trata apenas MyException
23  {
24      WriteLine(ex);
25  }
26  catch (MyOtherException ex) when (ex.Info == "4") // Trata apenas MyOtherException quando Info é 4
27  {
28      WriteLine(ex);
29  }
30  catch (Exception ex) // Trata qualquer outro erro
31  {
32      WriteLine("Erro desconhecido!");
33  }
34  finally
35  {
36      WriteLine(nome);
37  }
38
39  public class MyException : Exception
40  {
41      public override string Message => "Deu um grande e catastrófico erro!";
42  }
43
44  public class MyOtherException : Exception
45  {
46      public string Info { get; set; }
47      public MyOtherException(string info)
48          => this.Info = info;
49
50      public override string Message => $"Falha por motivos de {Info}!";
51  }
```

Em geral você lançará erros para indicar qual foi o tipo de falha para seu usuário (usuário esse que pode ser outros programadores que usam suas classes ou até mesmo você) ao invés de uma mensagem de um erro de uma linha específica. Tratar erros é importante para evitar que a aplicação pare de rodar sem motivo algum.

6 Bibliotecas de Classe

Para criar nossa biblioteca de classe basta usar dotnet new classlib. Assim criaremos um projeto que não pode ser executado. No exemplo o final desta aula usaremos uma biblioteca de classe para implementar códigos para usar arquivos do computador como uma espécie de 'banco de dados' para nossa aplicação de sistema Escolar.

7 Interfaces

Talvez a parte mais complexa desta aula sejam elas: As Interfaces (não tem nada de interface gráfica aqui, ok?). Interfaces são como classes abstratas em sua funcionalidade. Não podem ser estanciadas e servem como base para outros objetos. A diferença fundamental é que interface é uma espécie de contrato. Você não herda de uma interface, você implementa uma interface. Você atende o que ela pede para que você possa passar o objeto para algumas funções. Dito isso, uma classe pode implementar quantas interfaces quiser. Interfaces não podem ter implementações (isso pode mudar nas próximas versões do C#, mas então teremos implementações padrões, e não implementações internas da interface). Interfaces não podem declarar variáveis ou mudar o estado de quem a implementa. Por isso, interfaces são bem diferentes, menos impactantes na estrutura de um programa e mais leves para o design orientado a objetos. Veja um simples exemplo antes de aplicarmos ele no nosso exemplo:

```
1  operate(new Sum(), 1, 2);
2  operate(new Sub(), 10, 5);
3
4  float operate(Operation op, float a, float b)
5      => op.GetResult(a, b);
6
7  public class Sum : Operation
8  {
9      public float GetResult(float a, float b)
10         => a + b;
11 }
12
13 public class Sub : Operation
14 {
15     public float GetResult(float a, float b)
16         => a - b;
17 }
18
19 public interface Operation
20 {
21     float GetResult(float a, float b);
22 }
```

Alguns comentários importantes: Primeiramente é difícil ver a utilidade de interfaces quando já se tem herança. No começo é difícil usar corretamente também, não se preocupe tanto com isso. Porém, você vai perceber que interfaces acabam representando mais uma pequena característica de um objeto do que ele como um todo (diferente do exemplo acima). Por exemplo, a Interface IDisposable (em C# usamos a letra I na frente das interfaces para diferenciá-las mais facilmente), diz que a classe tem a função Dispose que libera recursos/memória. Várias classes que herdam e são outras classes acabam por implementar o IDisposable. Essa interface apenas diz que estamos lidando com um recurso que pode liberar memória, sendo apenas uma pequena característica do que o objeto é como um todo. Muito

embora, se use bastante as interfaces no lugar da classe abstrata - isso se faz mais quando não precisamos de uma implementação por baixo dos panos como funções protegidas e afins e não precisamos declarar estado das classes, funcionando apenas como um comportamento único e direto.
Outro fator importante é que interfaces podem ser até mesmo genéricas, sendo ferramentas interessantes para algumas aplicações. Nas aulas 13 em diante usaremos bastante esses recursos.

8 Exemplo 3

Por fim, vamos ao nosso exemplo de sistema escolar. Vamos começar estruturando o projeto em uma pasta com os seguintes comandos:

```
mkdir Front
mkdir Model
mkdir DataBase

cd Database
dotnet new classlib
cd..

cd Model
dotnet new classlib
dotnet add reference ..\DataBase\DataBase.csproj
cd ..

cd Front
dotnet new console
dotnet add reference ..\Model\Model.csproj
```

Neste código acima criamos três pastas para separar o trabalho em 3 projetos. Note que isso não é realmente necessário, mas separar em 3 projetos torna mais fácil o reaproveitamento de cada um deles já que gerarão dlls separadas. Iremos retirar as configurações de Nullable e ImplicitUsings dos três projetos para ter que escrever as usings e não ter warnings confusas do Nullable. Após isso, por exemplo, o csproj do front deve ficar assim:

Front\Front.csproj

```
1  <Project Sdk="Microsoft.NET.Sdk">
2
3    <ItemGroup>
4      <ProjectReference Include="..\Model\Model.csproj" />
5    </ItemGroup>
6
7    <PropertyGroup>
8      <OutputType>Exe</OutputType>
9      <TargetFramework>net6.0</TargetFramework>
10   </PropertyGroup>
11
```

```
12  </Project>
```

O ".." volta uma pasta, da Front para pasta do projeto, assim podemos ver a Model e adicioná-la. No Front a interface com o usuário, o Model os modelos que fazem sentido para a aplicação (mas podem ser utilizadas em outras aplicações também) e por fim, o DataBase é a lógica de conexão com o banco (arquivos de texto) que pode ser reaproveitada várias vezes.

DataBase/DataBaseObject.cs

```
1   namespace DataBase;
2
3   public abstract class DataBaseObject
4   {
5       // Só pode ser visto dentro da bilbioteca DB e por classes quer herdam está fora de DataBaseObject
6       // C# tem várias modificadores de acesso diferentes se você quiser algo específico. Isso aqui poderia
7       // Ser publico, mas é interessante que possamos limitar um pouco mais quando quisermos
8       internal protected abstract void LoadFrom(string[] data);
9       internal protected abstract string[] SaveTo();
10  }
```

DataBase/DB.cs

```
1   using System.IO;
2   using System.Collections.Generic;
3
4   namespace DataBase;
5
6   using System;
7   using Exceptions;
8
9   public class DB<T>
10      // Restrição genérica, T deve herdar de DataBaseObject e possui um construtor vazio
11      where T : DataBaseObject, new()
12  {
13      // Cada instância de DB tem um caminho base que diz onde serão salvos os arquivos
14      private string basePath;
15
16      private DB(string basePath)
```

```
17          => this.basePath = basePath;
18
19      // Monta o path do arquivos considerando o base path e a classe que queremos salvar (cada classe vai em um arquivo
    diferente)
20      public string DBPath
21      {
22          get
23          {
24              // Pega o nome da classe genérica
25              var fileName = typeof(T).Name;
26              var path = this.basePath + fileName + ".csv";
27              return path;
28          }
29      }
30
31      // Funções privadas apenas para separar melhor as implementações
32      private List<string> openFile()
33      {
34          List<string> lines = new List<string>();
35          StreamReader reader = null;
36          var path = this.DBPath;
37
38          if (!File.Exists(path))
39              File.Create(path).Close();
40
41          try
42          {
43              reader = new StreamReader(path);
44              // Lê linhas de um arquivo até que ele acabe e preenche uma lista com as linhas
45              while (!reader.EndOfStream)
46                  lines.Add(reader.ReadLine());
47          }
48          catch
49          {
50              lines = null; // Falha
51          }
52          finally
53          {
54              reader?.Close(); // Fecha o arquivo, liberando seu uso
```

```
55          }
56
57          return lines;
58      }
59
60      private bool saveFile(List<string> lines)
61      {
62          StreamWriter writer = null;
63          bool success = true;
64          var path = this.DBPath;
65
66          if (!File.Exists(path))
67              File.Create(path).Close();
68
69          try
70          {
71              writer = new StreamWriter(path);
72              // Escrever linhas em um arquivo até que a lista acabe
73              for (int i = 0; i < lines.Count; i++)
74              {
75                  var line = lines[i];
76                  writer.WriteLine(line);
77              }
78          }
79          catch
80          {
81              success = false; // Falha
82          }
83          finally
84          {
85              writer.Close(); // Fecha o arquivo, liberando seu uso
86          }
87          return success;
88      }
89
90      // Retorna uma lista com todos os obejtos
91      public List<T> All
92      {
93          get
```

```
94          {
95              var lines = openFile();
96              if (lines is null)
97                  throw new DataCannotBeOpenedException(this.DBPath); // Estouramos nosso erro personalizado
98
99              var all = new List<T>();
100             try
101             {
102                 for (int i = 0; i < lines.Count; i++)
103                 {
104                     var line = lines[i];
105                     var obj = new T(); // Só podemos fazer isso por causa da restrição genérica where T : new()
106                     var data = line.Split(',', StringSplitOptions.RemoveEmptyEntries); // Splita removendo qualquer dado
    vazio
107                     obj.LoadFrom(data); // Só podemos fazer isso porquê T : DataBaseObject
108                     all.Add(obj);
109                 }
110             }
111             catch
112             {
113                 throw new ConvertObjectError();
114             }
115             return all;
116         }
117     }
118
119     // Salva uma lista com todos os objetos
120     public void Save(List<T> all)
121     {
122         List<string> lines = new List<string>();
123         for (int i = 0; i < all.Count; i++)
124         {
125             var data = all[i].SaveTo();
126             string line = string.Empty; // Linha começa vazia
127             for (int j = 0; j < data.Length; j++)
128                 line += data[j] + ",";
129             lines.Add(line);
130         }
131
```

```
132             if (saveFile(lines))
133                 return;
134 
135             throw new DataCannotBeOpenedException(this.DBPath);
136         }
137 
138         // Variável estática para obter uma instância de DB que salva dados na pasta temporária
139         private static DB<T> temp = null;
140         public static DB<T> Temp
141         {
142             get
143             {
144                 if (temp == null)
145                     temp = new DB<T>(Path.GetTempPath());
146                 return temp;
147             }
148         }
149 
150         // Variável estática para obter uma instância de DB que salva dados na pasta do executável
151         private static DB<T> app = null;
152         public static DB<T> App
153         {
154             get
155             {
156                 if (app == null)
157                     app = new DB<T>("");
158                 return app;
159             }
160         }
161 
162         // Variável estática para obter uma instância de DB que salva dados em uma pasta customizável
163         private static DB<T> custom = null;
164         public static DB<T> Custom
165         {
166             get
167             {
168                 if (custom == null)
169                     throw new CustomNotDefinedException();
170                 return custom;
```

```
171             }
172         }
173
174         public static void SetCustom(string path)
175             => custom = new DB<T>(path);
176     }
```

DataBase/Exceptions/ConvertObjectError.cs

```
1   using System;
2
3   namespace DataBase.Exceptions;
4
5   public class ConvertObjectError : Exception
6   {
7       public override string Message => "Algum elemento do banco está mal formatado e não pode ser convertido.";
8   }
```

DataBase/Exceptions/ CustomNotDefinedException.cs

```
1   using System;
2
3   namespace DataBase.Exceptions;
4
5   public class CustomNotDefinedException : Exception
6   {
7       public override string Message => "O arquivo custom não foi definido. Use DB<T>.SetCustom para definir seu local";
8   }
```

DataBase/Exceptions/ DataCannotBeOpenedException.cs

```
1   using System;
2
3   namespace DataBase.Exceptions;
```

```
4
5   public class DataCannotBeOpenedException : Exception
6   {
7       private string file;
8       public DataCannotBeOpenedException(string file)
9           => this.file = file;
10
11      public override string Message => $"Os dados não puderam ser lidos/escritos no arquivo {file}";
12  }
```

Model/Aluno.cs

```
1   using DataBase;
2
3   namespace Model;
4
5   public class Aluno : DataBaseObject
6   {
7       public string Nome { get; set; }
8       public int Idade { get; set; }
9
10      protected override void LoadFrom(string[] data)
11      {
12          this.Nome = data[0];
13          this.Idade = int.Parse(data[1]);
14      }
15
16      protected override string[] SaveTo()
17          => new string[]
18          {
19              this.Nome,
20              this.Idade.ToString()
21          };
22  }
```

Model/Professor.cs

```
1  using DataBase;
2
3  namespace Model;
4
5  public class Professor : DataBaseObject
6  {
7      public string Nome { get; set; }
8      public string Formacao { get; set; }
9
10     protected override void LoadFrom(string[] data)
11     {
12         this.Nome = data[0];
13         this.Formacao = data[1];
14     }
15
16     protected override string[] SaveTo()
17         => new string[]
18         {
19             this.Nome,
20             this.Formacao.ToString()
21         };
22 }
```

Model/IRepository.cs

```
1  using System.Collections.Generic;
2
3  namespace Model;
4
5  // Representa um repositório de dados de um tipo T qualquer. Você pode implementar várias vezes para conectar com qualquer
   tipo de coisa:
6  // Arquivos, Banco de Dados, Nuvem ou dados estáticos. O interessante é perceber que apenas trocando a implementação, de
   um objeto para
```

```
// outro, trocamos a forma como nossa aplicação se relaciona com dados sem quebrar nada, pois todos os repositórias implementarão as mesmas
// funcionalidades, porém com diferentes comportamentos
public interface IRepository<T>
{
    List<T> All { get; }
    void Add(T obj);
}
```

Model/AlunoFakeRepository.cs

```
using System.Collections.Generic;

namespace Model;

// Repositório Fake com uma lista interna. Pode ser usado para testes sem se comprometer em afetar os dados reais da aplicação.
public class AlunoFakeRepository : IRepository<Aluno>
{
    List<Aluno> alunos = new List<Aluno>();
    public AlunoFakeRepository()
    {
        alunos.Add(new Aluno()
        {
            Nome = "Pamella",
            Idade = 22
        });

        alunos.Add(new Aluno()
        {
            Nome = "Xispita",
            Idade = 18
        });
    }

    public List<Aluno> All => alunos;

    public void Add(Aluno obj)
```

```
        => this.alunos.Add(obj);
}
```

Model/ProfessorFakeRepository.cs

```
using System.Collections.Generic;

namespace Model;

// Repositório Fake com uma lista interna. Pode ser usado para testes sem se comprometer em afetar os dados reais da aplicação.
public class ProfessorFakeRepository : IRepository<Professor>
{
    List<Professor> profs = new List<Professor>();
    public ProfessorFakeRepository()
    {
        profs.Add(new Professor()
        {
            Nome = "Gilmar",
            Formacao = "Doutor"
        });
    }

    public List<Professor> All => profs;

    public void Add(Professor obj)
        => this.profs.Add(obj);
}
```

Model/AlunoFileRepository.cs

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

namespace Model;

```

```
// Esse repositório sim, salva os dados em arquivos e não temporáriamente na memória
public class AlunoFileRepository : IRepository<Aluno>
{
    public List<Aluno> All
        => DB<Aluno>.App.All;

    public void Add(Aluno obj)
    {
        var newAll = All;
        newAll.Add(obj);
        DB<Aluno>.App.Save(newAll);
    }
}
```

Model/ProfessorFileRepository.cs

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

namespace Model;

public class ProfessorFileRepository : IRepository<Professor>
{
    public List<Professor> All
        => DB<Professor>.App.All;

    public void Add(Professor obj)
    {
        var newAll = All;
        newAll.Add(obj);
        DB<Professor>.App.Save(newAll);
    }
}
```

Front/Program.cs

```
1  using static System.Console;
2  using Model;
3
4  IRepository<Aluno> alunoRepo = null;
5  IRepository<Professor> profRepo = null;
6
7  // Se você estiver no modo debug (dotnet run) não acessará arquivos, apenas um repositório fake que traz dados de mentira
8  // para facilitar testes sem ter que apagar ou reiniciar os dados da aplicação. Sem em release (dotnet run -c release)
9  // você está gerando o produto final que acessa o 'banco de dados' nos arquivos. Você ainda pode criar um novo tipo de
   repositório
10 // na Model, conectando com o banco de dados SQL, por exemplo, e só alterar aqui sem ter que alterar mais nada da
   aplicação.
11 #if DEBUG
12
13 alunoRepo = new AlunoFakeRepository();
14 profRepo = new ProfessorFakeRepository();
15
   #else
16
17 alunoRepo = new AlunoFileRepository();
18 profRepo = new ProfessorFileRepository();
19
   #endif
20
21 while (true)
22 {
23     try
24     {
25         Clear();
26         WriteLine("1 - Cadastrar Professor");
27         WriteLine("2 - Cadastrar Aluno");
28         WriteLine("3 - Ver Professores");
29         WriteLine("4 - Ver Alunos");
30         WriteLine("5 - Sair");
31         int opt = int.Parse(ReadLine());
```

```
32
33          switch (opt)
34          {
35              case 1:
36                  break;
37
38              case 2:
39                  Aluno aluno = new Aluno();
40                  aluno.Nome = ReadLine();
41                  aluno.Idade = int.Parse(ReadLine());
42                  alunoRepo.Add(aluno);
43                  break;
44
45              case 3:
46                  var profs = profRepo.All;
47                  for (int i = 0; i < profs.Count; i++)
48                  {
49                      WriteLine(profs[i].Nome);
50                      WriteLine(profs[i].Formacao);
51                      WriteLine();
52                  }
53                  break;
54
55              case 4:
56                  var alunos = alunoRepo.All;
57                  for (int i = 0; i < alunos.Count; i++)
58                  {
59                      WriteLine(alunos[i].Nome);
60                      WriteLine(alunos[i].Idade);
61                      WriteLine();
62                  }
63                  break;
64
65              case 5:
66                  return;
67          }
68      }
69      catch
70      {
```

```
71          // Um Catch voltado ao usuário final e não mais a renomear o erro
72          WriteLine("Erro na aplicação, por favor consulte a TI");
73      }
74
75      WriteLine("Aperte qualquer coisa para continuar...");
76      ReadKey(true);
77  }
```

9 Exercícios Propostos

Complemente a implementação do Exemplo 3 adicionando a adição do professor e tudo que for necessário para turmas e disciplinas.