Invented for life

BOSCH

C#

Docupedia Export

Author:Silio Leonardo (SO/OPM-TS21-BR)
Date:23-Jan-2025 12:25

Table of Contents

Aqui estão reunidos todos os conteúdos de C# do básico ao avançado.

1 Cursos

2 1 - C# Básico

2.1 Aulas

2.2 Duração Estimada

- 24 horas de conteúdo.
- 16 horas de desafios.
- 4 horas de revisão (opcional).
- 4 horas de avaliação (opcional).
- Total: 40 a 48 horas.

2.3 Overview

Neste curso de C# Básico você aprenderá sobre operações básicas, variáveis, estruturas de controle de fluxo e o fundamental sobre orientação a objetos para iniciar suas competências em desenvolvimento .NET.

2.4 Competências

As Competências a serem desenvolvidas ao longo do curso são:

1. **Conhecer** o .NET como tecnologia.
2. **Aplicar** operações e números binárias.
3. **Aplicar** a programação em C# para construir projetos executáveis.
4. **Aplicar** operações básicas da linguagem.
5. **Aplicar** funções para manipular texto.
6. **Compreender** sistema de tipos e variáveis.
7. **Aplicar** estruturas de controle de fluxo.
8. **Aplicar** o pilar OO da Abstração.
9. **Aplicar** o pilar OO do Encapsulamento.
10. **Compreender** a organização da memória.
11. **Compreender** o gerenciamento de dados e ponteiros.
12. **Compreender** os tipos genéricos.
13. **Usar** Listas.
14. **Aplicar** o pilar OO da Herança.
15. **Aplicar** o pilar OO do Polimorfismo.
16. **Aplicar** tipos anuláveis e propagação nula.
17. **Aplicar** System.IO no acesso de arquivos.

2.5 Aula 1 - Introdução ao .NET

- Console, Input e Output
- Variáveis
- Conversões
- Operadores

2.5.1 Introdução

Este curso é introdutório ao C# bem como vários aspectos da linguagem e da computação em si, perfeito para quem deseja desenvolver-se mais profundamente na área. Contudo, ele não é um curso introdutório de programação. Recomenda-se fortemente que tenha-se lógica antes de adentrar o curso, visto que não se irá ensinar com cuidado coisas como For, If, conceitos básicos de variávies, etc. Apenas o necessário de como usar tais coisas no C#.

2.5.2 Linguagens Compiladas e Interpretadas

Para que um programa seja executado em um computador é necessário que exista alguma forma de transformar o texto do código fonte em algo que o computador compreenda, o então chamado código de máquina.
Para isso, existem o que chamamos de **Compilador**. Um Compilador é um programa que recebe arquivos de código fonte de uma linguagem de programação e faz a sua conversão para código de máquina.
É importante perceber que isso significa que o programador entra com um código e recebe como saída um executável (.exe) que é capaz de ser executado em um computador. Mas note que:

- Cada com sistema operacional pode ter dependências (bilbiotecas base) e ainda ser 32 ou 64 bits (veremos isso mais a frente).
- Cada processador tem um código de máquina único.

Assim você precisa compilar para cada arquitetura computacional. Quando trabalhar com linguagens compiladas você precisa ter isso em mente. Chamamos o alvo da compilação simplismente de target.
Exemplos de linguagens compiladas: C, C++, Fortran, Cobol.
Por outro lado existe também outra técnica de tradução que é o **Interpretador**. A ideia é simples porém brilhante: Faça um programa que leia um código fonte e execute imediatamente. Por exemplo: Ao ler uma linha de código 'print("ola")' o programa imediatamente executa o 'printf' da linguagem C, por exemplo, tendo como parâmetro 'ola'. Se ele ler uma linha 'x = 4', ele imediatamente salva em algum lugar o valor 4 apontando que o mesmo se encontra numa variável chamada 'x'.
A ideia é inteligente pois podemos fazer um Interpretador para cada arquitetura e um mesmo programa pode rodar em diferentes arquiteturas, pois será interpretado por diferentes programas para diferentes arquiteturas mas que tem um mesmo funcionamento.
Por isso, linguagens como Python rodam em todo lugar: Basta ter um interpretador Python (que é escrito em C) compilador para a arquitetura destino.
O mesmo acontece com JavaScript, Css e Html (as duas últimas não são linguagens de programação, apenas linguagens de estilização e estruturação, respectivamente). Um navegador não é nada mais, nada menos, que um interpretador que recebe código online e apresenta o seu processamento.
A desvantagem das linguagens interpretadas é que costumam ser mais lentas, afinal, é preciso ler cada linha de código, interpretar o que ela faz e então executar o código de máquina equivalente escrito em C. Além disso, para que uma aplicação funcione no seu cliente, é necessário que você exponha o código original do seu software para ele.
Outra vantagem é que na linguagem interpretada você não precisa passar por um processo para que ela se torne uma linguagem executável (.exe) toda vez que quiser testá-la, tornando-a, em geral, mais ágil.

Característica	Compilada	Interpretada
Velocidade de Compilação	Lenta	Inexistente
Velocidade de Execução	Rápida	Mais Lenta
Compatibilidade sem Recompilação	Não	Sim

2.5.3 Ahead-Of-Time Compilation, Representações Intermediárias e Linguagens Híbridas

Uma técnica poderosa utilizada pelas linguagens modernas é a **Compilação Antecipada**, Ahead-Of-Time, ou simplismente AOT. Está técnica consiste em traduzir um código mais alto-nível (mais inteligível para humanos) em um código mais baixo-nível (mais próximo ao código de máquina).
Ou seja, você pode transformar JavaScript em C++, por exemplo. Mas é muito mais do que isso. A utilização mais comun é criar uma representação intermediária. Essa representação intermediária (IR) do código pode se utilizada para diminuir o trabalho da compilação.
Por exemplo, você pode ter várias linguagens que se transformam em uma mesma IR, assim todas elas seriam compatíveis, ao passo que seria necessário apenas transformar o IR em código de máquina uma única vez.
Entenda: Se nós temos 3 linguagens e 5 arquiteturas desejadas, seguindo os conceitos tradicionais precisariamos desenvolver 15 compiladores, 1 para cada linguagem sendo convertida para cada arquitetura. Usando AOT, precisamos de 3 compiladores, de cada linguagem para a IR, e 5 compiladores, da IR para cada arquitetura. Assim 8 no total e, de quebra, ganhamos alta compatibilidade entre as linguagens.
E o mais poderoso: Podemos construir um compilador para transformar da linguagem fonte na IR e criarmos um interpretador para a IR executar. Assim temos linguagens de compilação **Híbridas**.

2.5.4 Java e Just-In-Time Compilation

A grande vantagem das linguagens Híbridas são linguagens como o **Java**. Muitas linguagens como Java, Kotlin, Scala, são convertidas na mesma IR que é levada ao cliente e interpretada em diferentes máquinas. Basta ter um software de tempo de execução chamada JVM ou Java Virtual Machine, instalada na arquitetura alvo.
Isso também permite que o mesmo código uma vez compilado (IR) execute em praticamente qualquer arquitetura, basta existir uma implementação da JVM para ela.
Além disso, o Java utliza uma ténica chamada de **Just-In-Time** ou JIT. O JIT é uma forma de interpretação aperfeiçoada: Ela compila a representação intermediária no momento em que ela deveria ser executada, possibilitando a execução diretamente na CPU do computador, realizando ainda optimizações que não poderiam ser feitas em outro momento, possibilitando que as desvantagens de linguagens interpretadas sejam liquidadas, desempenhando muito bem em diferentes arquiteturas. Além disso, apenas o código usado é convertido, partes do código não executadas não são convertidas e as partes convertidas ficam salvas, fazendo com que em futuras execuções o desempenho da aplicação melhore ainda mais.
Ou seja, JIT é poderoso, contudo, complexo de se implementar, mas dá um alto nível de flexibilidade e compatibilidade.

2.5.5 Breve História do C#

A Microsoft desejava otimizar o Java ainda mais para que tivesse um desempenho melhor no Windows, tentando tornar seu sistema operacional (e carro chefe da empresa) ainda mais atrativo. Assim foi criado o J++, basicamente, o mesmo Java, porém que rodaria em uma nova implementação da JVM que só funcionária no Windows.
A Sun, criadora do Java, não gostou disso e como havia algumas quebras de patentes foi instaurado um processo contra a Microsoft que perdeu nos tribunais. Assim o J++ morreu e a empresa de Bill Gates resolveu criar sua própria linguagem concorrente do Java.
Chamada inicialmente de Cool, essa linguagem também usaria uma máquina virtual e JIT para a compilação. Naquele tempo Steven Ballmer, CEO da Microsoft, tinha um cabeça, digamos, um pouco fechada. A nova linguagem vinha para ter um foco em Windows e até por isso, e por no começo estar incompleta, C#, como então foi batizada a linguagem, virou motivo de piada, uma cópia de Java.
O que não se esperava é que a linguagem vingaria tanto. Mudando os preceitos para rodar em mais lugares e melhor, trazendo novas atualizações e características únicas (algumas coisas implementadas no C# 2007 só vieram aparecer no Java depois de 2015, ou seja, Java começou a se inspirar na linguagem da mesma forma que o contrário ocorreu), C# se tornou extremamente competitiva e poderosa. Embora a velha guarda custe a perceber.
Com a compra da Unity pela Microsoft, do Xamarin entre outras novas soluções, hoje é possível usar C# para programar para: Desktop, Android, IOS, Web Frontend, Web Backend, Nuvem, Microcontroladores, aplicações IOT, Playstation, Xbox, Switch, entre outros.

2.5.6 Tipagem

Um tópico interessante da Ciência da Computação que nos ajuda a compreender melhor as características da linguagem C#, bem como outras, é a **Tipagem**.
A tipagem é como um plano Cartesiano que classifica as linguagens em 2 eixos:

- Tipagem Dinâmica vs Tipagem Estática
- Tipagem Forte vs Tipagem Fraca

Uma Tipagem Estática é uma linguagem onde os tipos de todas as variáveis são definidos ainda na fase de compilação. Ou seja, se a variável 'x' contém um número, isso será identificado ainda na compilação e 'x' só poderá receber um número. Isso acontece em linguagens como C ou Java.
Uma Tipagem Dinâmica é uma linguagem onde as variáveis podem mudar de tipos o tempo todo. Como Python e JavaScript, por exemplo. Então escrever 'x = 2' e em seguida 'x = "oi"' é aceitável numa linguagem Dinâmica, mas não em uma Estática.
Já um Tipagem Forte é onde os dados tem tipo bem definido. Por exemplo, vamos supor que criamos um variável 'x' valendo 4. Ao tentar obter o resulado da seguinte expressão 'x[0]' teríamos um erro pois 4 não é um vetor então não podemos acessar a posição zero dele.
Para uma Tipagem Fraca a expressão retornaria algum valor, mesmo que o mesmo seja tratado como 'indefinido'. O fato de não aparecer um erro faz com que não percebamos que algo deu errado. Isso pode trazer flexibilidade mas aumenta a quantidade de bugs.
Abaixo uma pequena tabela com exemplos:

Linguagem	Força	Dinamicidade
C K&R (bem antigo)	Fraca	Estática
C Ansi (mais moderno)	Forte	Estática
Java	Forte	Estática
C++	Forte	Estática
JavaScript	Fraca	Dinâmica
PHP	Fraca	Dinâmica
Python	Forte	Dinâmica
Ruby	Forte	Dinâmica

Nota: Aqui você percebe que Java e JavaScript tem muito pouco em comum.

Aqui vale uma ressalva: Permitir que várias conversões automáticas aconteçam como no JS torna uma linguagem mais fraca. Por exemplo ao computar '{} + {}', um dicionário vazio mais outro dicionário vazio se obtém '[object Object][object Object]', pois o JS tenta converter para um texto apresentável. O exemplo anterior 'x[0]' ou até mesmo '4[0]' apresenta valor indefinido. Isso é muito diferente do que fazer isso em uma linguagem como o C antigamente, onde 'x[0]' é literalmente acessar a memória do computador no endereço 4, pois o C entende (ou entendia) tudo como números e mais números e não da significado para seus valores. Assim é fácil perceber que existem diferentes níves de linguagem fraca.
Neste contexto, onde o C# se encontra? Bem, ao procurar online você verá que C# é uma linguagem Forte e Estática, é assim que a usamos. Porém, é importante salientar que C# possui recursos de tipagem dinâmica incluso no seu baú de recursos. E não só isso, C# possui várias características que podem tornar a linguagem um pouco menos forte, embora jamais deixe de ser uma linguagem inerentemente forte.

2.5.7 Paradigmas de Programação

Um paradigma de programação é a maneira como uma linguagem de programação se estrutura para orientar o pensamento do programador. Existem diversas abordagens para se programar; estamos mais

acostumados com a programação Imperativa e Estruturada, ou seja, dizemos como o programa deve fazer o processamento de dados e estruturamos isso em várias funções e variáveis, executando estruturalmente linha a linha.
Contudo, existem outras propostas: Ainda como linguagem imperativa temos a Orientação a Objetos, que muda a forma como modularizamos o estado (variáveis) e comportamento (métodos) do nosso programa, escondendo dentro de escopos e afastando da maneira estruturada onde deixamos tudo no mesmo programa.
Por outro lado, temos os paradigmas declarativos, onde se fala o que se deseja fazer, não se importando no como. Dentro deste mundo existem paradigmas como o Funcional, onde a definição de funções, uso de funções como dados (o que sim, parece confuso e de fato é), além de vários recursos prontos tornam a programação muito diferente da que estamos acostumados, mas pode aumentar muito a produtividade.
O C# lhe permite escrever código estruturado, embora seja uma linguagem inerentemente Orientada a Objetos. E dentro da programação imperativa, temos ainda a programação Orientada a Eventos que não será abordado neste curso, mas C# da suporte a mesma. C# também tem suporte a programação funcional entre outros aspectos de programação declarativa. Ou seja, C# é uma linguagem vasta considerada multi-paradigma.

2.5.8 Ecossistema .NET

.NET é a plataforma de desenvolvimento criado pela Microsoft. Ela, em si, é muito maior que somente o C#. Por isso, os profissionais se dizem programaodres .NET, e não apenas programadores C#. Vamos explorar rapidamente a fim de entender o que é o .NET. Para isso vamos aprender os seus componentes:

- .NET possui 3 linguagens principais: C#, Visual Basic e F# (uma interessante linguagem funcional). Todas são convertidas para uma linguagem intermediária (ou seja, um linguagem que é uma representação intermediária/IR) chamada CIL ou Common Intermediate Language. Ou seja, as linguagens são intercambiáveis e totalmente integráveis.
- CLR, Common Language Runtime é a máquina virtual que transforma o CIL em código nativo capaz de rodar em muitas arquiteturas diferentes.
- .NET Framework é uma espécie de biblioteca padrão com centenas de recursos para realizar qualquer tarefa: Trabalhar com arquivos, Criptografia, Comunicação em Redes, Desenvolvimento Web e afins. Para que programas rodem você precisa ter instalado na sua máquina as bibliotecas do .NET Framework na versão desejada.
- .NET Core: Aqui entra um discussão interessante que confunde muitos. Como anteriormente mencionado, o C# era voltado apenas para Windows nos anos 2000 até 2016. Embora C# pudesse rodar em qualquer lugar, o .NET Framework muitas vezes tinha implementações apenas para Windows. Com isso, no final de 2014 anunciou-se o .NET Core que vinha como uma nova implementação melhorada, tornando até mesmo várias funções antigas mais rápidas. Assim, por muito tempo tivemos dois frameworks principais: .NET Framework ou simplesmente .NET avançado até as versões 4.X, e o .NET Core, avançado até a versão 3.1. Neste momento, o .NET Framework 'morreu'. A partir do .NET Framework 5.0, temos na verdade o .NET Core que a partir desta versão, chamamos apenas de .NET ou .NET Framework. Então se você está usando o .NET na versão 4.6, por exemplo, você está usando o antigo .NET. Se você está usando o .NET Core, ou o .NET 5 em diante, você está usando o novo framework que surgiu em 2016.
- .NET Standard é uma especificação que aponta o mínimo que um Framework precisa ter para se tornar mais compatível. Explicando: Quando desenvolvedores queriam criar bibliotecas compatíveis tanto com .NET Core quanto .NET Framework, eles utilizavam apenas o que era assegurado estar no .NET Standard, que seria basicamente as implementações compartilhadas por ambos os frameworks. Utilizar algo que não estava no .NET Standar poderia fazer com que a biblioteca ficasse incompatível, ou com .NET Framework, ou com o .NET Core.
- .NET SDK ou SDK (Software Development Kit) é o Kit de desenvolvimento do .NET. É possível instalar apenas os recursos de execução, sendo um usuário, e executar programas .NET sem ter o Kit de desenvolvimento. Para os desenvolvedores, basta instalar o SDK e ter acesso as ferramentas para construir novas aplicações.

- .NET CLI (Command Line Interface) é um programa que utilizamos para criar projetos, testar, executar entre várias operações no momento de gerenciar projetos em .NET. Com o .NET SDK instalado, basta utilizar o comando 'dotnet' no terminal para utilizar o .NET.

2.5.9 dotnet CLI

Como já comentado, o dotnet CLI é um programa que te ajuda a desenvolver em C#. A partir dele você irá criar e executar aplicações. Aqui uma lista de comandos que você pode executar em qualquer terminal PowerShell ou semelhante:

- dotnet --version: Mostra a versão instalada do dotnet
- dotnet --help: Mostra lista de comandos disponíveis
- dotnet new --list: Mostra a lista de tipos de projetos que você pode desenvolver
- dotnet new console: Cria uma nova aplicação console
- dotnet new gitignore: Cria um arquivo gitignore especial para C# a ser utilizado junto com o github
- dotnet run: Roda o programa
- dotnet build: Compila o projeto e mostra os possíveis erros de compilação, sem executar a aplicação
- dotnet clean: Limpa todos os arquivos de compilação. Pode ser útil para se livrar de alguns erros desconhecidos
- dotnet add package : Baixa e instala um pacote da internet (nuget).

2.5.10 Função Main e Arquivo Top-Level

Assim como programas C/C++, todo programa C# se inicia na função Main. É comum ver exemplos do famoso Hello World apresentados da seguinte forma:

```
1  using static System.Console;
2
3  namespace MeuProjeto
4  {
5      public class Program
6      {
7          public static void Main(string[] args)
8          {
9              WriteLine("Olá mundo!");
10         }
11     }
12 }
```

Mas não se preocupe com a aparente complexidade de um programa simples, vamos entender cada um desses 3 elementos que você vê (namespace, class e a função Main) ao longo do curso. Por enquanto vamos falar da função Main, já que ela não é comum para programadores de funções como Python e JavaScript.

Uma função Main é o ponto inicial de todo programa C# e só devemos ter uma única função Main. Não se preocupe com a declaração da função e seus detalhes pois vamos abordar cada aspecto separadamente, bem como a forma de se declarar uma função no C#.

De inicio, é bom salientar que void significa que a função não deve retornar nada. Se você aprendeu funções através de Python, JavaScript, entre outras linguagens não está acostumado a declarar o tipo de retorno da função, mas isso acontece em C# e temos a opção void, isentando a função de retornar qualquer coisa.

Outro aspecto é o vetor de string chamado comumente de args. Esses são os argumentos para o programa. Todo programa possui argumentos que mudam seu comportamento. O melhor exemplo é o dotnet CLI, onde ao executar o programa 'dotnet', mandamos a string 'new' como parâmetro. A cada espaço de divisão, um novo parâmetro é gerado. Note que na grande maioria das vezes esse parâmetro é ignorado.

Apesar disso tudo, o C# evoluiu para ser mais simples e menos confuso. Então na versão 6.0 em diante temos os arquivos Top-Level. Em todo programa você pode ter um único arquivo Top-Level. Ele não possui nenhuma das estruturas no exemplo anterior. Todo seu código será jogado dentro de uma função Main artificial gerada na compilação. Assim o código a seguir, torna-se válido, útil e simples:

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

Note que 2 arquivos Top-Level geram um erro pois acabam por gerar 2 funções Main o que é inválido. Algumas coisas que podem passar pela sua cabeça são as seguintes:

- **Então eu não consigo mais acessar o vetor 'args'?** Na verdade consegue, bastar usar a palavra reservada args.
- **Não é possível colocar uma função em um arquivo Top-Level? Já que isso seria colocar uma função dentro da outra.** Felizmente, C# suporta funções declaradas dentro de outras funções.
- **Posso usar outras estruturas como no primeiro exemplo? Existe alguma limitação?** Existem algumas, mas a maioria das estruturas que não são códigos executáveis são automaticamente jogadas para fora da função Main.

Outro detalhe importante é que você precisa de código executável no Top-Level para que ele reconheça a existência de uma função Main. Você verá que existem estruturas que se colocadas sozinhas no arquivo Top-Level acabam por indicar para o compilador que na verdade aquele não é um arquivo Top-Level e que não existe uma função Main e isso acarreta em um erro de compilação.

2.5.11 Console, Input e Output

Como você pode observar, para apresentar informações na tela basta usar o print do C#, chamado de WriteLine. Ele é muito útil para várias coisas, como checar erros ou fazer aplicações para Console, que é o nome dado para aplicações de tela preta/Terminal. Para que você use-a é necessário importá-la da seguinte maneira:

using static System.Console;

Depois de incluir esse código no topo do seu arquivo você tem acesso a muitas funcionalidades:

- WriteLine: Escreve algo na tela e pula uma linha logo a seguir.
- Write: Escreve algo na tela.
- ReadLine: Lê a próxima linha digitada pelo usuário e retorna o valor.
- ReadKey: Lê apenas um caracter digitado pelo usuário e retorna suas informações. Você pode mandar o parâmetro 'true' para evitar que o que for digitado apareça na tela.
- Beep: Emite um som em determinada frequência.
- Clear: Limpa o Console.
- BackgroundColor: Variável onde você pode definir a cor do fundo do que será escrito.
- ForegroundColor: Variável onde você pode definir a cor da fonte do que será escrito.
- Title: Variável onde você pode definir o titulo do Console.
- CursorLeft: Variável onde você pode definir a posição horizontal do cursor.
- CursorTop: Variável onde você pode definir a posição vertical do cursor.
- CursorVisible: Variável onde você pode definir se o cursor é visível.
- WindowWidth: Variável onde você pode definir a largura da janela do Console.
- WindowHeight: Variável onde você pode definir a altura da janela do Console.

Entre outras funções. Segue um pequeno exemplo:

```
1  using static System.Console;
2
3  WriteLine("Digite algo...");
4  string texto = ReadLine();
5
6  BackgroundColor = ConsoleColor.White;
7  ForegroundColor = ConsoleColor.Blue;
8  CursorVisible = false;
9
10 Clear();
11
12 Write("O usuário digitou: ");
13
```

```
ForegroundColor = ConsoleColor.Red;
WriteLine(texto);
```

2.5.12 Variáveis

Como você pode perceber no último exemplo, para declarar uma variável em C# você só precisa digitar seu tipo seguido do nome e atribuição. Como discutido anteriormente, C# tem tipagem estática em 99% do tempo e você precisa apontar o tipo usado. Existem uma grande variedade de tipos:

```
byte b1 = 0; // Inteiro com 2^8 valores de 0 a 255 (naturalmente sem sinal).
sbyte b2 = 0; // Inteiro com sinal (signed) com 2^8 valores de -128 a 127.

short s1 = 0; // Inteiro com 2^16 valores de -32'768 a 32'767.
ushort s2 = 0; // Inteiro sem sinal (unsigned) com 2^16 valores de 0 a 65'535.

int i1 = 0; // Inteiro com 2^32 valores de -2'147'483'648 a 2'147'483'647.
uint i2 = 0; // Inteiro sem sinal (unsigned) com 2^32 valores de 0 a 4'294'967'295.

long l1 = 0; // Inteiro com 2^64 valores de -9'223'372'036'854'775'808 a 9'223'372'036'854'775'807.
ulong l2 = 0; // Inteiro sem sinal (unsigned) com 2^64 valores de 0 a 18'446'744'073'709'551'615.

nint n1 = 0; // int se o sistema for de 32 bits, longo se o sistema for de 64 bits, ou seja, um inteiro nativo.
nuint n2 = 0; // O mesmo que o nint, porém, podendo ser uint ou ulong.

char c = 'a'; // Caracter.
string s = "texto"; // Cadeia de caracteres/texto.

bool b = true; // Booleano, ou seja, verdadeiro ou falso (true/false).

float f = 0f; // Tipo de ponto flutuante, isso é, armazena números com virgula até uma determinada potência com uma quantidade limitada de casas decimais.
double d = 0.0; // Semelhante ao float, porém com o dobro de armazenamento, podendo ter mais casas decimais e representar maiores números.
decimal m = 0m; // Tipo desenvolvido para guardar dinheiro. Dobro do armazenamento do double. Menor range de valores mas muito mais casas decimais.

var x = 0; // Descobre automaticamente o tipo da variável de forma Implícita, sem precisar escrever.
// x = "oi"; // Mas não permite a mudança do tipo da variável.

dynamic y = 0; // Desabilita verificações e guarda qualquer tipo de dado.
y = "oi"; // Pode mudar de tipo ao longo do programa, porém, pouco utilizado.

int i3 = int.MaxValue; // Obtém maior valor possível para int.
int i4 = 0b_0000_0101; // Valor binários.

//int i4 = 5; // Erro, não é possível declarar duas variáveis com mesmo nome.
i4 = 6; // Ok.

```

```
37  //byte b3 = 300; // Erro, valor muito grande para byte.
38
39  int[] vetor = new int[10]; // Criando um vetor com 10 posições. Todas as
    posições devem ser um int. Se iniciam todas as posições como 0.
40  var vetor2 = new string[] { "Olá", "Mundo", "!" }; // Inicialização com
    valores em uma variável Implícita usando var.
```

Usar variáveis ocupa menos espaço e pode ser útil e muitas aplicações, então vale a pena compreender um pouco dos tipos e seus usos.

2.5.13 Conversões

Converter valores é uma funcionalidade importante, básica e corriqueira durante a programação. Assim sendo é importante aprender a converter tipos em uma linguagem com a tipagem do C#. Duas são muito comuns:

- String Parse

```
1  string numeroEmTexto = ReadLine();
2  int numero = int.Parse(numeroEmTexto);
```

- Type Cast

```
1  int valor = 100;
2  // byte outroValor = valor; // Erro! valor pode ser um número muito grande
   para uma variável byte, ou até negativo. Por isso o C# bloqueia essa
   operação.
3  byte outroValor = (byte)valor; // Ok.
```

2.5.14 Operadores

Também temos vários operadores em C#, bem como em outras linguagens. A seguir uma lista de várias operações váldas no C#:

```
1   int a = 3;
2   int b = 2;
3
4   int r1 = a + b; // 5
5   int r2 = a - b; // 1
6   int r3 = a * b; // 6
7   int r4 = a / b; // 1 (divisão de inteiros)
8   int r5 = a % b; // 1 (resto da divisão)
9   int r6 = a << b; // 12 shift (operação binária) equivalente a 'a * 2^b'.
10  bool r7 = a > b; // true
11  bool r8 = a <= b; // false
12  bool r9 = a == b; // false
13  bool r10 = r7 && r8; // Operador And
14  bool r11 = r7 || r8; // Operador Or
15  bool r12 = !r7; // Operador Not
16
17  int r13 = r1++; // 5. Retorna r1 (5, no caso) e depois soma 1 a variável
    r1 que agora vale 6.
18  int r14 = ++r1; // 7. Soma 1 na variável r1 que agora vale 7 e retorna
    este valor após isso.
```

2.6 Aula 2 - Escovando bits com C# e Strings

- Operações com Strings
- Exercícios Propostos

2.6.1 Operadores: Uma visão em binário

A compreensão de números binários é importante para compreender algumas operações, bem como permitir que certas coisas sejam feitas mais eficientemente. Desta forma, vamos ver algumas das operações mostradas no final da Aula 1 olhando para números binários.
Primariamente, precisamos entender bem o que são números binários.
Um número binário é uma sequência de bits, que são nada mais, nada menos, que 0's e 1's. Vamos usar bastante a noção de byte daqui para frente, que é o conjunto de 8 bits, indo de 0 (00000000) a 255 (11111111) bem como o tipo definido em C#.
Assim como em decimal, que tem os digitos de 0 a 9, ao olhar o próximo número apenas somamos 1 ao digito mais a direita:
34 -> 35
O sucessor de 34 é 35, pois 4 + 1 é 5. Note que quando passamos do último digito (9) voltamos para o primeiro (0) e somamos 1 a casa da esquerda:
49 -> 50
O sucessor de 49 é 50, pois 9 + 1 = 10, assim o 9 torna-se 0 e somamos 1 ao 4 que torna-se 5. Da mesma forma:
9199 -> 9200
Em binário, os mesmos mecânismos acontecem. Porém, só temos 2 digitos, assim:
000 -> 001 001 -> 010 010 -> 011 100 -> 101 101 -> 110 110 -> 111
Pela tabela podemos ver como isso ocorre:

Binário	Decimal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Note que existem formas mais fáceis de realizar conversões binário-decimal após compreender seu funcionamento. Então vejamos elas: Vamos tentar converter 10100101 de binário para decimal. Basta ignorar os 0's e para cada 1 você soma uma potência de 2. Essa potência deve ser correspondente a posição de cada 1 no binário. Assim a contribuição dos 0's é zero e a dos 1's depende da sua posição no binário. Observe:

digito	7°	6°	5°	4°	3°	2°	1°	0°	resultado
binário	1	0	1	0	0	1	0	1	
potência	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
resultado	128	64	32	16	8	4	2	1	

digito	7°	6º	5°	4º	3°	2º	1°	0º	resultado
contribuição	128	0	32	0	0	4	0	1	165

Da mesma forma, podemos converter de um decimal para um binário apenas buscando os bits onde a conversão de correta. Interessantemente, só existe uma combinação de 0's e 1's para cada decimal. Um truque para realizar essa operação é ler da esquerda para direita, assim se a soma do bit não extrapolar o decimal, consideramos o mesmo como 1. Observe a conversão de 202 para binário:

passo	binário	conversão	é maior que 202	ação
1	1_____	128	não	próximo bit
2	11_____	192	não	próximo bit
3	111_____	224	sim	reverte
4	110_____	192	não	próximo bit
5	1101____	208	sim	reverte
6	1100____	192	não	próximo bit
7	11001___	200	não	próximo bit
8	110011__	204	sim	reverte
9	110010__	200	não	próximo bit
10 R:	1100101_ 11001010	202 202	é igual fim	termina, próximos bits iguala a 0

2.6.2 Exercícios Propostos

a. Converta os seguintes valores:
- a) 17 para binário
- b) 102 para binário
- c) 254 para binário
- d) 1111_1100 para decimal
- e) 1000_0000 para decimal
- e) 1100_0001 para decimal
- g) 2641 para binário
- h) 1100_1010_1100 para decimal

Podemos observar agora algumas das operações do C# com um pouco de distinção e pensar nos números binários a partir disto. Vamos começar com o Shift ou deslocamento binário:
00000001 << 1 = 00000010
Basicamente, temos o deslocamento uma casa a esquerda. Outro exemplo:
01010101 << 1 = 10101010
Isso ocorre a todos os bits. Note que um deslocamento maior que 1 pode ocorrer:
00000001 << 7 = 10000000
Outra noção importante é que se um número sai para fora dos bits (depende se for um byte, short, int ou long) ele simplesmente desaparece:
11110000 << 2 = 11000000

O Shift também pode ser aplicado a direita:
11110000 >> 2 = 00111100
Ele também segue as mesmas regras que o outro shift:
11111111 >> 3 = 00011111
Você ainda pode escrever em código C#:
byte b = 4;
b <<= 2;
Da mesma forma como se usa +=, -=, *=, /= em C# e outras linguagens.
Você também pode usar operações lógicas fazendo operações bit-a-bit:
11100000 | 00000111 = 11100111 (Ou, aplicado bit-a-bit)
11111000 & 00011111 = 00011000 (E, aplicado bit-a-bit)
11110000 ^ 11001100 = 00111100 (Ou exclusivo, aplicado bit-a-bit)
~00001111 = 11110000 (Não, aplicado bit-a-bit)
É importante salientar que os números com sinal são representados na forma complemento de 2. Isso significa, entre outras coisas, que o primeiro bit representa se o número é negativo ou não. Logo, caso alguma operação que você realizar jogue um 1 no primeiro bit o número ficará negativo.
Outro ponto rápido: existe uma diferença entre && e || de & e |. A repetição do símbolo denota circuito-curto, o que significa que a depender do primeiro valor, o segundo não é considerado. Assim true || false não considera a segunda parte da expressão enquanto o circuito-completo true | false considera a expressão como um todo. Com isso &&/|| não pode ser usado em tipos binários, mas pode evitar cálculos desnecessários e é bom ser usado em condicionais.

2.6.3 Estruturas Básicas

Como você deve ter aprendido em outras linguagens, para programar é necessário de algumas estruturas para controlar o fluxo de execução. Agora, vamos ver como produzi-las em C#:

- if

```
1   using static System.Console;
2
3   int idade = int.Parse(ReadLine());
4   if (idade > 17)
5   {
6       WriteLine("É maior de idade.");
7   }
8   else if (idade > 15)
9   {
10      WriteLine("Não é maior de idade. Mas em alguns países já pode dirigir")
    ;
11  }
12  else
13  {
14      WriteLine("Menor de idade.");
15  }
```

Então, caso a idade digitada seja 18 para cima a condição no primeiro escopo é executada. Note que a segunda condição também seria verdadeira, porém, apenas uma cláusula é executada por vez. Caso nem o 'if', nem o 'else if' seja executado, executamos o 'else'.

- switch

```
1   using static System.Console;
2
3   int idade = int.Parse(ReadLine());
4   switch (idade)
5   {
6       case 18:
7           WriteLine("É maior de idade.");
8       break;
9
10      case 19:
11          WriteLine("Está ficando velho!");
```

```
12      goto case 18;
13
14      case 16:
15      case 17:
16          WriteLine("Não é maior de idade. Mas em alguns países já pode
    dirigir.");
17      break;
18
19      default:
20          WriteLine("Menor de idade.");
21      break;
22  }
```

O switch usa uma forma diferente de execução, tornando-o mais rápido a depender da situação. Se existem poucos intervalos e muitas opções distintas, o switch é uma boa opção. Você precisa fechar todos os casos com break, return ou goto. O mais comum é se utilizar break, mas como pode observar, você pode usar goto para que 2 casos diferentes executem. Você também pode usar 2 cases seguidos para indicar o mesmo comportamento para diferentes valores da variável.

- while

```
1   using static System.Console;
2
3   WriteLine("Fatorial de qual valor você irá querer?");
4   int n = int.Parse(ReadLine());
5
6   int fatorial = 1;
7
8   while (n > 1)
9   {
10      fatorial *= n;
11      n--;
12  }
```

- do while

```
1   using static System.Console;
2
3   int numeroSecreto = 540;
4   int numero;
5
6   do
7   {
8       WriteLine("Tente adivinhar o número secreto...");
9       numero = int.Parse(ReadLine());
10
11      if (numero > numeroSecreto)
12          WriteLine("O numero secreto é menor");
13      else if (numero < numeroSecreto)
14          WriteLine("O numero secreto é maior");
15
16  } while (numero != numeroSecreto);
```

No While e Do...While temos as mais básicas estruturas de repetição. O bloco é executado apenas enquanto a condição seja verdadeira. O que muda é apenas o momento em que a condição é considerada.

- for

```
1   using static System.Console;
2
3   int[] vetor = new int[50];
4
```

```
WriteLine("Digite 50 valores");
for (int i = 0; i < vetor.Lenth; i++)
{
    vetor[i] = int.Parse(ReadLine());
}
```

Também temos o For que é ótimo para acessar vetores pois o mesmo tem inicialização, condição e incremento no final de cada loop. É bom lembrar que também temos coisas como break, que interrompe um loop e continue, que pula para o próximo loop.

2.6.4 Funções

A partir de agora, vamos parar de fazer tantas operações fora de funções. Funções são uma das estruturas mais básicas e importantes dentro da programação e por isso vamos aprender a utilizá-las em C#. A estrutura básica é:
retorno nome(parâmetros) { implementação }
Em C# você não precisa declarar a função antes de usá-la (ela pode estar depois no código), mas você precisa seguir algumas regras:

- Se a função pede algum retorno, você deve retorná-lo, caso contrário você terá um erro de compilação.
- Você deve garantir que todos os caminhos retornam algo.
- Cada parâmetro é como uma variável que deve ser enviado na chamada, a não ser que seja opicional.
- Se você declarar uma parâmetro com mesmo nome de outra variável no programa, ao usar a variável estará usando a declarada dentro da função.

A seguir exemplos:

```
int value = modulo(-3);

int modulo(int i)
{
    if (i < 0)
        return -i;
    return i;
}
```

```
using static System.Console;

WriteLine("Fatorial de qual valor você irá querer?");
int n = int.Parse(ReadLine());

WriteLine("O resultado é: " + fatorial(n));

int fatorial(int n)
{
    if (n < 2)
        return 1;

    return n * fatorial(n - 1);
}
```

```
using static System.Console;

mostreUmTexto("Hello World!");
mostreUmTexto(); // mostrará 'Olá mundo'

void mostreUmTexto(string texto = "Olá Mundo!") // não retorna nada
{
    WriteLine("Texto:");
```

```
    WriteLine(texto);
    //return; // pode ter um return aqui, mas o mesmo não é obrigatório
}
```

2.6.5 Operações com Strings

Você pode usar algumas operações com strings que são mostradas abaixo. Para isso considere a existência das variáveis text do tipo string valendo "Xispita" e a num do tipo int valendo 76:

- text.Contains("xis"), retorna verdadeiro se "xis" está em text.
- text.EndsWith("pita"), retorna verdadeiro se "pita" está no final de text.
- text.IndexOf("i"), retorna a posição (base 0) da primeira ocorrência da string "i", no caso índice 1.
- text.Insert("pi", 5), Retorna um novo texto com "pi" inserido na variável text, no caso "Xispipita".
- string.IsNullOrEmpty(text), retorna verdadeiro se o texto é null (veremos no futuro) ou "" (string vazia).
- string.IsNullOrWhiteSpace(text), retorna verdadeiro se o texto é null ou espaços vazios.
- text.Replace("X", "Ch"), retorna nova string substituindo um valor por outro.
- text.Trim(), retorna nova string com todos os espaços no inicio e no fim removidos.
- text.Split("i"), retorna vetor de strings que são o texto original dividido em todas as ocorrências de "i", no caso ["X", "sp", "ta"].
- text.Substring(0, 2), retorna uma nova string sendo está os caracteres a partir do índice 0 contando 2 caracteres.
- $"O valor é {num}", interpola uma string substituindo {num} pela variável num.
- num.ToString(), converte num para string.

2.6.6 Exercícios Propostos

a. Escreva uma função C# que converta um número decimal para binário (retornando uma string).
b. Faça um programa que dado dois números dados pelo usuário a e b, desenhe na tela, em binário, o resultado de a << b;

2.7 Aula 3 - Desafio 1

Agora que já aprendemos o básico do C# e já podemos escrever programas simples vamos ao primeiro desafio: Um compressor de dados com perdas.
A ideia é simples: Em arquivos como imagens e vídeos a perda de informação durante a compressão é aceitável e até desejado. Isso advém do fato de que é pouco perceptível aos nossos olhos.
A ideia é eliminar parte dos dados e reduzir o espaço gasto por eles. Ao ver uma cor RGB (255, 255, 255) que é branco podemos perceber que em binário temos:
11111111, 11111111, 11111111
Ao jogarmos fora os 4 bits menos significativos (da direita), poderíamos guardar apenas o 4 dígitos que mais impactam no cálculo do binário:
1111, 1111, 1111
Assim, criamos um compressor de 50%. Ao tentar voltar para 8 bits para poder rever a cor podemos preencher os bits que perdemos com valores como 0:
11110000, 11110000, 11110000
Que é o RGB (240, 240, 240). Procure no goolge por 'rgb(240, 240, 240)' e veja a pequena diferença para o branco 'rgb(255, 255, 255)'. E perceba que este é o pior caso possível, onde mais se tem perda de informação. Se o RGB já fosse (240, 240, 240), por exemplo, não teríamos perda alguma.
Seu trabalho é fazer uma função que receba um vetor de byte que será compactada desta forma. Note que precisará retornar um vetor com a metade do tamanho e pior: A cada 2 bytes você deve colocar 4 bits de cada um e um único bit. Exemplo:
Original: 11111111, 00000000, 10101111, 11111010 Compactado: 1111, 0000, 1010, 1010 Resultado: 11110000, 1010101

```
byte[] compact(byte[] originalData)
{
    // Implemente aqui
```

```
4  }
```

Desafio Opcional: Faça uma compactação apenas para 3 bytes, ao invés de 4.

Agora implemente a função de descompactação que retorna os dados aos dados originais, preenchendo os bits que foram perdidos com 0.

```
1  byte[] decompact(byte[] compactedData)
2  {
3      // Implemente aqui
4  }
```

Desafio Opcional: Adicione um parâmetro opcional chamado preenchimento que permite o usuário definir o modo de preenchimento, assim, caso seja 0 (que é o valor padrão), será preenchido com 0. Se for 1, os bits serão preenchidos com 1 (1111 = 15). Caso seja 2, você fará um preenchimento mais inteligente: usará a metade do valor máximo perdido (15 / 2 = 7 = 0111) para minimizar o erro da compreesão.

```
1  byte[] decompact(byte[] compactedData, int fillingMode = 0)
2  {
3      // Implemente aqui
4  }
```

2.8 Aula 4 - Orientação a Objetos básica

2.8.1 Orientação a Objetos

Agora que já compreendemos o básico de C# e como programar no mesmo usando programação estruturada, podemos finalmente aprender como usar a Orientação a Objetos na linguagem, bem como o que é este paradigma de programação.

A Orientação a Objetos é basicamente a separação das implementações e dados de um programa em estruturas chamadas Objetos. Um Objeto é como se fosse qualquer dado que seu computador salva durante a execução de uma aplicação, ou seja, um espacinho na memória com bits de dados. Porém, um objeto pode conter uma grande quantidade de dados, como se fosse um conjunto de informações específicas sobre aquela instância. Outra característica de um objeto é que além de um estado, que é como os dados são apresentados dentro dele, ele também tem várias funções que desempenham uma funcionalidade diferente a depender do estado deste objeto.

Um exemplo poderia ser um cadastro de clientes. Você tem vários clientes, cada um com nome, endereço, cpf, entre outras informações. Em um único bloco de dados você guarda todas essas informações. Perceba que você tem vários objetos que tem a mesma estrutura, porém com dados diferentes. Ainda assim, você tem funções que quando executadas, tem um comportamento diferente a depender do objeto para qual são chamadas. Por exemplo, a função AtualizarEndereco vai mudar o endereço de um cliente específico, mas não de todos. Então o comportamento da função depende exclusivamente do objeto associado.

Isso é bem útil, porque sem esse tipo de ferramenta é difícil expressar a existência de vários objetos estruturalmente parecidos, mas com dados diferentes. Precisaríamos criar vários vetores com todos os dados de forma global (como era feito antigamente) para conseguir criar aplicações dessa natureza.

Para criar um objeto precisamos antes de um modelo, esse modelo especifica estruturalmente quais dados teremos bem como quais funções poderemos executar sobre esses objetos. A partir deste modelo iremos criar vários objetos. Este modelo se chama **Classe**. Abaixo, a forma de criar uma classe cliente como mencionada anteriormente em C#:

```
1  public class Cliente
```

```
{
    public string Nome;
    public string Endereco;
    public long Cpf;

    public void AtualizarEndereco(string novoEndereco)
    {
        Endereco = novoEndereco;
    }
}
```

É importante perceber algumas coisinhas no código acima. Primeiramente a palavra reservada 'public' faz com que tudo que você está colocando seja visto e possa ser utilizado. Segundamente, o código acima não é executável: Você não deve usar dentro de outras funções, dentro de um For ou If. Ele não é executado em momento algum, é apenas uma declaração de estrutura, mas não um código executado linha a linha. As únicas coisas que eventualmente são executadas são as funções colocadas dentro destas classes.
O nome das variáveis que você vê dentro da classe chamam-se **Campos** em C#; já a função, que não segue mais o exemplo de função da matemática já que seu comportamento varia a depender do estado do objeto, recebe o nome de **Método**.
A utilização do código acima é bem simples:

```
using static System.Console;

Cliente cliente1 = new Cliente();
cliente1.Nome = "Gilmar";
cliente1.Endereco = "Rua do Gilmar";
cliente1.Cpf = "12345678-09"

Cliente cliente2 = new Cliente();
cliente2.Nome = "Pamella";
cliente2.Endereco = "Rua da Pamella";
cliente2.Cpf = "87654321-90"

cliente2.AtualizarEndereco("Avenida da Pamella");
```

Note que ao criarmos uma classe, estamos criando um tipo completamente novo. Assim, fazemos duas variáveis cliente1 e cliente2 e usamos a palavra reservada 'new' seguido do nome da classe e parênteses. Note que os dois objetos são diferentes e tem dados diferentes. Usamos a notação de ponto (variável.Campo/Método) para acessar os campos/métodos dentro da classe. Ao executar o AtualizarEndereco do cliente 2 apenas o endereço dele será atualizado.
A capacidade que a OO (Orientação a Objetos) nos da de representar um objeto real com suas características é um dos pilares da OO e chamamos ela de **Abstração**. A partir dessa característica construiremos aplicações bem modularizadas e poderosas.
Para criar uma classe, basta adicionar ao fim do arquivo Top Level, ela será automaticamente considerada fora da função Main.

2.8.2 Padrões de Nomenclatura

Agora que conhecemos a classe, vamos rapidamente entender o padrão de nomenclatura utilizada no C#:

- Para coisas públicas, utilizamos PascalCase: Escrevemos o nome onde cada palavra começa com letra maiúscula.
- Para coisas não-públicas, incluindo variáveis internas de métodos e parâmetros de funções públicas, usamos camelCase: A primeira palavra começa em minúsculo e as próximas em maiúsculo.

Verá que existem momentos em que podemos desrespeitar levemente essas regras, mas em geral, as siga.

2.8.3 Um exemplo OO usando Overload (Sobrecarga)

Overload é uma das capacidades da OO em que dentro de classes podemos ter várias funções com mesmo nome, desde que tenham parâmetros diferentes. Abaixo segue um exemplo legal onde podemos usar a OO para representar uma classe para horários e, adicionalmente, criamos um método Adicionar, que avança no tempo para podermos alterar o horário armazenado.

Importante: Não basta o nome dos parâmetros ser diferente, mas sim a quantidade ou os tipos devem diferir.

```
using static System.Console;

Horario h1 = new Horario();
Horario h2 = new Horario();

h1.Adicionar(20, 40, 30); //h1 = 20:40:30
h2.Adicionar(20, 30); //h2 = 00:20:30

h1.Adicionar(h2); //h1 = 21:01:00, h2 inalterado
h2.Adicionar(h1); //h2 = 00:20:30 + 21:01:00 = 21:21:30, h1 inalterado

WriteLine(h2.Formatar()); //21:21:30

public class Horario
{
    // Valor inicial é 00:00:00
    public int Hora = 0;
    public int Minuto = 0;
    public int Segundo = 0;

    public void Adicionar(int segundos, int minutos, int horas)
    {
        Segundo += segundos;
        if (Segundo > 59)
        {
            minutos += Segundo / 60;
            Segundo = Segundo % 60;
        }

        Minuto += minutos;
        if (Minuto > 59)
        {
            horas += Minuto / 60;
            Minuto = Minuto % 60;
        }

        Hora += horas;
        if (Hora > 23)
        {
            Hora = Hora % 24;
        }
    }

    public void Adicionar(int segundos, int minutos)
    {
        Adicionar(segundos, minutos, 0);
    }

    public void Adicionar(int segundos)
    {
        Adicionar(segundos, 0, 0);
    }

    public void Adicionar(Horario horario)
    {
        Adicionar(horario.Hora, horario.Minuto, horario.Segundo);
    }
```

```
58
59      public string Formatar()
60      {
61          return $"{Hora}:{Minuto}:{Segundo}";
62      }
63  }
```

2.8.4 Construtores

Além disso, podemos fazer construtores, que são funções chamadas no momento que os objetos são criados a partir das classes. É possível ainda usar a sobrecarga de métodos para ter vários construtores diferentes. Esses construtores são usados, em geral, para inicializar os objetos e não costuma ter código muito pesado dentro deles. Observe o exemplo anterior, agora com construtores:

```
1   using static System.Console;
2
3   Horario h1 = new Horario(20, 40, 30); //h1 = 20:40:30
4   Horario h2 = new Horario(); //h2 = 00:00:00
5
6   h2.Adicionar(20, 30); //h2 = 00:20:30
7
8   h1.Adicionar(h2); //h1 = 20:40:30 + 00:20:30 = 21:01:00, h2 inalterado
9   h2.Adicionar(h1); //h2 = 00:20:30 + 21:01:00 = 21:21:30, h1 inalterado
10
11  WriteLine(h2.Formatar()); //21:21:30
12
13  public class Horario
14  {
15      // Valor inicial é 00:00:00
16      public int Hora = 0;
17      public int Minuto = 0;
18      public int Segundo = 0;
19
20      public Horario() // Construtores não tem retorno e o nome é o nome da
    classe
21      {
22          // Vazio não altera os dados
23      }
24
25      public Horario(int horas, int minutos, segundos)
26      {
27          // Geralmente inicializamos 1 a 1, mas aqui preferimos usar a
    função Adicionar que já está pronta
28          // Hora = horas;
29          // Minuto = minutos;
30          // Segundo = segundos;
31          Adicionar(horas, minutos, segundos);
32      }
33
34      public void Adicionar(int horas, int minutos, int segundos)
35      {
36          Segundo += segundos;
37          if (Segundo > 59)
38          {
39              minutos += Segundo / 60;
40              Segundo = Segundo % 60;
41          }
42
43          Minuto += minutos;
44          if (Minuto > 59)
```

```
45          {
46              horas += Minuto / 60;
47              Minuto = Minuto % 60;
48          }
49
50          Hora += horas;
51          if (Hora > 23)
52          {
53              Hora = Hora % 24;
54          }
55      }
56
57      public void Adicionar(int minutos, int segundos)
58      {
59          Adicionar(0, minutos, segundos);
60      }
61
62      public void Adicionar(int segundos)
63      {
64          Adicionar(0, 0, segundos);
65      }
66
67      public void Adicionar(Horario horario)
68      {
69          Adicionar(horario.Hora, horario.Minuto, horario.Segundo);
70      }
71
72      public string Formatar()
73      {
74          return $"{Hora}:{Minuto}:{Segundo}";
75      }
76  }
```

2.8.5 Encapsulamento

Observe o exemplo do horário e responda: Não é perigoso que um programador desavisado tente executar um código como o abaixo?

```
1  horario.Segundo += 1;
```

O código simples somaria um segundo no campo Segundo. O grande problema é que isso poderia fazer com que o valor de segundos chegasse a mais de 60 sem adicionar um valor no minuto. O programador não sabe ou esquece que o horário deve manter-se consistente e que deve utilizar o método Adicionar para fazer isso. E qual é o problema? Isso pode acarretar em Bugs. Este caso é simples, mas o mesmo problema pode escalar de formas astronômicas.

Outra situação: Imagine que você cria um sistema que utiliza uma classe da seguinte forma:

```
1  public class Cliente
2  {
3      public string Login;
4      public string Senha;
5  }
```

Depois de anos seu sistema é comprado e agora exige-se que a senha seja criptografada por questões de segurança. Isso faz com que você faça algo do tipo:

```
1  public class Cliente
2  {
3      public string Login;
4      public string Senha;
5
```

```
    public void DefinirSenha(string value)
    {
        string senhaCriptografada = "";
        // Criptografa o value e guarda na variável senhaCriptografada
        Senha = senhaCriptografada;
    }
}
```

Ainda assim, você esquece de trocar algumas partes do software, as atribuições de senha pela função DefinirSenha e pior ainda, desavisados, os seus colegas acabam realizando implementações esquecendo-se de usar o DefinirSenha. Isso acarreta que agora várias senhas salvas estão sem criptografia e muito pior que isso: Você não sabe exatamente quais estão criptografadas e quais são só estranhas.
Isso pode causar muita dor de cabeça, e gerar muitos bugs. Graças a isso, a OO nos trás o pilar do **Encapsulamento** - o segundo pilar, depois da abstração, de um total de 4 pilares da OO. Para aplicá-lo usaremos a palavra reservada 'private' que esconde as estruturas de um código:

```
using static System.Console;

Cliente c = new Cliente();
c.senha = "Xispita"; //Erro pois senha é agora privada
WriteLine(c.ObterSenha());

public class Cliente
{
    public string Login;
    private string senha; // Letra minúscula (CamelCase), pois agora é privado

    public void DefinirSenha(string value)
    {
        string senhaCriptografada = "";
        // Criptografa o value e guarda na variável senhaCriptografada
        senha = senhaCriptografada;
    }

    public string ObterSenha()
    {
        return senha;
    }
}
```

Na verdade, nenhum campo deve ser público. Devemos usar essas funções de Definir e Obter, quando possível, para acessar os valores. Usamos o inglês Get e Set para as mesmas:

```
using static System.Console;

Cliente c = new Cliente();
c.SetLogin("Gilmar");
c.SetSenha("Xispita");

public class Cliente
{
    private string login;
    private string senha;

    public void SetSenha(string value)
    {
        string senhaCriptografada = "";
        // Criptografa o value e guarda na variável senhaCriptografada
        senha = senhaCriptografada;
```

```
17      }
18
19      public string GetSenha()
20      {
21          return senha;
22      }
23
24      public void SetLogin(string value)
25      {
26          login = value;
27      }
28
29      public string GetLogin()
30      {
31          return login;
32      }
33  }
```

Note que as funções GetLogin e SetLogin parecem inúteis, diferente da senha. Mas é justamente isso que é interessante. Em qualquer momento que precisemos alterar a forma de como o código conversa com os dados de login, basta alterar esses métodos. Ou seja, não precisamos reestruturar todo o código para que as coisas funcionem. Olhe como a vida seria mais fácil se usassemos Get/Set na senha desde o inicio:

```
1   // Antes da adição de criptografia, parecem métodos inúteis
2   private senha;
3   public void SetSenha(string value)
4   {
5       senha = value;
6   }
7
8   public string GetSenha()
9   {
10      return senha;
11  }
12
13  // Depois da adição da criptografia, pequena alteração e não precisa
    alterar mais nada no sistema
14  private senha;
15  public void SetSenha(string value)
16  {
17      string senhaCriptografada = "";
18      // Criptografa o value e guarda na variável senhaCriptografada
19      senha = senhaCriptografada;
20  }
21
22  public string GetSenha()
23  {
24      return senha;
25  }
```

Voltando ao nosso exemplo de Classe Horario, poderíamos reestruturá-la. Note que o Set pode ter uma implementação peculiar:

```
1   public class Horario
2   {
3       private int hora = 0;
4       private int minuto = 0;
5       private int segundo = 0;
6
7       public Horario() { }
```

```
 8
 9      public Horario(int horas, int minutos, int segundos)
10      {
11          Adicionar(horas, minutos, segundos);
12      }
13
14      public int GetHora()
15      {
16          return hora;
17      }
18
19      public int GetMinuto()
20      {
21          return minuto;
22      }
23
24      public int GetSegundo()
25      {
26          return segundo;
27      }
28
29      public void SetHora(int value)
30      {
31          hora = 0;
32          Adicionar(value, 0, 0);
33      }
34
35      public void SetMinuto(int value)
36      {
37          minuto = 0;
38          Adicionar(value, 0);
39      }
40
41      public void SetSegundo(int value)
42      {
43          segundo = 0;
44          Adicionar(value);
45      }
46
47      public void Adicionar(int horas, int minutos, int segundos)
48      {
49          segundo += segundos;
50          if (segundo > 59)
51          {
52              minutos += segundo / 60;
53              segundo = segundo % 60;
54          }
55
56          minuto += minutos;
57          if (minuto > 59)
58          {
59              horas += minuto / 60;
60              minuto = minuto % 60;
61          }
62
63          hora += horas;
64          if (hora > 23)
65          {
66              hora = hora % 24;
67          }
```

```
68      }
69
70      public void Adicionar(int minutos, int segundos)
71      {
72          Adicionar(0, minutos, segundos);
73      }
74
75      public void Adicionar(int segundos)
76      {
77          Adicionar(0, 0, segundos);
78      }
79
80      public void Adicionar(Horario horario)
81      {
82          Adicionar(horario.Hora, horario.Minuto, horario.Segundo);
83      }
84
85      public string Formatar()
86      {
87          return $"{hora}:{minuto}:{segundo}";
88      }
89  }
```

Infelizmente, é cansativo escrever um Get e Set. Se você fosse um programador Java, teria que se contentar com geradores automáticos que ainda deixam o código gigantesco. Felizmente, como desenvolvedor .NET, o C# possui algumas melhorias de sintaxe. Você pode escrever o Get/Set de login, por exemplo, da seguinte forma:

```
1   private string login;
2   public string Login
3   {
4       get
5       {
6           return login;
7       }
8       set
9       {
10          login = value;
11      }
12  }
```

A palavra reservada contextual 'value' pode ser usada no set sem declarar. Além disso, ao usar este get/set você pode usar como se fosse uma variável. Vamos supor que você queira adicionar um "@" no início do login de um usuário. Veja a diferença de código usando o tradicional vs get/set desta forma:

```
1   // Tradicional
2   cliente.SetLogin("@" + cliente.GetLogin());
3
4   // Forma C#
5   cliente.Login = "@" + cliente.Login;
```

O nome desta forma mais simples chama-se propriedade.
Além disso você pode usar a forma autoimplementada. Ela cria o campo privada escondido por trás e implementa da forma mais simples possível o get/set:

```
1   public string Login { get; set; }
```

Você ainda pode deixar o set privado, veja uma possibilidade de implementação para a classe Horario usando esses recursos:

```
1  public class Horario
2  {
3      //É possível ler Hora, Minuto e Segundo, mas alterá-la só com a função
   Adicionar, Construtor ou internalmente na classe
4      public int Hora { get; private set; } = 0;
5      public int Minuto { get; private set; } = 0;
6      public int Segundo { get; private set; } = 0;
7
8      public Horario() { }
9
10     public Horario(int horas, int minutos, int segundos)
11     {
12         Adicionar(horas, minutos, segundos);
13     }
14
15     public void Adicionar(int horas, int minutos, int segundos)
16     {
17         Segundo += segundos;
18         if (Segundo > 59)
19         {
20             minutos += Segundo / 60;
21             Segundo = Segundo % 60;
22         }
23
24         Minuto += minutos;
25         if (Minuto > 59)
26         {
27             horas += Minuto / 60;
28             Minuto = Minuto % 60;
29         }
30
31         Hora += horas;
32         if (Hora > 23)
33         {
34             Hora = Hora % 24;
35         }
36     }
37
38     public void Adicionar(int minutos, int segundos)
39     {
40         Adicionar(0, minutos, segundos);
41     }
42
43     public void Adicionar(int segundos)
44     {
45         Adicionar(0, 0, segundos);
46     }
47
48     public void Adicionar(Horario horario)
49     {
50         Adicionar(horario.Hora, horario.Minuto, horario.Segundo);
51     }
52
53     public string Formatar()
54     {
55         return $"{Hora}:{Minuto}:{Segundo}";
56     }
57 }
```

Assim com o Encapsulamento podemos esconder dados e decidir como eles serão acessados/processados.

2.8.6 Dicas

Você pode, em implemenações de uma única linha, substituir as chaves por uma seta => para deixar o código mais enchuto. Também é recomendável que você utilize a palavra reservada 'this' - que se refere a própria classe - para melhorar a legibilidade quando você acessar algo dentro da propria classe que pertença a mesma:

```
1   public class Horario
2   {
3       public int Hora { get; private set; } = 0;
4       public int Minuto { get; private set; } = 0;
5       public int Segundo { get; private set; } = 0;
6
7       public Horario() { }
8
9       public Horario(int horas, int minutos, int segundos)
10          => Adicionar(horas, minutos, segundos);
11
12      public void Adicionar(int horas, int minutos, int segundos)
13      {
14          this.Segundo += segundos;
15          if (this.Segundo > 59)
16          {
17              minutos += this.Segundo / 60;
18              this.Segundo = this.Segundo % 60;
19          }
20
21          this.Minuto += minutos;
22          if (this.Minuto > 59)
23          {
24              horas += this.Minuto / 60;
25              this.Minuto = this.Minuto % 60;
26          }
27
28          this.Hora += horas;
29          if (this.Hora > 23)
30          {
31              this.Hora = this.Hora % 24;
32          }
33      }
34
35      public void Adicionar(int minutos, int segundos)
36          => Adicionar(0, minutos, segundos);
37
38      public void Adicionar(int segundos)
39          => Adicionar(0, 0, segundos);
40
41      public void Adicionar(Horario horario)
42          => Adicionar(horario.Hora, horario.Minuto, horario.Segundo);
43
44      public string Formatar()
45          => $"{this.Hora}:{this.Minuto}:{this.Segundo}";
46  }
```

2.8.7 Exercício Proposto

Crie uma classe cliente com Nome, Endereço e Cpf. O Cpf deve possuir um campo privado no tipo long que guarda o Cpf como um número. A propriedade get/set do Cpf deve receber uma string na forma "123.456.789-90" e converte-lá num número, depois, no get, o contrário deve ocorrer retornando a string formatada.

2.9 Aula 5 - Desafio 2

Crie um cadastro para um sistema de ponto a ser utilizado em uma empresa. Para isso, seu sistema precisará armazenar um vetor de empregados. O Empregado possui Nome, Cpf, Data de Nascimento, Senha e se ele é o administrador.
Para bater o ponto o empregado deve apenas digitar no sistema seu Cpf, com ou sem pontos, sua senha de 6 caracteres, data e hora para ser armazenados no sistema. Para isso você precisará de uma classe Ponto que recebe o Cpf do empregado a data do ponto e o horário de início ou término do trabalho. Não precisa indicar se foi início ou término: O primeiro ponto do dia é entrada e o segundo é saída. Além disso, para o horário você só precisa de minuto e hora.
Caso seja o administrador a abrir o ponto, deve ser possível que o mesmo escolha entre cadastrar um novo empregado, listar os empregados ou mostrar quantas horas um empregado trabalhou em um determinado mês, digitando seu Cpf, mês e ano.
Use do encapsulamento para impedir que operações sejam usadas indevidamente.
Considere que a empresa só terá no máximo 100 empregados, e só armazenará no máximo 1000 registros de ponto. Use um vetor e o preencha conforme os dados sejam cadastrados.

2.10 Aula 6 - Gerenciando dados e memória

- Heap, Stack e ponteiros
- Tipos por Referência e Tipos por valor
- Class vs Struct
- Null, Nullable e propagação nula
- Associação, Agregação e Composição
- Garbage Collector
- Exemplo 1: Fazendo uma Lista Encadeada com Ponteiros para armazenar uma quantidade variável de clientes

2.10.1 Heap, Stack e ponteiros

Todo programa C# pode guardar seus dados em 2 lugares: O Heap e o Stack. O Heap é uma área na memória onde dados ficam armazenados fora de ordem, e são referenciados para uso. Isso significa que para acessar algo no Heap nós precisamos de algo chamado ponteiro. O ponteiro é uma variável que armazena não o valor, mas sim o endereço de onde algo está no Heap. Pode não parecer, mas ao criarmos um objeto de uma classe estamos usando um ponteiro escondido.
Por outro lado, alguns valores podem ser armazenados na Stack. A stack é uma região onde os valores são colocados sequencialmente. Quando criamos a variável aluno, os dados do objeto Aluno foram colocados no Heap, porém a referência foi colocada na stack. Ao usarmos a variável acessamos a Stack, buscamos a referência e assim vamos buscar os dados na Stack. Não só isso, vários dados como int's e outros tipos primitivos são armazenados na Stack. Quando chamamos uma função, o ponto de retorno, ou seja, para onde a função deve retornar depois de ser executada fica guardada na Stack. E quando chamamos muitas funções a Stack pode crescer tanto que invade a área do Heap resultado no famoso erro Stack Overflow.

2.10.2 Tipos por Referência e Tipos por valor

No C# existem tipos que são por referência e tipos que são por valor. Isso significa que ao chamar uma função ou fazer qualquer atribuição, os dados são copiados se for um tipo por valor, e caso for um tipo por referência, uma referência é armazenada na nova variável para os mesmos dados. Observe o exemplo a seguir que demonstra essa dinâmica:

```
1  using static System.Console;
2
3  int idade = 16;
4  int idade2 = idade;
5  idade2++;
6  WriteLine(idade); // 16
7  WriteLine(idade2); // 17
8
9  Aluno aluno = new Aluno();
10 aluno.Idade = 16;
11 Aluno aluno2 = aluno;
12 aluno2.Idade++;
13 WriteLine(aluno.Idade); // 17
```

```
14  WriteLine(aluno2.Idade); // 17
15
16  Aluno aluno = new Aluno();
17  aluno.Idade = 16;
18  Aluno aluno2 = new Aluno();
19  aluno2.Idade = aluno.Idade;
20  aluno2.Idade++;
21  WriteLine(aluno.Idade); // 16
22  WriteLine(aluno2.Idade); // 17
23
24  public class Aluno
25  {
26      public string Nome { get; set; }
27      public int Idade { get; set; }
28  }
```

Podemos observar que ao passar o valor 16 de uma variável int para outra, esse valor é copiado. Ao alterar o 'idade2', o 'idade' não é alterado pois a variável 'idade2' tem só uma mera cópia do seu valor. No segundo exemplo, criamos um objeto do tipo Aluno que é por referência, ao escrevermos 'aluno2 = aluno', estamos copiando a referência que a variável 'aluno' tem na Stack para a variável 'aluno2'. Já no terceiro exemplo, os dados de idade que estão no Heap são copiados de um objeto para outro, mas dois objetos diferentes são criados.
Ao chamarmos uma função, fenômenos semelhantes acontecem:

```
1   int numero = 76;
2   AlterarNumero(numero);
3   // Aqui número vale 76, seu valor foi copiado e não foi alterado
4
5   Aluno aluno = new Aluno();
6   aluno.Nome = "Pamella";
7   AlterarNome(aluno);
8   // Aqui o nome do Aluno é Gilmar, foi passado a referência do objeto que
    foi alterado dentro da função
9
10  void AlterarNumero(int valor)
11  {
12      valor = valor + 1;
13  }
14
15  void AlterarNome(Aluno aluno)
16  {
17      aluno.Nome = "Gilmar";
18  }
19
20  public class Aluno
21  {
22      public string Nome { get; set; }
23      public int Idade { get; set; }
24  }
```

2.10.3 Class vs Struct

Neste sentido o C# possibilita a utilização de duas estruturas diferentes: a Classe e a Estrutura. Um struct é extremamente parecido com uma classe, sua utilização é semelhante, porém, enquanto a classe cria um tipo por referência, o struct cria um tipo por valor. Você pode ter ganhos de desempenho ao usar struct pois acessar/remover dados da Stack é muito mais rápido do que no Heap. Abaixo um exemplo de uso de struct:

```
1   Aluno aluno = new Aluno("Gilmar", 25);
2
3   public struct Aluno
```

```
{
    public Aluno(string nome, int idade)
    {
        this.Nome = nome;
        this.Idade = idade;
    }

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

Podemos ver que o uso é idêntico a classe, só trocamos a palavra reservada e pronto: Temos um tipo por valor. Neste ponto fica cada vez mais evidente que int, byte, long, float, bool, char, entre outros tipos, são structs quando analisados por baixo dos panos.
Todos os vetores, incluindo string que é um vetor de caracteres são tipos por referências e são armazenados no Heap.
Lembrando que, tipos por valor que estiverem dentro de objetos por refência estarão no Heap, pois estarão dentro de coisas que ficam dentro do Heap.

2.10.4 Null, Nullable e propagação nula

Para tipos por referência, você pode ainda criar ponteiros que não apontam para nada. Isso é útil quando queremos criar variáveis que ainda não tem valor algum. Para isso usamos a palavra reservada 'null'.

```
Aluno aluno = null;

var nome = aluno.Nome; // Erro, aluno é nulo e não podemos ler seu nome, o famoso NullPointerException (erro /exceção de ponteiro nulo)
```

Isso ajuda a inicializar objetos mas também trará uma dor de cabeça (inevitável) ao usar ponteiros que não foram ainda atribuidos com referências. Ainda podemos criar tipos nulos a partir de tipos por valor. Abaixo uma implementação de como isso seria feito:

```
//int j = null; // Erro de compilação: int é um tipo por valor e não pode receber nulo
int? i = null; // OK

// Testa para ver se i é nulo, caso seja, atribui um valor
if (!i.HasValue)
    i = 76;

int valorReal = i.Value; // Da erro se i for nulo
```

Ainda que sejam ferramentas úteis, por vezes (em várias linguagens) os nulos trazem grande dor de cabeça. Observe o exemplo a seguir:

```
Aluno aluno = null;
string nome = aluno.Nome; // NullPointerException

public struct Aluno
{
    public string Nome { get; set; }
    public int Idade { get; set; }
}
```

Para tratar isso C# trás uma brilhante feature que é a propagação nula:

```
Aluno aluno = null;
string nome = aluno?.Nome; // Se aluno for nulo, não temos um erro, mas .Nome retornará nulo, automaticamente.

public struct Aluno
```

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

Além disso, temos ainda outra tratativa para valores nulos que é bem interessante:

```
Aluno aluno = null;
string nome = aluno?.Nome ?? "Pamella"; // Se aluno?.Nome resultar em nulo, temos o valor padrão "Pamella".

public struct Aluno
{
    public string Nome { get; set; }
    public int Idade { get; set; }
}
```

2.10.5 Associação, Agregação e Composição

Além dos tipos que usamos para Campos e Propriedades de uma classe, também podemos usar um ponteiro a outros tipos. O nome disso é Associação, quando dois tipos estão ligados por uma referência. Ainda temos a Agregação e Composição que essencialmente são a mesma coisa mas com formas de utilização diferente. Enquanto a Associação é fraca e objetos sabem que outros existem mas são independentes, uma Agregação ocorre quando um objeto B está referenciado por um objeto A e o segundo não faz sentido sozinho como objeto sem o A. Uma Composição é quando A não faz sentido sem o B. Abaixo um exemplo de Agregação: A data não faz sentido se não tiver um significado associado ao cliente, mas o cliente continua a ser um cliente mesmo sem data de aniversário.

```
Cliente cliente = new Cliente();
cliente.Nome = "Pamella";

Data data = new Data();
data.Dia = 7;
data.Mes = 6;
data.Ano = 2000;

cliente.DataNascimento = data;

public class Data
{
    public int Dia { get; set; }
    public int Mes { get; set; }
    public int Ano { get; set; }
}

public class Cliente
{
    public string Nome { get set; }
    public Data DataNascimento { get; set; }
}
```

O importante desta discussão é apenas perceber que é possível usar seus tipos como variáveis dentro de outras classes e fazer estruturas muito complexas. Note que quando fazemos isso com classes estamos falando de ponteiros. Quando usamos estruturas estamos falando dos dados em si. Isso significa que se eu fizer uma struct A com uma propriedade do tipo A estaria colocando um loop infinito de dados, pois A está dentro do A que tem um A dentro e assim por diante. Isso resulta em um erro de compilação.

2.10.6 Garbage Collector

Para finalizar estes tópicos avançados sobre memória e ponteiros, é importante mencionar o Garbage Collector, Coletor de Lixo ou simplesmente GC. O GC é basicamente um subsistema do .NET que tem a missão de limpar dados não utilizados ao longo do tempo.

Em muitas linguagens você faz o gerenciamento de memória. Isso significa que se você criar dados dinâmicos, caso você esqueça de apagá-los, eles ficarão para sempre na sua memória até que a aplicação seja finalizada. Isso é o que chamamos de Memory Leak e que faz com que algumas aplicações fiquem muito mais pesadas do que deveriam após intenso uso.
O GC automatiza isso para você. Todas as informações gerenciadas, ou seja, que ficam no Heap são então gerenciadas pelo GC e quando não existem mais ponteiros para esses dados, eles são automaticamente limpos.
É claro que isso gera alguns efeitos colaterais. Abusar do Heap pode deixar sua aplicação muito mais lenta e criar gargalos por muitos motivos. Um deles é a desfragmentação, que é o que ocorre quando o GC move os dados para tirar buracos que aparecem quando dados são limpos no meio da memória. É um processo pesado, mas necessário para otimizar seu uso.
Isso nos torna ainda mais fãs dos dados não gerenciados, que podemos usar utilizando tipos por valor em funções que ficarão na Stack, que não é gerenciada pelo GC.

2.10.7 Exemplo 1: Fazendo uma Lista Encadeada com Ponteiros para armazenar uma quantidade variável de clientes

```
1  using static System.Console;
2
3  LinkedList clientes = new LinkedList();
4
5  Cliente cliente1 = new Cliente();
6  cliente1.Nome = "Gilmar";
7  clientes.Add(cliente1);
8
9  Cliente cliente2 = new Cliente();
10 cliente2.Nome = "Pamella";
11 clientes.Add(cliente2);
12
13 // Se Get retornar null, Nome deve retornar null ao invés de estourar um
   erro, se Nome retornar null deve-se substituir pelo valor padrão
14 // 'Cliente não encontrado'
15 var result = clientes.Get(1)?.Nome ?? "Cliente não encontrado";
16 WriteLine(result);
17
18 // Cadastrando uma quantidade variável de clientes
19 string input = ReadLine();
20 while (input != "")
21 {
22     Cliente newClient = new Cliente();
23     newClient.Nome = input;
24     newClient.Cpf = long.Parse(ReadLine());
25     clientes.Add(newClient);
26
27     input = ReadLine();
28 }
29
30 WriteLine("Procure um  cpf de cliente pelo nome:");
31 input = ReadLine();
32 for (int i = 0; i < clientes.Count; i++)
33 {
34     var pesquisa = clientes.Get(i);
35     if (pesquisa?.Nome == input)
36     {
37         WriteLine(pesquisa.Cpf);
38         break;
39     }
40 }
```

```
41
42  // Classe cliente a qual queremos armazenas
43  public struct Cliente
44  {
45      public string Nome { get; set; }
46      public long Cpf { get; set; }
47  }
48
49  // Um nó representa um valor na lista com um ponteiro para o próximo valor
50  public class LinkedListNode
51  {
52      public Cliente Value { get; set; }
53      public Node Next { get; set; }
54  }
55
56  public class LinkedList
57  {
58      // Ponteiro vazio = lista vazia
59      private LinkedListNode first = null;
60
61      public int Count { get; private set; }
62
63      // Função para adicionar um valor no final da lista
64      public void Add(Cliente value)
65      {
66          Count++;
67
68          // Se first for nulo, vamos inicializá-lo com um novo elemento
69          if (first == null)
70          {
71              first = new LinkedListNode();
72              first.Value = value;
73              return;
74          }
75
76          // Caso first != null precisamos buscar o último elemento da lista
    para então
77          // preenche-lo
78
79          // Nó atual
80          var crr = first;
81          // Enquanto existir um próximo, avance para ele
82          while (crr.Next != null)
83              crr = crr.Next;
84
85          // Aqui crr.Next é nulo
86          var next = new LinkedListNode();
87          next.Value = value;
88          crr.Next = next;
89      }
90
91      // Função para ler em uma posição específica do vetor, retornal null
    se o indice for inválido (fora da lista)
92      public Cliente? Get(int index)
93      {
94          if (index < 0)
95              return null;
96
97          // Busca até atingir o índice ou ter crr nulo
98          var crr = first;
```

```
        for (int i = 0; i < index && crr != null; i++)
            crr = crr.Next;

        // Se crr for nulo, retorna nulo, caso contrário retorna seu Value
        return crr?.Value;
    }
}
```

2.11 Aula 7 - Listas e Genéricos

- Listas
- This e Indexação
- Genéricos
- Exercícios Propostos

2.11.1 Listas

Talvez você tenha percebido no desafio 6 a dificuldade de trabalhar com uma quantidade arbitrária de dados. Usamos um vetor de tamanho fixo e gerenciamos o seu uso. Com a ajuda da OO isso pode ficar mais fácil, pois podemos implementar estruturas que façam isso para a gente e usar seus objetos. Dito isso, vamos implementar agora uma matriz dinâmica que é, basicamente, um vetor que cresce todas as vezes que você precisa de um vetor maior. Diferente da lista encadeada vista do Exemplo 1, está lista é mais rápida ao se acessar um valor qualquer (pois estamos falando de um vetor, afinal de contas), mas para muitas adições pode ter desempenho comprometido. Observe a implementação com cuidado:

```
using static System.Console;

ListInt list = new ListInt();
list.Add(4);
list.Add(10);
list.Add(76);

for (int i = 0; i < list.Count; i++)
{
    WriteLine(list.Get(i));
}

public class ListInt
{
    private int pos = 0;
    private int[] vetor = new int[10];

    public void Add(int value)
    {
        int len = vetor.Length;
        // Vetor estouraria, vamos aumentá-lo
        if (pos == len)
        {
            // Criamos um vetor maior
            int[] newVetor = new int[2 * len];

            // Copiamos
            for (int i = 0; i < pos; i++)
                newVetor[i] = vetor[i];

            // Jogamos a referência antiga fora e agora o ponteiro 'vetor'
irá apontar para um novo vetor
            vetor = newVetor;
        }

```

```
35          vetor[pos] = value;
36          pos++;
37      }
38
39      public int Count => pos;
40
41      public void Set(int i, int value)
42      {
43          this.vetor[i] = value;
44      }
45
46      public int Get(int i)
47      {
48          return this.vetor[i];
49      }
50  }
```

2.11.2 This e Indexação

Outra ferramenta que o C# dispõe é a indexação. Essa é a implementação que permite que você crie seus próprios tipos que possam usar os colchetes '[]' para acessar um dado. Para isso, basta criar uma propriedade com a palavra reservada 'this'. Observe a melhoria no código anterior:

```
1   using static System.Console;
2
3   ListInt list = new ListInt();
4   list.Add(4);
5   list.Add(10);
6   list.Add(76);
7
8   for (int i = 0; i < list.Count; i++)
9   {
10      // Usamos como se fosse um vetor, extremamente mais legível
11      WriteLine(list[i]);
12  }
13
14  public class ListInt
15  {
16      private int pos = 0;
17      private int[] vetor = new int[10];
18
19      public int this[int index]
20      {
21          get => this.vetor[index];
22          set => this.vetor[index] = value;
23      }
24
25      public void Add(int value)
26      {
27          int len = vetor.Length;
28          // Vetor estouraria, vamos aumentá-lo
29          if (pos == len)
30          {
31              // Criamos um vetor maior
32              int[] newVetor = new int[2 * len];
33
34              // Copiamos
35              for (int i = 0; i < pos; i++)
36                  newVetor[i] = vetor[i];
37
```

```
            // Jogamos a referência antiga fora e agora o ponteiro 'vetor'
irá apontar para um novo vetor
            vetor = newVetor;
        }

        vetor[pos] = value;
        pos++;
    }

    public int Count => pos;
}
```

2.11.3 Genéricos

Um problema do código acima é claro: Só funciona para int. Isso é um grande problema, pois para cada lista que quisermos fazer precisamos de uma nova implementação. Mas como é de se esperar, o C# tem uma solução a este imenso problema: Os genéricos. Bascimente, podemos criar um parâmetro que recebe um tipo e varia suas funcionalidade a partir disso. Observe o exemplo simplificado:

```
Caixa<int> caixa1 = new Caixa<int>();
Caixa<string> caixa2 = new Caixa<string>();

caixa1.Conteudo = 76; // Essa variável é um int
caixa2.Conteudo = "Pamella"; // Já está é uma string

public class Caixa<T> // Parâmetro Genérico
{
    public T Conteudo { get; set; } // Variável Genérica
}

Podemos aplicar o mesmo principio a nossa lista:

using static System.Console;

List<int> list = new List<int>();
list.Add(4);
list.Add(10);
list.Add(76);

for (int i = 0; i < list.Count; i++)
{
    // Usamos como se fosse um vetor, extremamente mais legível
    WriteLine(list[i]);
}

public class List<T>
{
    private int pos = 0;
    private T[] vetor = new T[10]; // Vetor do tipo T

    public T this[int index]
    {
        get => this.vetor[index];
        set => this.vetor[index] = value;
    }

    public void Add(T value)
    {
        int len = vetor.Length;
        // Vetor estouraria, vamos aumentá-lo
```

```
42          if (pos == len)
43          {
44              // Criamos um vetor maior
45              T[] newVetor = new T[2 * len];
46
47              // Copiamos
48              for (int i = 0; i < pos; i++)
49                  newVetor[i] = vetor[i];
50
51              // Jogamos a referência antiga fora e agora o ponteiro 'vetor'
    irá apontar para um novo vetor
52              vetor = newVetor;
53          }
54
55          vetor[pos] = value;
56          pos++;
57      }
58
59      public int Count => pos;
60  }
```

2.11.4 Exercícios Propostos

1. Agora é sua vez. Use a nossa classe List genérica como base para implementar a classe Stack (sim, que nem a estrutura do C#). Stack é uma pilha e ela funciona desta maneira, como uma pilha de roupas. Você coloca e tira coisas apenas do topo, nunca debaixo. Ela possui os seguintes componentes:
    a. Push: Adiciona um valor ao topo da pilha
    b. Pop: Remove e retorna o valor no topo da pilha
    c. Peek: Retorna, sem remover, o valor no topo da pilha
    d. Count: Tamanho da pilha
2. Depois disso implementaremos a Queue - uma fila. Na fila, entram coisas de um lado e se retiram do outro:
    a. Enqueue: Adiciona um valor no início da fila
    b. Dequeue: Remove e retorna o valor no final da fila
    c. Peek: Retorna, sem remover, o valor no final da fila
    d. Count: Tamanho da fila

2.12 Aula 8 - Herança e Polimorfismo

- Herança
- Métodos virtuais e sobrescrita
- Classes abstratas
- Polimorfismo
- Object
- Exemplo 2: Batalha de Bots no Jogo da Velha e Geração de Números Randômicos

2.12.1 Herança

Na OO temos 4 pilares principais. Quatro conceitos primordiais que conduzem bastante o comportamento e a forma de raciocínio usado em uma aplicação OO. Já falamos sobre a Abstração e o Encapsulamento, agora iremos falar sobre o que é a Herança em um código OO.
A Herança é a capacidade de um objeto de herdar todas as características de outro. Para isso, basta que sua classe indique de qual outra classe essas características devem ser herdadas. Observe um rápido exemplo:

```
1  using static System.Console;
2
3  A a = new A();
4  B b = new B();
5
```

```
WriteLine(a.Value); // 2
WriteLine(a.OtherValue); // Erro: Other Value não existe em A

WriteLine(b.Value); // 2
WriteLine(b.OtherValue); // 3

public class A
{
    public int Value { get; set; } = 2;
}

public class B : A
{
    public int OtherValue { get; set; } = 3;
}
```

B não tinha a propriedade Value, mas ao digitarmos 'B : A' estamos dizendo que tudo que A tem, B também tem. Assim, B agora tem Value e tudo mais que A possa implementar, inclusive métodos. Note que isso não faz com que A seja alterada de maneira alguma.
Podemos utilizar a Herança de muitas formas, mas em geral devemos fazer a pergunta "é um?". Ou seja, se o objeto B é um A, isso significa que B tem tudo que A tem e assim a herança é valida. Mas isso não é 100% perfeito que torna a Herança alvo de várias críticas. Vamos observar 3 exemplos de herança e avaliar isso por nós mesmos. A Herança é a capacidade de um objeto herdar todas as características de outro. Para isso, basta que sua classe indique de qual outra classe essas características devem ser herdadas. Observe um rápido exemplo:

```
using static System.Console;

Gato gato = new Gato("Edjalma");
Cao cao = new Cao("Cesar");

public class Pet
{
    public Pet(string nome)
    {
        this.Nome = nome;
    }

    public string Nome { get; set; }
}

public class Cao : Pet
{
    // A classe Pet precisa de um nome para ser construído. Assim todas as suas classes bases precisam
    // de um construtor equivalente. Caso contrário obtemos um erro. Você pode ainda usar ': base(nome)'
    // para chamar a funcionalidade do construtor da classe mãe (Pet).
    public Cao(string nome) : base(nome) { }

    public void Latir()
    {
        WriteLine("Au!");
    }
}

public class Gato : Pet
{
    public Gato(string nome) : base(nome) { }

```

```
    public void Miar()
    {
        WriteLine("Miau!");
    }
}
```

```
using static System.Console;

public class Conta
{
    private decimal saldo;

    public bool Saque(decimal value)
    {
        if (value > saldo)
            return false;

        saldo -= value;

        return true;
    }
}

public class Debito : Conta
{
    // saldo não pode ser visto aqui pois ela é privada na classe base só podemos acessar o dado
    // usando o método Saque
    public bool Pagar(decimal value)
        => Saque(value);
}

public class Credito : Conta
{
    private decimal limite = 1000;
    private decimal fatura = 0;

    public bool Pagar(decimal value)
    {
        if (fatura + value > limite)
            return false;

        fatura += value;
    }

    public bool PagarFatura()
    {
        bool pago = Saque(value);

        if (pago)
            fatura = 0;

        return pago;
    }
}
```

```
using static System.Console;

RelogioDeXadrez relogio = new RelogioDeXadrez();
relogio.Acrescimo = 2;
relogio.Iniciar(0, 5, 0);

while (true)
{
    WriteLine(relogio.Minuto + ":" + relogio.Segundo);
    relogio.Tick();
}

public class Relogio
{
    public int Segundo { get; private set; }
    public int Minuto { get; private set; }
    public int Hora { get; private set; }

    // Semelhante a um valor privado, porém pode ser visto também nas classes filhas
    protected void adicionar(int segundos, int minutos, int horas)
    {
        this.Segundo += segundos;
        if (this.Segundo > 59)
        {
            minutos += this.Segundo / 60;
            this.Segundo = this.Segundo % 60;
        }

        this.Minuto += minutos;
        if (this.Minuto > 59)
        {
            horas += this.Minuto / 60;
            this.Minuto = this.Minuto % 60;
        }

        this.Hora += horas;
        if (this.Hora > 23)
        {
            this.Hora = this.Hora % 24;
        }
    }

    // Semelhante a um valor privado, porém pode ser visto também nas classes filhas
    protected void remover(int segundos, int minutos, int horas)
    {
        this.Segundo -= segundos;
        if (this.Segundo < 0)
        {
            minutos -= this.Segundo / 60;
            this.Segundo = -(this.Segundo % 60);
        }

        this.Minuto -= minutos;
        if (this.Minuto < 0)
        {
            horas -= this.Minuto / 60;
            this.Minuto = -(this.Minuto % 60);
```

```
58          }
59
60          this.Hora -= horas;
61          if (this.Hora < 0)
62          {
63              this.Hora = 0;
64          }
65      }
66
67      // Passa 1 segundo
68      public void Tick()
69      {
70          adicionar(1, 0, 0);
71      }
72  }
73
74  public class Timer : Relogio
75  {
76      // Tem o mesmo nome da função Tick da classe base, assim ela 'esconde'
    a existência da antiga
77      // função
78      public void Tick()
79      {
80          remover(1, 0, 0);
81          if (this.Hora == 0 && this.Minuto == 0 && this.Segundo == 0)
82              Apitar();
83      }
84
85      public void Zerar()
86      {
87          remover(this.Segundo, this.Minuto, this.Hora);
88      }
89
90      public void Iniciar(int hora, int minuto, int segundo)
91      {
92          Zerar();
93          adicionar(segundo, minuto, hora);
94      }
95
96      protected void Apitar()
97      {
98          WriteLine("O tempo acabaou");
99      }
100 }
101
102 public class RelogioDeXadrez : Timer
103 {
104     public int Acrescimo { get; set; }
105
106     public void JogadaFeita()
107     {
108         adicionar(Acrescimo);
109     }
110 }
```

O terceiro exemplo apresenta-se muito interessante, mostrando que: podemos esconder métodos; existe uma terceira palavra reservada como modificador de acesso chamada 'protected'; e mostrando também que podemos fazer uma hierarquia completa de heranças. O RelogioDeXadrez herda de Timer que herda de Relógio.

2.12.2 Métodos virtuais e sobrescrita

Melhor que esconder é sobrescrever. Quando um método é marcado com a palavra reservada 'virtual' isso permite que você mude o comportamento deste método nas suas classes filhas usando a palavra reservada 'override'. Observe:

```
1  using static System.Console;
2
3  Printer printer = new Printer();
4  BeautyPrinter beauty = new BeautyPrinter();
5
6  printer.Print("Xispita");
7  // A seguir uma mensage:
8  // Xispita
9
10 beauty.Print("Xispita");
11 // --------------------
12 // Xispita
13 // --------------------
14
15 public class Printer
16 {
17     protected virtual void printSuperior()
18     {
19         WriteLine("A seguir uma mensage:");
20     }
21
22     protected virtual void printInferior()
23     {
24
25     }
26
27     public void Print(string message)
28     {
29         printSuperior();
30         WriteLine(message);
31         printInferior()
32     }
33 }
34
35 public class BeautyPrinter : Printer
36 {
37     protected override void printSuperior()
38     {
39         WriteLine("--------------------");
40     }
41
42     protected override void printInferior()
43     {
44         WriteLine("--------------------");
45     }
46 }
```

2.12.3 Classes abstratas

Além disso, você pode criar classes abstratas - classes que não podem ser instanciadas. Isso é perfeito para situações onde a classe mãe não existe na prática. Você ainda pode colocar implementações abstratas na classe. Isso significa que você pode adicionar um método abstrato que você não implementa, apenas declara. Todas as classes base são obrigadas a implementar aquela função. Isso é extremamente útil em muitos cenários. Observe um interessante exemplo:

```
public abstract class Language
{
    public abstract string TranslateNewGame();
    public abstract string TranslateQuit();
    public abstract string TranslateLoadGame();
    public abstract string TranslateOptions();
}

public class English : Language
{
    public override string TranslateNewGame()
        => "New Game";
    public override string TranslateQuit()
        => "Quit";
    public override string TranslateLoadGame()
        => "Load Game";
    public override string TranslateOptions()
        => "Options";
}

public class Portuguese : Language
{
    public override string TranslateNewGame()
        => "Novo Jogo";
    public override string TranslateQuit()
        => "Sair";
    public override string TranslateLoadGame()
        => "Carregar Jogo";
    public override string TranslateOptions()
        => "Opções";
}
```

2.12.4 Polimorfismo

Além disso, existe outro recurso poderosissímo na Orientação a Objetos que é o nosso quarto pilar: O **Polimorfismo**. O Polimorfismo é a capacidade de uma referência/variável apresentar diferentes comportamentos a depender do objeto nele contido, apesar de seu tipo ser um único. Isso será possível pois existe um conceito chamado **Variância**. A Variância permite que coloquemos objetos da classe filha em uma variável da classe mãe. Considerando o exemplo de tradução visto acima observe que isso seria possível:

```
Language lang = null;

WriteLine("Select your language:");
WriteLine("1 - English")
WriteLine("2 - Portugues")
var selected = ReadLine();

if (selected == "1")
{
    lang = new English();
}
else if (selected == "2")
{
    lang = new Portuguese();
}
else
{
    WriteLine("Error: Unknown Input.");
```

```
    return;
}

WriteLine($"1 - {lang.TranslateNewGame()}");
WriteLine($"2 - {lang.TranslateLoadGame()}");
WriteLine($"3 - {lang.TranslateOptions()}");
WriteLine($"4 - {lang.TranslateQuit()}");
```

Embora seja um variável do tipo Language que nem mesmo pode ser instanciado, lang tem comportamentos distintos a depender do objeto que ele recebe. Isso é o fênomeno chamado de Polimorfismo. Muito melhor do que criar uma variável que armazena o id da linguagem (1, 2, etc.) e realizar um 'if' toda vez que quiser escrever algo.

2.12.5 Object

Isso tudo os leva ao object, um tipo fundamental no C#. Todos os objetos C# herdam de object, e consequentemente tem tudo que nele tem e podem ser colocados em variáveis do tipo:

```
Cliente client = new Cliente();
object obj = client;

client.Nome = "Gilmar";
obj.Nome = "Pamella"; // Erro: Object não contém uma definição de 'Nome'

Cliente x = (Cliente)obj; // Se obj não tive um objeto do tipo cliente estourará um erro
x.Saldo = 100000;

public class Cliente
{
    public string Nome { get; set; }
    public decimal Saldo { get; set; }
}
```

2.12.6 Exemplo 2: Batalha de Bots no Jogo da Velha e Geração de Números Randômicos

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

int randomWins = 0;
int smartWins = 0;
int ties = 0;

RandomBot randomBot = new RandomBot();
SmartLineBot smartLineBot = new SmartLineBot();

for (int i = 0; i < 100; i++)
{
    Game game = new Game();
    bool randomIsX = i % 2 == 0;
    Bot X = randomIsX ? randomBot : smartLineBot;
    Bot O = !randomIsX ? randomBot : smartLineBot;
    Bot crr = X;
    PlayResult result = new PlayResult();

    do
    {
        if (crr == X)
```

```
23          {
24              game.Play(X, true);
25              crr = O;
26          }
27          else
28          {
29              game.Play(O, false);
30              crr = X;
31          }
32      } while (!result.XWins && !result.OWins && !result.Tie &&
    result.IsValid);
33
34      if (result.XWins && randomIsX)
35          randomWins++;
36      else if (result.OWins && !randomIsX)
37          randomWins++;
38      else if (result.Tie || !result.IsValid)
39          ties++;
40      else smartWins++;
41  }
42
43  WriteLine("Placar:");
44  WriteLine($"RandomBot ganhou {randomWins} vezes.");
45  WriteLine($"SmartLineBot ganhou {smartWins} vezes.");
46  WriteLine($"Empatou {ties} vezes.");
47
48  // Representa o resultado de uma jogada
49  public struct PlayResult
50  {
51      public bool IsValid { get; set; } = false;
52      public bool XWins { get; set; } = false;
53      public bool OWins { get; set; } = false;
54      public bool Tie { get; set; } = false;
55  }
56
57  // Representa um movimento que pode ser feito
58  public struct Move
59  {
60      public int Line { get; set; }
61      public int Column { get; set; }
62  }
63
64  // Representa um Bot que joga jogo da velha
65  public abstract class Bot
66  {
67      public abstract Move PlayGame(int[] game, bool isX);
68  }
69
70  public class RandomBot : Bot
71  {
72      // Inicializa um objeto que gera valores pseudo-aleatórios
73      private Random rand = new Random();
74      public override Move PlayGame(int[] game, bool isX)
75      {
76          // Gera um índice entre 0 e 8
77          int index = rand.Next(9);
78
79          // Procura uma posição sem jogada
80          while (game[index] != 0)
81              index = rand.Next(9);
```

```
82
83          Move move = new Move();
84          move.Line = index / 3;
85          move.Column = index % 3;
86
87          return move;
88      }
89  }
90
91  public class SmartLineBot : Bot
92  {
93      // Tem um random bot internamente
94      private RandomBot randBot = new RandomBot();
95
96      public override Move PlayGame(int[] game, bool isX)
97      {
98          // Defende ou ataca se alguma linha está quase completa
99          for (int i = 0; i < 3; i++)
100         {
101             for (int j = 0; j < 3; j++)
102             {
103                 // Compara as posições 0 e 1, 1 e 2, 2 e 0
104                 // Note que as 3 posições da linha podem ser representadas
    como:
105                 // i, (i + 1) % 3, (i + 2) % 3
106                 if (game[i + 3 * j] == game[((i + 1) % 3) + 3 * j])
107                 {
108                     Move move = new Move();
109                     move.Line = (i + 2) % 3;
110                     move.Column = j;
111                     return move;
112                 }
113             }
114         }
115
116         // Caso contrário, joga como um bot aleatório
117         return randBot.PlayGame(game, isX);
118     }
119 }
120
121 // Representa o jogo da velha
122 public class Game
123 {
124     private byte[] game = new byte[9];
125
126     // Iremos chegar a vitória de um jeito diferente. O vetor representa
    os seguines valores:
127     // [ Gap X na linha 1, Gap X na linha 2, Gap X na linha 3, Gap X na
    coluna 1, Gap X na coluna 2
128     // Gap X na colune 3, Gap X na diagonal 1, Gap X na diagonal 2]
129     // Onde Gap X é quantos X temos a mais que O. Para que um deles ganhem
    basta que o Gap seja +3 ou -3
130     // em qualquer lugar do vetor.
131     private byte[] winChecker = new sbyte[8];
132     private int playCount = 0;
133
134     public PlayResult Play(Bot bot, bool isX)
135     {
136         var move = bot.PlayGame(this.game, isX);
137         return Play(move.Line, move.Column);
```

```
    }

    public PlayResult Play(int i, int j, bool isX)
    {
        PlayResult result = new PlayResult();
        if (game[i + 3 * j] != 0)
            return result;

        game[i + 3 * i] = isX ? 1 ? 2;

        sbyte value = (sbyte)(isX ? 1 : -1);
        game[i] += value;
        game[3 + j] += value;
        if (i == j)
            game[7] += value;
        if (i + j == 2)
            game[8] += value;

        for (int i = 0; i < 8; i++)
        {
            if (winChecker[i] == 3)
                result.XWins = true;

            if (winChecker[i] == -3)
                result.OWins = true;
        }

        if (result.XWins || result.OWins)
            return result;

        playCount++;
        if (playCount == 9)
           result.Tie = true;

        return result;
    }
}
```

2.13 Aula 9 - Desafio 3

Um jogo Clicker é basicamente um jogo onde o jogador deve clicar para conseguir algum determinado recurso. Ao conquistar bastante recurso o jogador pode trocar este recurso por máquinas que ajudam-o a produzir ainda mais deste recurso.
Implemente um Bosch Clicker onde a cada clique você produz um bico injetor. Você pode consultar a loja para trocar bicos injetores por várias máquinas famosas (que você pode escolher). Cada máquina deve ter um preço, um valor de produção por segundo, quanto já foi produzida com está máquina, quanto ela suporta e se ela está quebrada.
O jogador deve abrir o maquinário para ver suas máquinas e resgatar os bicos produzidos. Para saber quanto cada máquina produziu você deve usar o struct do System chamado DateTime, onde ao acessar o DateTime.Now você pode obter o momento atual do sistema. Use isso, salvando a última hora de resgate para saber quanto cada máquina produziu. Além disso, cada máquina suporta um limite.
Exemplo:
Você compra um Torno CNC por 100 bicos injetores. Essa máquina produz 1 bico injetor por segundo e pode armazenar até 60 bicos injetores. Após 40 segundos você abre seu maquinário e busca ela e clicar em resgatar (ou aperta o uma tecla como 'R', por exemplo) e recebe os 40 bicos, agora a máquina está zerada e já produziu 40 bicos. Após 100 segundos você retorna a máquina. Embora tenham passado 100 segundos, a máquina só produziu 60, que é o limite dela. Você resgata e ganha 60 bicos injetores, agora a máquina aponta que produziu 40 + 60 = 100.

As máquinas sorteiam momentos para quebrar, e ao acontecer isso ela perde todos os bicos injetores. Ou seja, se passar muito tempo elas quebram e você deve abri-las na tela do maquinário e clicar em consertar (ou apertar o uma tecla como 'C', por exemplo).

> *Desafio Opcional: Faça com que a máquina ao quebrar só não faça mais bicos injetores, sem perder os antigos. Para isso uma pequena correção de cálculo deve bastar.*

A qualquer momento você pode abrir a loja e comprar mais máquinas, fortalecendo o seu maquinário.

> *Desafio Opcional: Faça a tela de RH que possibilita a contratação de funcionários. Os funcionários, a cada clique seu, visitam máquinas aleatórias e consertam elas se elas estiverem quebradas, caso contrário, eles resgatam os bicos injetores automaticamente.*

Dica: Subtrair dois DateTimes irá retornar um TimeSpan que diz quanto tempo se passou entre os DateTimes (faça final - inicial).

2.14 Aula 10 - Desafio 4

Em 2017 Nicky Case criou um jogo para estudar a Teoria dos Jogos que ficou popular no YouTube e redes sociais. The Evolution of Trust é um jogo onde analisamos como a sociedade se comporta e como ela evolui diante de exercícios de confiança que acontecem no dia-a-dia. Neste desafio iremos implementar uma versão semelhante ao jogo de Nicky Case que estudará a sobrevivência da sociedade perante as oportunidades de cooperação.

Iremos implementar um sistema de prosperidade que funciona da seguinte forma:

- Existe uma população mundial que começa com uma quantidade qualquer de indivíduos
- Cada indivíduo começa com 10 moedas
- Indivíduos podem interagir, representando uma oportunidade de cooperar
- Quando indivíduos interagem o seguinte processo acontece:
    - Eles devem escolher colocar ou não uma moeda em uma máquina
    - Se um indivíduo coloca a moeda na máquina, o mesmo perderá a moeda
    - Se ambos colocarem a moeda da máquina, ou seja, cooperarem, ambos ganham duas moedas
    - Se algum deles trapacear, não colocando a moeda na máquina, mas o outro indivíduo colocar, o trapaceiro ganha 4 moedas
    - Se ambos trapacearem, ninguém ganha moeda alguma
- Em cada rodada os indivíduo irão interagir da seguinte forma:
    - A escolha dos indivíduos que vão interagir podendo ou não cooperar é aleatória
    - Numa única rodada ocorre 1 interação para cada 2 indivíduos
    - Alguns indivíduos tem a oportunidade de interagir mais de uma vez, outros nenhuma
- No final da rodada todos os indivíduos perdem 1 moeda como custo de sobrevivência
- Todo indivíduo tem 10% de chance de perder uma moeda por puro azar
- Se algum indivíduo não puder pagar o custo de sobrevivência, ele morre e sai do jogo
- Após isso, se algum indivíduo obter 20 moedas, ele se clona dividindo suas moedas entre ele e seu novo clone
- Se restar apenas um indivíduo, considere o fim do jogo com a morte do mesmo
- Acompanhe a população mundial para ver se o planeta prospera ou perece.

Para isso você deve implementar a classe Mundo, que controla o fluxo acima e a classe indivíduo. A classe indivíduo deve ter um método que decide se o mesmo vai cooperar ou trapacear. Além de conter as moedas do mesmo. Os indivíduos não podem saber quem vão interagir, mas podem saber o resultado depois.

Como subclasses da classe indivíduo você deve ter algumas implementações:

- Colaborativo: Sempre Coopera
- Trapaceiro: Sempre Trapaceia
- Rabugento: Sempre Coopera, até ser trapaceado, a partir dai sempre trapaceará
- Copiador: Copia o comportamento do adversário no último round
- Tolerante: Coopera quase sempre, mas se for enganado 3 vezes trapaceará nas próximas 3 oportunidades, depois reiniciará ao estado inicial
- Invente qualquer pessoa tipo de indivíduo que você queira testar

Monte populações mistas dos tipos que você criou e teste buscando ver se as populações vão ser prosperas, estáveis ou vão acabar por se autodestruir.
Alguns testes interessantes:

- 499 Rabugentos e 1 Trapaceiro
- 375 Colaborativos e 125 Trapaceiros
- 437 Copiadores e 63 Trapaceiros
- 500 Matemáticos (vide Desafio Opcional)

Desafio Opcional: Implemente o Matemático que Coopera com uma probabilidade p e Trapaceia com uma probabilidade (1 - p)(100% - probabilidade de cooperar). Encontre o p que torne o Matemático mais eficiente possível.

3 2 - C# Intermediário

3.1 Aulas

3.2 Duração Estimada

- 28 horas de conteúdo.
- 12 horas de desafios.
- 4 horas de revisão (opcional).
- 4 horas de avaliação (opcional).
- Total: 40 a 48 horas.

3.3 Overview

Neste curso de C# Intermediário você aprenderá recursos mais avançados do .NET e como construir aplicações modernas utilizando este framework.

3.4 Competências

As Competências a serem desenvolvidas ao longo do curso são:

1. **...**

3.5 Aula 11 - Design Avançado de Objetos

3.5.1 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

```
1  <Project Sdk="Microsoft.NET.Sdk">
2
```

```
  <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 { }
}
```

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.

```
using Namespace1;

Classe1 obj1 = new Classe1();
Classe2 obj2 = new Classe2();
```

```
using Namespace1.Namespace2;

Classe1 obj1 = new Classe1(); // OK
Classe2 obj2 = new Classe2(); // Error
Classe3 obj3 = new Classe3(); // OK
```

```
using Namespace1.Namespace2;
using Namespace1.Namespace3.Namespace4;

Classe3 obj3 = new Classe3();
Classe4 obj4 = new Classe4();
Classe5 obj5 = new Classe5();
```

```
Classe2 test = new Classe2(); // Error

namespace Namespace1
{
    public class Test
    {
        public static void Main()
        {
            Classe2 obj2 = new Classe2(); // OK, Classe2 e este código estão dentro do Namespace1
        }
    }
}
```

```
using Namespace1.Namespace3.Namespace4;

namespace Namespace1; // Todo código abaixo está no Namespace1

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

```
6
7   public class Test
8   {
9       public static void Main()
10      {
11          Classe1 obj1 = new Classe1();
12          Classe2 obj2 = new Classe2();
13          Classe3 obj3 = new Classe3();
14          Classe4 obj4 = new Classe4();
15          Classe5 obj5 = new Classe5();
16      }
17  }
```

3.5.2 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:

```
namespace System
{
    public class ClasseA
    {
        public void Test()
        {
            Console.WriteLine("Classe A");
        }
    }
}

namespace MyProduct
{
    public class ClasseB
    {
        public void Test()
        {
            // Console.WriteLine("Classe B"); Erro
            global::System.Console.WriteLine("Classe B");
        }
    }

    namespace System
    {
        public class ClasseC
        {
            public void Test()
            {
                // Console.WriteLine("Classe C"); Erro
                global::System.Console.WriteLine("Classe C");
            }
        }
    }
}
```

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.

3.5.3 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:

```
using System;

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.

```
using System;

// Sua própria classe de Console personalizada
public static class MyConsole
{
    public int? ReadLineInt()
    {
        var str = Console.ReadLine();

        // Um código novo para os aventureiros:
        // Se a conversão é bem sucedida, cria um int i e joga o valor convertido para fora da função
        // TryParse usando a palavar-reservada out. A palavra ainda retorna verdadeiro. Caso contrário
        // nenhum erro estoura e retorna falso
        if (int.TryParse(str, out int i))
            return i;

        return null;
    }

    public void Print(object obj)
    {
        // Usando a conversão para string que todo objeto tem
        var str = obj.ToString();
        Console.WriteLine(str);
    }
}
```

Usando a classe MyConsole sem using estática:

```
MyConsole.Print("Digite um número");
int? a = MyConsole.ReadLineInt();

MyConsole.Print("Digite outro número");
int? b = MyConsole.ReadLineInt();

if (a is null || b is null)
    MyConsole.Print("Números inválidos.");
else MyConsole.Print(a + b);
```

Usando a classe MyConsole com using estática:

```
using static MyConsole;

Print("Digite um número");
int? a = ReadLineInt();

Print("Digite outro número");
int? b = ReadLineInt();

if (a is null || b is null)
    Print("Números inválidos.");
else Print(a + b);
```

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

```
public static class GameConfiguration
{
```

```
    public static bool AutoSave { get; set; }
    public static bool SoundOn { get; set; }
    public static string MenuButton { get; set; }

    // Construtor estático, usando uma vez, quando você usa a classe GameConfiguration pela primeira vez
    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;
    }
}
```

3.5.4 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:

```

public enum DiasDaSemana
```

```
{
    Domingo,
    Segunda,
    Terça,
    Quarta,
    Quinta,
    Sexta,
    Sábado
}
```

Sua utilização é fácil:

```
var dia = DiasDaSemana.Quarta;

Console.WriteLine("Que semana!");
if (dia == DiasDaSemana.Quarta)
    Console.WriteLine("Mas ainda é quarta-feira!");
```

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

```
using System;

Key key1 = Key.Ctrl;
Key key2 = (Key)(2); // C
Key key = key1 | key2; // União de Ctrl e C

// Código complexo abaixo, analisar com cuidado
if ((key & Key.C) > 0 && (key & Key.Ctrl) > 0)
    Console.WriteLine("Copiar");
else if ((key & Key.V) > 0 && (key & Key.Ctrl) > 0)
    Console.WriteLine("Colar");

public enum Key : byte
{
    Ctrl = 1,
    C = 2,
    V = 4
}
```

3.5.5 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
{
    WriteLine("O número aleatório foi muito grande!");
}
catch (MyException ex) // Trata apenas MyException
{
    WriteLine(ex);
}
catch (MyOtherException ex) when (ex.Info == "4") // Trata apenas MyOtherException quando Info é 4
{
    WriteLine(ex);
}
catch (Exception ex) // Trata qualquer outro erro
{
    WriteLine("Erro desconhecido!");
}
finally
{
    WriteLine(nome);
}

public class MyException : Exception
{
    public override string Message => "Deu um grande e catastrófico erro!";
}

public class MyOtherException : Exception
{
    public string Info { get; set; }
    public MyOtherException(string info)
        => this.Info = info;

    public override string Message => $"Falha por motivos de {Info}!";
}
```

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.

3.5.6 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.

3.5.7 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.

3.5.8 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

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

namespace DataBase;

using System;
using Exceptions;

public class DB<T>
    // Restrição genérica, T deve herdar de DataBaseObject e possui um construtor vazio
    where T : DataBaseObject, new()
{
    // Cada instância de DB tem um caminho base que diz onde serão salvos os arquivos
    private string basePath;

    private DB(string basePath)
        => this.basePath = basePath;

    // Monta o path do arquivos considerando o base path e a classe que queremos salvar (cada classe vai em um arquivo diferente)
    public string DBPath
    {
        get
        {
            // Pega o nome da classe genérica
            var fileName = typeof(T).Name;
            var path = this.basePath + fileName + ".csv";
            return path;
        }
    }

    // Funções privadas apenas para separar melhor as implementações
    private List<string> openFile()
    {
        List<string> lines = new List<string>();
        StreamReader reader = null;
        var path = this.DBPath;

        if (!File.Exists(path))
            File.Create(path).Close();

        try
        {
            reader = new StreamReader(path);
            // Lê linhas de um arquivo até que ele acabe e preenche uma lista com as linhas
            while (!reader.EndOfStream)
                lines.Add(reader.ReadLine());
        }
        catch
        {
            lines = null; // Falha
        }
        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
```

```
    {
        get
        {
            if (custom == null)
                throw new CustomNotDefinedException();
            return custom;
        }
    }

    public static void SetCustom(string path)
        => custom = new DB<T>(path);
}
```

DataBase/Exceptions/ConvertObjectError.cs

```
using System;

namespace DataBase.Exceptions;

public class ConvertObjectError : Exception
{
    public override string Message => "Algum elemento do banco está mal formatado e não pode ser convertido.";
}
```

DataBase/Exceptions/ CustomNotDefinedException.cs

```
using System;

namespace DataBase.Exceptions;

public class CustomNotDefinedException : Exception
{
    public override string Message => "O arquivo custom não foi definido. Use DB<T>.SetCustom para definir seu local";
}
```

DataBase/Exceptions/ DataCannotBeOpenedException.cs

```
using System;

namespace DataBase.Exceptions;

public class DataCannotBeOpenedException : Exception
{
    private string file;
    public DataCannotBeOpenedException(string file)
        => this.file = file;

    public override string Message => $"Os dados não puderam ser lidos/escritos no arquivo {file}";
}
```

Model/Aluno.cs

```
using DataBase;

namespace Model;

public class Aluno : DataBaseObject
{
    public string Nome { get; set; }
    public int Idade { get; set; }

    protected override void LoadFrom(string[] data)
    {
        this.Nome = data[0];
        this.Idade = int.Parse(data[1]);
    }

    protected override string[] SaveTo()
        => new string[]
        {
            this.Nome,
            this.Idade.ToString()
        };
}
```

Model/Professor.cs

```
using DataBase;

namespace Model;

public class Professor : DataBaseObject
{
    public string Nome { get; set; }
    public string Formacao { get; set; }

    protected override void LoadFrom(string[] data)
    {
        this.Nome = data[0];
        this.Formacao = data[1];
    }

    protected override string[] SaveTo()
        => new string[]
        {
            this.Nome,
            this.Formacao.ToString()
        };
}
```

Model/IRepository.cs

```
using System.Collections.Generic;

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
7   // outro, trocamos a forma como nossa aplicação se relaciona com dados sem
    quebrar nada, pois todos os repositórias implementarão as mesmas
8   // funcionalidades, porém com diferentes comportamentos
9   public interface IRepository<T>
10  {
11      List<T> All { get; }
12      void Add(T obj);
13  }
```

Model/AlunoFakeRepository.cs

```
1   using System.Collections.Generic;
2
3   namespace Model;
4
5   // Repositório Fake com uma lista interna. Pode ser usado para testes sem
    se comprometer em afetar os dados reais da aplicação.
6   public class AlunoFakeRepository : IRepository<Aluno>
7   {
8       List<Aluno> alunos = new List<Aluno>();
9       public AlunoFakeRepository()
10      {
11          alunos.Add(new Aluno()
12          {
13              Nome = "Pamella",
14              Idade = 22
15          });
16
17          alunos.Add(new Aluno()
18          {
19              Nome = "Xispita",
20              Idade = 18
21          });
22      }
23
24      public List<Aluno> All => alunos;
25
26      public void Add(Aluno obj)
27          => this.alunos.Add(obj);
28  }
```

Model/ProfessorFakeRepository.cs

```
1   using System.Collections.Generic;
2
3   namespace Model;
4
5   // Repositório Fake com uma lista interna. Pode ser usado para testes sem
    se comprometer em afetar os dados reais da aplicação.
6   public class ProfessorFakeRepository : IRepository<Professor>
7   {
```

```
    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

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

IRepository<Aluno> alunoRepo = null;
IRepository<Professor> profRepo = null;

// Se você estiver no modo debug (dotnet run) não acessará arquivos, apenas um repositório fake que traz dados de mentira
// para facilitar testes sem ter que apagar ou reiniciar os dados da aplicação. Sem em release (dotnet run -c release)
// você está gerando o produto final que acessa o 'banco de dados' nos arquivos. Você ainda pode criar um novo tipo de repositório
// na Model, conectando com o banco de dados SQL, por exemplo, e só alterar aqui sem ter que alterar mais nada da aplicação.
#if DEBUG

alunoRepo = new AlunoFakeRepository();
profRepo = new ProfessorFakeRepository();

#else

alunoRepo = new AlunoFileRepository();
profRepo = new ProfessorFileRepository();

#endif

while (true)
{
    try
    {
        Clear();
        WriteLine("1 - Cadastrar Professor");
        WriteLine("2 - Cadastrar Aluno");
        WriteLine("3 - Ver Professores");
        WriteLine("4 - Ver Alunos");
        WriteLine("5 - Sair");
        int opt = int.Parse(ReadLine());

        switch (opt)
        {
            case 1:
                break;

            case 2:
                Aluno aluno = new Aluno();
                aluno.Nome = ReadLine();
                aluno.Idade = int.Parse(ReadLine());
                alunoRepo.Add(aluno);
                break;

            case 3:
                var profs = profRepo.All;
                for (int i = 0; i < profs.Count; i++)
                {
```

```
                    WriteLine(profs[i].Nome);
                    WriteLine(profs[i].Formacao);
                    WriteLine();
                }
                break;

            case 4:
                var alunos = alunoRepo.All;
                for (int i = 0; i < alunos.Count; i++)
                {
                    WriteLine(alunos[i].Nome);
                    WriteLine(alunos[i].Idade);
                    WriteLine();
                }
                break;

            case 5:
                return;
        }
    }
    catch
    {
        // Um Catch voltado ao usuário final e não mais a renomear o erro
        WriteLine("Erro na aplicação, por favor consulte a TI");
    }

    WriteLine("Aperte qualquer coisa para continuar...");
    ReadKey(true);
}
```

3.5.9 Exercícios Propostos

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

3.6 Aula 12 - Coleções e Introdução a Language Integrated Query (LINQ)

- Coleções, IEnumerable e IEnumerator
- Métodos Iteradores
- Introdução ao LINQ
- Métodos de Extensão
- Inferência
- Observações Importantes
- Exercícios Propostos

3.6.1 Coleções, IEnumerable e IEnumerator

Como você pode ter percebido, existem muitas coleções no C#. Vetores, Listas encadeadas, arrays dinâmicos, pilhas, filas, entre outras possibilidades. Existe uma certa dificuldade de padronizar a forma como acessamos coleções. Pensando nisso, o C# trouxe um padrão de projeto chamado iterador. Este é antigo e nasceu para aumentar a velocidade com que nós varriamos listas encadeadas. Se você lembra bem, na nosso exemplo de lista encadeada nós sempre pegavamos o primeiro elemento e olhávamos o próximo até encontrar o elemento que queríamos. Fazer esse processo toda vez é lento se queremos ler todos os elementos da lista. Pensando nisso a Microsoft criou a seguinte interface:

```
public interface IEnumerator<T>
{
    T Current { get; }
    bool MoveNext();
    void Reset();
```

```
6   }
```

Vamos supor uma coleção aleatória. O V que você vê é a posição do iterador. Ele é isso, uma seta que aponta par algum lugar na coleção. Entende-se por coleção, qualquer conjunto de objetos, com ou sem ordem, com ou sem repetição. Ou seja, nem mesmo é um conjunto. Mesmo assim, podemos colocar os elementos enfileirados de qualquer forma que conseguirmos. O iterador começa em uma posição fora do vetor.

V										
-	7	9	3	4	8	0	4	6	2	1

Se verificarmos o valor de Current teremos então um erro. Ao usar a função MoveNext o iterador avança e a função retorna 'true', pois o avanço foi bem sucessido.

	V									
-	7	9	3	4	8	0	4	6	2	1

Agora Current tem o valor de 7. Chamando MoveNext 3 vezes teríamos então:

				V						
-	7	9	3	4	8	0	4	6	2	1

Se por acasos chamarmos Reset:

V										
-	7	9	3	4	8	0	4	6	2	1

Chamando MoveNext 1000 vezes o iterador fica na última posição possível, caso seja impossível avançar:

										V
-	7	9	3	4	8	0	4	6	2	1

Um iterador padrozina a forma que se lê uma coleção. Note que ele não permite que você altere os valores do iterador. Nem mesmo volte uma única casa. Apenas avançar, resetar e ler.
No C# toda coleção retorna um iterador, pois toda coleção implementa IEnumerable:

```
1   public interface IEnumerable<T>
2   {
3       IEnumerator<T> GetEnumerator();
4   }
```

Toda e qualquer coleção, vetor, pilhas, dicionário, lista de todas as formas, filas, hashs, tudo, implementa a interface IEnumerable e promete te retornar um iterador para que você possa ler ela.

```
1   using System;
2   using System.Collections;
3   using System.Collections.Generic;
4
5   List<int> lista = new List<int>();
6
7   lista.Add(1);
8   lista.Add(2);
9   lista.Add(3);
10
11  var it = lista.GetEnumerator();
```

```
12
13  while (it.MoveNext())
14      Console.WriteLine(it.Current);
15
16  public class ListIterator<T> : IEnumerator<T>
17  {
18      private int index = -1;
19      public List<T> List { get; set; }
20
21      public void Reset()
22      {
23          index = -1;
24      }
25
26      public bool MoveNext()
27      {
28          if (index >= List.Count)
29              return false;
30          index++;
31          return true;
32      }
33
34      public T Current
35      {
36          get
37          {
38              return List[index];
39          }
40      }
41
42      // IEnumerator genérico implementa o IEnumerator de object, por isso
    percisamos imlementar o Current que retorna um object
43      // Mas podemos chamar o nosso Current genérico como retorno
44      object IEnumerator.Current => Current;
45
46      // Libera recursos usados pelo iterador. Aqui não somos obrigados a
    fazer nada grandioso a não se que estejamos alocando recursos não
    gerenciados
47      public void Dispose() { }
48  }
49
50  public class List<T> : IEnumerable<T>
51  {
52      private int pos = 0;
53      private T[] vetor = new T[10];
54      public int Count => pos;
55
56      public T this[int index]
57      {
58          get => this.vetor[index];
59          set => this.vetor[index] = value;
60      }
61
62      public void Add(T value)
63      {
64          int len = vetor.Length;
65          if (pos == len)
66          {
67              T[] newVetor = new T[2 * len];
68              for (int i = 0; i < pos; i++)
```

```
                newVetor[i] = vetor[i];
            vetor = newVetor;
        }

        vetor[pos] = value;
        pos++;
    }

    // Simplismente retornamos o iterador que fizemos
    public IEnumerator<T> GetEnumerator()
    {
        ListIterator<T> it = new ListIterator<T>();
        it.List = this;
        return it;
    }

    // Mesma condição que na implementação do iterador
    IEnumerator IEnumerable.GetEnumerator()
        => GetEnumerator();
}
```

Outra vantagem de implementar um IEnumerable é que ele é a base do **foreach**. O foreach é um for automático que chama o iterador e tem um funcionamento idêntico ao while com o iterador que fizemos acima. Certamente, não é possível alterar a lista original, assim como não é possível alterar os dados no iterador, enquanto você percorre um foreach:

```
List<int> lista = new List<int>();

lista.Add(1);
lista.Add(2);
lista.Add(3);

foreach(var n in lista)
{
    Console.WriteLine(n);
}

// ...
```

3.6.2 Métodos Iteradores

Embora útil, a implementação pode ser um pouco tediosa. Pensando nisso o C# possui métodos iteradores. Qualquer função que deseja retornar um IEnumerable ou IEnumerator pode usa da palavra reservada yield para retornar os valores um por um. O código abaixo é equivalente ao código feito acima com a implementação completa do iterador:

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

List<int> lista = new List<int>();

lista.Add(1);
lista.Add(2);
lista.Add(3);

foreach(var n in lista)
    Console.WriteLine(n);

public class List<T> : IEnumerable<T>
```

```
{
    private int pos = 0;
    private T[] vetor = new T[10];
    public int Count => pos;

    public T this[int index]
    {
        get => this.vetor[index];
        set => this.vetor[index] = value;
    }

    public void Add(T value)
    {
        int len = vetor.Length;
        if (pos == len)
        {
            T[] newVetor = new T[2 * len];
            for (int i = 0; i < pos; i++)
                newVetor[i] = vetor[i];
            vetor = newVetor;
        }

        vetor[pos] = value;
        pos++;
    }

    public IEnumerator<T> GetEnumerator()
    {
        for (int i = 0; i < pos; i++)
        {
            yield return vetor[i];
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
        => GetEnumerator();
}
```

O fluxo é muito interessante: Toda vez que o iterador é chamado, por um foreach por exemplo, o código do GetEnumerator roda junto de chamadas do MoveNext mas trava na linha yield return. Ou seja, o código não executa infinitamente mas literalmente congela na linha do yield return e só descongela quando chamamos o MoveNext novamente. Ao chamar o Current, obtemos o valor do último yield return visto. Observe o exemplo abaixo para entender este complexo fluxo de informação:

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

Console.WriteLine("Vou chamar a função get");
var it = get();
Console.WriteLine("Chamei a função get");

Console.WriteLine("Vou chamar a função MoveNext");
it.MoveNext();
Console.WriteLine("Congelei");
Console.WriteLine(it.Current);

it.MoveNext();
Console.WriteLine("Congelei");
Console.WriteLine(it.Current);

```

```
17  it.MoveNext();
18  Console.WriteLine("Congelei");
19  Console.WriteLine(it.Current);
20
21  it.MoveNext();
22  Console.WriteLine("Congelei");
23  Console.WriteLine(it.Current);
24
25  it.MoveNext();
26  Console.WriteLine("Congelei");
27  Console.WriteLine(it.Current);
28
29  IEnumerator<int> get()
30  {
31      Console.WriteLine("Entrei no get");
32      yield return 1;
33      Console.WriteLine("Descongelei a primeira vez");
34      yield return 2;
35      Console.WriteLine("Descongelei a segunda vez");
36      yield return 3;
37      Console.WriteLine("Descongelei a última vez");
38  }
```

Como saída desse programa temos:

- Vou chamar a funçao get
- Chamei a funçao get
- Vou chamar a funçao MoveNext
- Entrei no get
- Congelei
- 1
- Descongelei a primeira vez
- Congelei
- 2
- Descongelei a segunda vez
- Congelei
- 3
- Descongelei a última vez
- Congelei
- 3
- Congelei
- 3

3.6.3 Introdução ao LINQ

E se você usasse o iterador para fazer funções universais de processamento de coleções? E se usassemos métodos iteradores para fazer isso por demanda, apenas fazendo cálculos quando os valores são solicitados? Essa é a ideia genial por trás de uma das mais brilhantes features do C#, a Consulta Integrada à Linguagem, Language Integrated Query ou, simplismente, LINQ. A ideia é criar uma função estática que receba uma coleção qualquer (que herde de IEnumerable) e use o iterador para processá-la. Observe:

```
1   using System;
2   using System.Collections.Generic;
3
4   List<int> list = new List<int>();
5   list.Add(1);
6   list.Add(2);
7   list.Add(3);
8
9   Stack<int> stack = new Stack<int>();
10  stack.Push(1);
```

```
stack.Push(2);
stack.Push(3);

int[] array = new int[] { 1, 2, 3 };

Console.WriteLine(Enumerable.Count(list)); // 3
Console.WriteLine(Enumerable.Count(stack)); // 3
Console.WriteLine(Enumerable.Count(array)); // 3

public static class Enumerable
{
    public static int Count(IEnumerable<int> coll)
    {
        int count = 0;

        var it = coll.GetEnumerator();
        while (it.MoveNext())
            count++;

        return count;
    }
}
```

A função Count conta quantos elementos existem em uma coleção, independete de qual seja. E é claro, melhoria 1: Count pode ser uma função genérica:

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

List<int> list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);

Stack<string> stack = new Stack<string>();
stack.Push("1");
stack.Push("2");
stack.Push("3");

IEnumerable[] array = new IEnumerable[] { list, stack };

Console.WriteLine(Enumerable.Count<int>(list)); // 3
Console.WriteLine(Enumerable.Count<string>(stack)); // 3
Console.WriteLine(Enumerable.Count<IEnumerable>(array)); // 2

public static class Enumerable
{
    public static int Count<T>(IEnumerable<T> coll)
    {
        int count = 0;

        var it = coll.GetEnumerator();
        while (it.MoveNext())
            count++;

        return count;
    }
}
```

Atenção especial ao vetor de coleções que tem list e stack nele, perceba que funciona perfeitamente.
Outro exemplo: A função Take. Recebe a coleção e um número N, pega apenas os primeiros N valores:

```
public static class Enumerable
{
    public static IEnumerable<T> Take<T>(IEnumerable<T> coll, int N)
    {
        List<T> list = new List<T>();

        var it = coll.GetEnumerator();
        for (int i = 0; i < N && it.MoveNext(); i++)
            list.Add(it.Current);

        return list;
    }
}
```

Note dois fatos peculiáremos: Primeiro, a função retorna um IEnumerable, ou seja, podemos usar yield return na função Take, essa é a nossa melhoria 2:

```
public static class Enumerable
{
    public static IEnumerable<T> Take<T>(IEnumerable<T> coll, int N)
    {
        var it = coll.GetEnumerator();
        for (int i = 0; i < N && it.MoveNext(); i++)
            yield return it.Current;
    }
}
```

Segundo, como temos uma coleção de resultado podemos usar várias funções LINQ uma depois da outra:

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

List<int> list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);

Stack<string> stack = new Stack<string>();
stack.Push("1");
stack.Push("2");
stack.Push("3");

Console.WriteLine(Enumerable.Count<int>(Enumerable.Take<int>(list,
2))); // 2
Console.WriteLine(Enumerable.Count<string>(Enumerable.Take<string>(stack,
10))); // 3

public static class Enumerable
{
    public static IEnumerable<T> Take<T>(IEnumerable<T> coll, int N)
    {
        var it = coll.GetEnumerator();
        for (int i = 0; i < N && it.MoveNext(); i++)
            yield return it.Current;
    }

```

```
26      public static int Count<T>(IEnumerable<T> coll)
27      {
28          int count = 0;
29
30          var it = coll.GetEnumerator();
31          while (it.MoveNext())
32              count++;
33
34          return count;
35      }
36  }
```

3.6.4 Métodos de Extensão

Apesar de ser uma feramente legal, está longe do ideal. Mas para entender como tudo melhora, precisamos compreender os métodos de extensão. Primeiramente observe o seguinte exemplo:

```
1   using System;
2
3   MyExtensionMethods.Print("Oi");
4
5   public static class MyExtensionMethods
6   {
7       public static void Print(string text)
8       {
9           Console.WriteLine(text);
10      }
11  }
```

Fizemos um print mais extenso do que o necessário. Não seria legal se o Print já existisse dentro da classe String? Infelizmente não podemos abrir a classe String e adicionar nós mesmos, não é? E usar um 'Console.WriteLine(this)' (print você mesmo) lá de dentro. Isso não seria porquê veríamos coisas privadas dentro da classe que não gostaríamos de ver. Mas existe uma solução: Métodos de Extensão. Observe:

```
1   using System;
2
3   // Quando isso compila...
4   "Oi".Print();
5   // Vira isso...
6   // MyExtensionMethods.Print("Oi");
7
8   // objeto estranho.PrintObj() também funciona
9   new AppDomainUnloadedException().PrintObj();
10
11  public static class MyExtensionMethods
12  {
13      // Basta adicionar 'this' antes do primeiro parâmetro no método
    estático e adicionamos o método Print dentor da string
14      public static void Print(this string text)
15      {
16          Console.WriteLine(text);
17      }
18
19      // Ou de QUALQUER objeto
20      public static void PrintObj(this object obj)
21      {
22          Console.WriteLine(obj);
23      }
24  }
```

Assim nossos métodos LINQ ganham a terceira melhoria, invertendo a ordem de chamada, deixando na ordem que nós usamos de fato:

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

List<int> list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);

Stack<string> stack = new Stack<string>();
stack.Push("1");
stack.Push("2");
stack.Push("3");

Console.WriteLine(list.Take<int>(2).Count<int>()); // 2
Console.WriteLine(stack.Take<string>(10).Count<string>()); // 3

public static class Enumerable
{
    public static IEnumerable<T> Take<T>(this IEnumerable<T> coll, int N)
    {
        var it = coll.GetEnumerator();
        for (int i = 0; i < N && it.MoveNext(); i++)
            yield return it.Current;
    }

    public static int Count<T>(this IEnumerable<T> coll)
    {
        int count = 0;

        var it = coll.GetEnumerator();
        while (it.MoveNext())
            count++;

        return count;
    }
}
```

3.6.5 Inferência

Para nossa quarta e última melhoria nesta aula nós temos a inferência: A capacidade do C# de perceber o tipo que está sendo passado pelo contexto. Assim alguns parâmetros genéricos torna-se desnecessários:

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

List<int> list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);

Stack<string> stack = new Stack<string>();
stack.Push("1");
stack.Push("2");
stack.Push("3");

Console.WriteLine(list.Take(2).Count()); // 2
Console.WriteLine(stack.Take(10).Count()); // 3
```

```
16
17  public static class Enumerable
18  {
19      public static IEnumerable<T> Take<T>(this IEnumerable<T> coll, int N)
20      {
21          var it = coll.GetEnumerator();
22          for (int i = 0; i < N && it.MoveNext(); i++)
23              yield return it.Current;
24      }
25
26      public static int Count<T>(this IEnumerable<T> coll)
27      {
28          int count = 0;
29
30          var it = coll.GetEnumerator();
31          while (it.MoveNext())
32              count++;
33
34          return count;
35      }
36  }
```

3.6.6 Observações Importantes

É importante comentar que todas as coleções geradas por métodos LINQ são imutáveis. Isso significa que ao executar a função Take, você não modifica a coleção original, mas gera uma nova coleção. Você não está de fato mudando a coleção inicial. Tenha sempre isso em mente. A cada função LINQ você gera uma nova coleção 'virtual' ou não. Evidentemente, se os objetos da coleção forem por referência, alterá-los altera seus valores nas coleções originais.

Outro fator importante é que o método é executado por demanda (a cada loop de um foreach ou quando você usa MoveNext em um iterador), ou seja, ao executar a função Take, praticamente nenhum trabalho é realizado. Só será realizado trabalho na execução de outras funções não iteradoras, como Count por exemplo, ou ao consumir os dados. Por isso, se você tem uma carga alta de trabalho, é bom lembrar que se você não requisitar os dados o trabalho não é feito.

3.6.7 Exercícios Propostos

Implemente as seguintes funções LINQ:

```
1   using System;
2   using System.Collections.Generic;
3
4   List<int> list = new List<int>();
5   list.Add(1);
6   list.Add(2);
7   list.Add(3);
8
9   Stack<string> stack = new Stack<string>();
10  stack.Push("1");
11  stack.Push("2");
12  stack.Push("3");
13
14  Console.WriteLine(list.Take(2).Count()); // 2
15  Console.WriteLine(stack.Take(10).Count()); // 3
16
17  public static class Enumerable
18  {
19      public static IEnumerable<T> Take<T>(this IEnumerable<T> coll, int N)
20      {
21          var it = coll.GetEnumerator();
22          for (int i = 0; i < N && it.MoveNext(); i++)
23              yield return it.Current;
```

```
24      }
25
26      public static int Count<T>(this IEnumerable<T> coll)
27      {
28          int count = 0;
29
30          var it = coll.GetEnumerator();
31          while (it.MoveNext())
32              count++;
33
34          return count;
35      }
36
37      // Pula os primeiros N valores e retorna o resto da coleção
38      public static IEnumerable<T> Skip<T>(this IEnumerable<T> coll, int N)
39      {
40          throw new NotImplementedException();
41      }
42
43      // Retorna a coleção com um elemento a mais no final dela
44      public static IEnumerable<T> Append<T>(this IEnumerable<T> coll, T
    value)
45      {
46          throw new NotImplementedException();
47      }
48
49      // Retorna a coleção com um elemento a mais no inicio dela
50      public static IEnumerable<T> Prepend<T>(this IEnumerable<T> coll, T
    value)
51      {
52          throw new NotImplementedException();
53      }
54
55      // Cria um array e preenche os elementos da coleção, convertendo a
    coleção para um array
56      public static T[] ToArray<T>(this IEnumerable<T> coll)
57      {
58          throw new NotImplementedException();
59      }
60
61      // Cria uma lista e preenche os elementos da coleção, convertendo a
    coleção para uma lista
62      public static List<T> ToList<T>(this IEnumerable<T> coll)
63      {
64          throw new NotImplementedException();
65      }
66
67      // Divide a coleção inicial em vários vetores de tamanho size
68      public static IEnumerable<T[]> Chunk<T>(this IEnumerable<T> coll, int
     size)
69      {
70          throw new NotImplementedException();
71      }
72
73      // Concatena duas coleções retornando uma coleção maior
74      public static IEnumerable<T> Concat<T>(this IEnumerable<T> coll,
    IEnumerable<T> second)
75      {
76          throw new NotImplementedException();
77      }
```

```
78
79      // Retorna o primeiro elemento da coleção, caso a coleção esteja vazia
    estoure um erro
80      public static T First<T>(this IEnumerable<T> coll)
81      {
82          throw new NotImplementedException();
83      }
84
85      // Retorna o primeiro elemento da coleção, caso a coleção esteja vazia
    retorne o valor padrão default(T).
86      public static T FirstOrDefault<T>(this IEnumerable<T> coll)
87      {
88          throw new NotImplementedException();
89      }
90
91      // Retorna o último elemento da coleção, caso a coleção esteja vazia
    estoure um erro
92      public static T Last<T>(this IEnumerable<T> coll)
93      {
94          throw new NotImplementedException();
95      }
96
97      // Retorna o último elemento da coleção, caso a coleção esteja vazia
    retorne o valor padrão default(T).
98      public static T LastOrDefault<T>(this IEnumerable<T> coll)
99      {
100         throw new NotImplementedException();
101     }
102
103     // Retorna o único elemento da coleção, caso ela estja vazia ou tenha
    mais de um elemento estoure um erro
104     public static T Single<T>(this IEnumerable<T> coll)
105     {
106         throw new NotImplementedException();
107     }
108
109     // Retorna o único elemento da coleção, caso ela tenha mais de um
    elemento estoure um erro, esteja vazia retorne o valor padrão default(T)
110      public static T SingleOrDefault<T>(this IEnumerable<T> coll)
111     {
112         throw new NotImplementedException();
113     }
114
115     // Retorna uma coleção com a ordem dos elementos invertida
116     public static IEnumerable<T> Reverse<T>(this IEnumerable<T> coll)
117     {
118         throw new NotImplementedException();
119     }
120
121     // Dada duas coleções, retorna uma tupla juntando cada elemento par a
    par.
122     // Exemplo: 1,2,3 com 'a','b','c' retornaria (1,'a'),(2,'b'),(3,'c')
123     // Note que está função trabalha com 2 parâmetros genéricos
124     public static IEnumerable<((T, R)> Zip<T, R>(this IEnumerable<T> coll,
    IEnumerable<T> second)
125     {
126         throw new NotImplementedException();
127     }
128 }
```

Give feedback

3.7 Aula 13 - Programação Funcional e LINQ

3.7.1 Introdução a Programação Funcional

Programação funciona é um mundo a parte dentro da programação. Ela é um paradigma de programação assim como Orientação a Objetos. Porém bem mais difícil e se acostumar. Na programação funcional a função é tratada como um objeto. Ela pode ser retornada, pode ser mandada como parâmetro e coisas desse tipo. Estruturamos o nosso pensamento em chamada de funções alto-nível que trabalham com dados imutáveis.

Vamos pegar leve olhando uma das programações funcionais mais simples e menos apegadas a uma programação funcional 'Hardcore' que é o JavaScirpt. Para você ter noção, enquanto JS tem loops e ifs, linguagens como Haskell não possuem loops nem estruturas condicionais. Implementar um merge-sort em F# usa recursão no lugar de loops:

```
let merge x y =
    let rec rMerge r x y =
        match x, y with
        | x, [] -> r @ x
        | [], y -> r @ y
        | x::xs, y::ys ->
            if x < y then rMerge (r @ [x]) xs (y::ys)
            else rMerge (r @ [y]) (x::xs) ys
    rMerge [] x y

let mergeSort x =
    let rec rMergeSort (x:'a list) =
        if x.Length < 2 then x
        else
            let split x =
                let rec rSplit (x:'a list) (y:'a list) =
                    if x.Length > y.Length
                    then rSplit x.Tail (x.Head::y)
                    else (x, y)
                rSplit x []
            let (a, b) = split x
            let sa = rMergeSort a
            let sb = rMergeSort b
            merge sa sb
    rMergeSort x

let x = [ 1; 3; 2; 5; 4; 6 ]
let r = mergeSort x
printfn "%A" r
```

A expressividade é poderosa mas complexa de se compreender. Vamos observar um pouco o JS que não é tão fortemente funcional:

```
// Declarar uma função
function f()
{
```

```
    console.log("Olá mundo")
}

// Chama uma função 3 vezes
// A função é passada como parâmetro
function chamar3Vezes(func)
{
    func()
    func()
    func()
}

// Printa 'Olá mundo' 3 vezes
chamar3Vezes(f)
```

Como você pode ver, a programação funcional permite que nós possamos usar as funções como dados, passando funções como parâmetro para outras funções. Não só isso, é possível retornar funções também:

```
function f()
{
    console.log("Olá mundo")
}

function montarFuncao(texto)
{
    return function func()
    {
        console.log(texto)
    }
}

f = montarFuncao("Olá mundo")
f()
```

A função montar função retorna uma função que pode ser usada em outros lugares. A variável 'texto' é capturada pela função func que é atribuída a variável f. Assim f é uma função e pode ser posteriormente chamada. Podemos fazer mesclagens complexas de código:

```
function montarFuncao(texto)
{
    return function func()
    {
        console.log(texto)
    }
}

function chamar3Vezes(func)
{
    func()
    func()
    func()
}

f = montarFuncao("Olá mundo")
chamar3Vezes(f)
```

3.7.2 Delegados

Para representar esse comportamento, o C# nos trás os delegados. Estruturas que são declaradas no mesmo nível que classes, structs, enums, interfaces e namespaces:

```
using System;

MeuDelegate f = func;
f();

void func()
{
    Console.WriteLine("Olá mundo");
}

public delegate void MeuDelegate();
```

O delegado define um novo tipo que podem armazenar qualquer função sem retorno (void) e sem parâmetros. Se tentarmos fazer essa operação com outro tipo de função teremos um erro. Abaixo alguns exemplos possíveis de utilização dos delegados:

```
using System;

MeuPrint print = Console.WriteLine;
print("Olá mundo");

public delegate void MeuPrint(string s);
using System;

// Delegados são apontadores de funções. Eles apontam para uma, nenhuma ou muitas funções.
// Abaixo um delegado começa não apontando para nada, após isso apontamos para count.
// Ao chamar f nós temos 'Count chamado' na tela.
MeuDelegate f = null;
f += count;
f("Olá mundo");

// Agora dicionamos parse ao ponteiro f. Ao chamar a função com a entrada "40" temos que tanto
// count quanto parse são chamados, na ordem que foram adicionados, ou seja, temos 'Count chamado'
// seguido de 'Parse chamado' e por fim a última saída, no caso do parse, é retornado e apresentado
// com o valor 40
f += parse;
int i = f("40");
Console.WriteLine(i);

int count(string s)
{
    Console.WriteLine("Count chamado");
    return s.Length;
}

int parse(string s)
{
    Console.WriteLine("Parse chamado");
    return int.Parse(s);
}

public delegate int MeuDelegate(string s);
using static System.Console;

```

```
executeNVezes(WriteLine, 10, "Xispita");

void executeNVezes(MeuDelegate func, int N, string param)
{
    for (int i = 0; i < N; i++)
        func(param);
}

public delegate void MeuDelegate(string s);
```

O próximo exemplo é muito interessante e mostra como usar delegados para representar funções matemáticas reais.

```

using static System.Console;

var f = linear(10, -5);
WriteLine(f(1));

FuncaoMatematica constante(float c)
{
    float funcaoConstante(float x)
    {
        return c;
    }
    return funcaoConstante;
}

FuncaoMatematica fx()
{
    float funcaoX(float x)
    {
        return x;
    }
    return funcaoX;
}

FuncaoMatematica soma(FuncaoMatematica f, FuncaoMatematica g)
{
    float funcaoSoma(float x)
    {
        return f(x) + g(x);
    }
    return funcaoSoma;
}

FuncaoMatematica produto(FuncaoMatematica f, FuncaoMatematica g)
{
    void funcaoProduto(float x)
    {
        return f(x) * g(x);
    }
    return funcaoProduto;
}

// a * x + b
FuncaoMatematica linear(float a, float b)
{
    var ax = produto(constante(a), fx());
    return soma(ax, constante(b));
}
```

```
48
49  public delegate float FuncaoMatematica(float x);
```

3.7.3 Métodos Anônimos

Podemos também declarar funções sem precisar declarar uma função com nome e assinatura. O nome disso são funções anônimas e existem algumas formas de fazer, por exemplo, usando a palavra reservada delegate. Mas a mais utilizada é usando a 'arrow function'. A setinha que usamos para tornar umas métodos de uma linha também pode ser usada para isso. Veja alguns exemplos:

```
1   using System;
2
3   MeuPrint print1 = delegate(string s) // OK
4   {
5       Console.WriteLine(s);
6   };
7
8   MeuPrint print2 = (string s) => // OK
9   {
10      Console.WriteLine(s);
11  };
12
13  MeuPrint print3 = s => // OK, usando inferência
14  {
15      Console.WriteLine(s);
16  };
17
18  MeuPrint print4 = s => Console.WriteLine(s); // OK, de uma única linha
19
20  public delegate void MeuPrint(string s);
```

Veja o exemplo das funções matemáticas refatorado com funções anônimas:

```
1   // Poderiamos usar o código abaixo para função constante
2   // FuncaoMatematica constante(float c)
3   // {
4   //      return x => c;
5   // }
6   // Mas como ela pode ser escrita em uma linha usamos 2 setas:
7   // A primeira para dizer qual a implementação da função constante em uma
    única linha
8   // A segunda para definir a função f(x) = c (função que independente do
    número recebido retorna uma constante)
9   FuncaoMatematica constante(float c)
10      => x => c;
11
12  FuncaoMatematica fx()
13      => x => x;
14
15  FuncaoMatematica soma(FuncaoMatematica f, FuncaoMatematica g)
16      => x => f(x) + g(x);
17
18  FuncaoMatematica produto(FuncaoMatematica f, FuncaoMatematica g)
19      => x => f(x) * g(x);
20
21  FuncaoMatematica linear(float a, float b)
22      => x => a * x + b;
23
24  public delegate float FuncaoMatematica(float x);
```

3.7.4 Exercícios Propostos

1. Faça uma função que recebe um double x e retorna uma função que recebe um valor e retorna valor^x. Use System.Math.Pow para isso.
2. Faça uma função que receba um função F, número N e um número T. F recebe um int e retorna um int. Chame a função F repetidas vezes onde o parâmetro de F é o resultado da chamada anterior de F até que o resultado de F seja T. Inicie com parâmetro N. Dica: Teste a conjectura de collatz: Se um número for par, divida por 2, se for impar multiplique por 3 e some 1. Repetindo esse processo a conjectura aponta que qualquer N deve chegar a valer T = 1 em algum momento.
3. Faça uma função que receba duas funções, uma que leva uma int em um string e outra que leva um string em um int e faça a composição das duas funções retornando a função h(x) = f(g(x)).

3.7.5 Delegados Genéricos

Não comentamos ainda, mas sim, é possível fazer delegados genéricos:

```
Func<string, int> converter = int.Parse;
Func<int, int, int> somador = soma;

int soma(int i, int j) => i + j;

public delegate R Func<T, R>(T entrada);
public delegate R Func<T1, T2, R>(T1 entrada, T2 entrada2);

No namespace System temos delegados genéricos que você pode usar a
vontade. Trata-se do Func que recebe vários parâmetros genéricos (até mais
de 10) e retorna o último parâmetro (como o exemplo acima). E Action, que
não retorna nada (void) e recebe vários parâmetros genéricos para
determinar seus parâmetros de função. Existe ainda o Predicate que sempre
retorna bool e em muitas siutações pode se substituído por Func.

using System;

Action<string> f = Console.WriteLine;
Func<double, double, double> g = Math.Pow;
Func<int, int, bool> h = (m, n) => m > n;

if (h(100, 10))
{
    var value = g(5, 2).ToString();
    f(value);
}
```

3.7.6 Select e Where

Agora que temos os poderosos recursos da Programação Funcional vamos entender como ele impacta o C# em sua principal função dentro da linguagem. Dentro das funções LINQ:

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

List<int> list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);

// Retorna o Primeiro valor par, caso contrário retorna o valor default
(0)
int value = list.FirstOrDefault(x => x % 2 == 0); // 2
```

```
11
12  public static class Enumerable
13  {
14      public static T FirstOrDefault<T>(IEnumerable<T> coll, Func<T, bool>
    func)
15      {
16          // Busca todos os objetos da coleção
17          foreach (var obj in coll)
18          {
19              // Se a condição enviada for verdadeira...
20              bool condition = func(obj);
21
22              // Retorna o Obejto
23              if (condition)
24                  return obj;
25          }
26          return default(T);
27      }
28  }
```

Este é o LINQ o mais próximo possível do seu poder máximo. Usando apenas funções anônimas podemos operar sobre dados de forma dinâmica e poderosa. Abaixo uma função de filtro chamada Where:

```
1   using System;
2   using System.Collections.Generic;
3
4   List<int> list = new List<int>();
5   list.Add(1);
6   list.Add(2);
7   list.Add(3);
8
9   // Retorna todos os valores impares da coleção
10  var values = list.Where(x => x % 2 == 1);
11
12  public static class Enumerable
13  {
14      public static IEnumerable<T> Where<T>(IEnumerable<T> coll, Func<T,
    bool> func)
15      {
16          // Busca todos os objetos da coleção
17          foreach (var obj in coll)
18          {
19              // Se a condição enviada for verdadeira...
20              bool condition = func(obj);
21
22              // Retorna o Obejto
23              if (condition)
24                  yield return obj;
25          }
26      }
27  }
```

O Where é usado para filtrar os dados, ou seja, ler apenas os dados que atendem uma determinada condição.
Mesmo não podendo alterar a lista original podemos criar novos dados a partir de uma lista anterior de forma imutável. Para isso utilizamos o Select.

```
1   using System;
2   using System.Collections.Generic;
```

```
3
4   List<int> list = new List<int>();
5   list.Add(1);
6   list.Add(2);
7   list.Add(3);
8
9   // De uma coleção de inteiros [1, 2, 3] retornamos uma lista de string dos
    quadrados
10  // ou seja, ["1", "4", "9"].
11  var values = list.Select(x => (x * x).ToString());
12
13  public static class Enumerable
14  {
15      public static IEnumerable<R> Select<T, R>(IEnumerable<T> coll, Func<T,
    R> func)
16      {
17          // Transforma os objetos do tipo T no tipo R
18          foreach (var obj in coll)
19              yield return func(obj);
20      }
21  }
```

O select por sua vez chama a a função anônima várias vezes e altera os objetos um por um. É um código incrível. Com poucas palavras em questão de sintaxe expressamos funções extremamente poderosas.

3.7.7 System.Linq

Evidentemente, tudo isso que você está utilizando já existe e está pronto. Basta importar System.Linq e você terá uma sequência infindável de funções e sobrecargas para utilizar. Você pode consultar a diversidade de implementações na páginas: https://learn.microsoft.com/pt-br/dotnet/api/system.linq.enumerable?view=net-7.0.

3.7.8 Exercícios Propostos

1. Considere o seguinte código de base para o exercício:

```
1   using System;
2   using System.Collections.Generic;
3
4   IEnumerable<int> get()
5   {
6       for (int i = 0; i < 1000; i++)
7           yield return i + 1;
8   }
```

Filtre os dados da função get para obter apenas os números múltiplos de 4.

1. Ainda considerando o exemplo do exercícios anterior, aplique a transformação da função collatz 3 vezes, ou seja, se o número for par, divida por 2, se for impar, multiplique por três e some 1. Depois disso, Conte (usando o Count da aula passada) todos os valores maiores que 1000. Repita o processo, só que dessa vez conte quantos valores são menores que 5.
2. Faça uma classe Pessoa com nome e idade e crie uma lista com várias pessoas. Apresente o nome de todos os maiores de idade.
3. Implemente o SkipWhile e TakeWhile. Funções como Skip e Take, mas que recebem uma função como o Where e pulam/pegam enquanto a condição for verdadeira.
4. Implemente a função Zip que pega duas coleções e opera elementos par-a-par em uma função dada.

```
1   using System;
2   using System.Collections.Generic;
3
4   int[] arrA = new int[] { 1, 3, 5, 7 };
5   int[] arrB = new int[] { 2, 3, 5, 9};
```

```
6
7   foreach (var k in arrA.Zip(arrB, (i, j) => j - i))
8       Console.Write(k);
9   // Result: 1002
10
11  public static class Enumerable
12  {
13      public static IEnumerable<T> TakeWhile<T>(IEnumerable<T> coll, Func<T,
    bool> func)
14      {
15          throw new NotImplementedException();
16      }
17
18      public static IEnumerable<T> SkipWhile<T>(IEnumerable<T> coll, Func<T,
    bool> func)
19      {
20          throw new NotImplementedException();
21      }
22
23      public static IEnumerable<R> Zip<T, U, R>(IEnumerable<T> coll,
    IEnumerable<U> second, Func<T, U, R> func)
24      {
25          throw new NotImplementedException();
26      }
27  }
```

Give feedback

3.8 Aula 14 - LINQ Avançado

- Agrupamento
- Objetos Anônimos
- Ordenamento
- LINQ como queries

3.8.1 Agrupamento

O Linq permite que nós agrupemos valores de uma coleção e tratemos cada coleção de forma diferente, Para isso usaremos o GroupBy, o exemplo a seguir mostra como essa operação funciona. Ela pode ser confusa para os iniciantes com LINQ em geral.

```
1   using System;
2   using System.Linq;
3   using System.Collections.Generic;
4
5   List<Pessoa> list = new List<Pessoa>();
6   list.Add(new Pessoa()
7   {
8       Nome = "Gilmar",
9       AnoNascimento = 1970,
10      MesNascimento = 4
11  });
12  list.Add(new Pessoa()
13  {
14      Nome = "Xispita",
15      AnoNascimento = 1999,
16      MesNascimento = 2
17  });
18  list.Add(new Pessoa()
19  {
20      Nome = "Trevisan",
21      AnoNascimento = 1999,
```

```
    MesNascimento = 4
});
list.Add(new Pessoa()
{
    Nome = "Pamella",
    AnoNascimento = 2000,
    MesNascimento = 6
});
list.Add(new Pessoa()
{
    Nome = "Bernardo",
    AnoNascimento = 2000,
    MesNascimento = 1
});

// Agrupa por Ano de Nascimento
var query = list.GroupBy(p => p.AnoNascimento);

// Cada vez que o foreach roda ele pega um grupo diferente
foreach (var group in query)
{
    // g.Key pega a chave que foi usada para separar os grupos, no caso o ano de nascimento
    Console.WriteLine($"As seguintes pessoas nasceram no ano de {group.Key}:");
    foreach (var pessoa in group)
    {
        Console.WriteLine(pessoa.Nome);
    }
    Console.WriteLine();
}
// Resultado:
// As seguintes pessoas nasceram no ano de 1970:
// Gilmar
//
// As seguintes pessoas nasceram no ano de 1999:
// Xispita
// Trevisan
//
// As seguintes pessoas nasceram no ano de 2000:
// Pamella
// Bernardo
//

public class Pessoa
{
    public string Nome { get; set; }
    public int AnoNascimento { get; set; }
    public int MesNascimento { get; set; }
}
```

Note que 'group' é uma coleção de elementos que estão no mesmo grupo. Considerando o código acima, poderíamos escrever o seguinte:

```

list.GroupBy(p => p.AnoNascimento)
    .Select(g => "As seguintes pessoas nasceram no ano de {g.Key}:\n" + string.Concat(g.Select(p => p.Nome + "\n")));

foreach (var textin query)
```

```
5       Console.WriteLine(text);
```

Para cada grupo que obtivemos processamos usando um Select. O Select retorna o texto 'As seguintes pessoas nasceram no ano de...' seguido de '\n' (quebra de linha) somado a um string.Concat. O string.Concat junta uma coleção de valores em uma única string. Para determinar essas strings, simplesmente usamos um Select dentro de um Select que para cada pessoa dentro do grupo seleciona seu nome seguida de uma quebra de linha. Analisando vemos que o resultado do código acima é igual ao resultado do primeiro.

3.8.2 Objetos Anônimos

Um objeto anônimo é um objeto que você pode criar sem uma classe e usá-lo por inferência dos seus campos. Basta usar a palavra reservada 'new' seguida de uma espécie de JSON que é uma estrutura nome valor. Note que tudo funciona por inferência, não é necessário dizer os tipos dos objetos.

```
1   using static System.Console;
2
3   var obj = new { nome = "Pamella", AnoNascimento = 2000 };
4
5   WriteLine(obj.nome);
6   WriteLine(obj.AnoNascimento);
```

O código acima funciona e temos um objeto de uma classe que nesse contexto não existe. Ele não pode ser exatamente retornado na maiorias das vezes pois não conseguiremos utilizar em outros contextos visto que não conhecemos o tipo do objeto contido de obj. Nem mesmo podemos mandá-los em parâmetros. Mas podemos usar isso em consultas LINQ que ficam contidas em um único escopo:

```
1   using System;
2   using System.Linq;
3   using System.Collections.Generic;
4
5   string texto = "Tles platos de tligo pala Tles Tligles Tlistes";
6
7   var query = texto
8       .GroupBy(c => c) // Agrupa as letra do texto por elas mesmas, ou seja,
    agrupa caracteres
9       .Where(g => g.Key != ' ') // Ignora o grupo de espaço
10      .Select(g => new { // Seleciona um objeto anônimo composto da letra
    maiúscula (ToUpper) com a quantidade de vezes que ela aparece (tamanho do
    grupo)
11          letra = g.Key.ToString().ToUpper(),
12          quantidade = g.Count()
13      })
14      .Select(x => $"A letra {x.letra} foi escrita {x.quantidade} vezes"); /
    / Formata em texto para apresentação
15
16  foreach (var s in query)
17      Console.WriteLine(s);
```

3.8.3 Ordenamento

Por fim, é possível, também, ordenar:

```
1   using System;
2   using System.Linq;
3   using System.Collections.Generic;
4
5   string texto = "Tles platos de tligo pala Tles Tligles Tlistes";
6
7   var query = texto
8       .GroupBy(c => c)
9       .Where(g => g.Key != ' ')
```

```
    .Select(g => new {
        letra = g.Key.ToString().ToUpper(),
        quantidade = g.Count()
    })
    .OrderBy(x => x.letra) // Ordena pela letra, ou seja, deixa em ordem alfabética
    .Select(x => $"A letra {x.letra} foi escrita {x.quantidade} vezes");

foreach (var s in query)
    Console.WriteLine(s);
```

3.8.4 LINQ como queries

Uma coisa muito bonitas no LINQ é que podemos transformar o código LINQ em consultas SQL. Ou seja, usar consultas de banco de dados (levemente diferentes) no C# é válido e o mesmo é convertido para as funções que estamos acostumados. O código abaixo gera um código idêntico ao código acima e que tem o mesmo comportamento. Só que usando uma query alto nível de fácil compreensão para humanos:

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

string texto = "Tles platos de tligo pala Tles Tligles Tlistes";

var query =
    from c in texto
    group c by c into g
    where g.Key != ' '
    select new {
        letra = g.Key.ToString().ToUpper(),
        quantidade = g.Count()
    } into x
    orderby x.letra
    select $"A letra {x.letra} foi escrita {x.quantidade} vezes";

foreach (var s in query)
    Console.WriteLine(s);
```

Ela se traduz da seguinte forma: Para cada c na variável texto use a função GroupBy, agrupando a variável c por c (ela mesma) e criando um grupo chamado g (usando into). Depois um where que é equivalente a nossa função Where. O mesmo podemos dizer do select que cria os objetos anônimos e usa novamente a palavra reservada into para dizer que esses objetos podem ser acessados usando a letra x. Ordenamos por x.letra usando o 'orderby' e por fim mais um select em x.
É importante notar que muitas vezes esse código é mais legível mas é menos poderoso pois não conseguimos usar algumas funções como Zip por exemplo. Mas sempre podemos usar ambas as notações juntas. Veja mais alguns exemplos de conversões de notação:

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

int[] arr = new int[]
{ 1, 3, 5, 7, 9 };

var query1a = from n in arr select n * n;

var query1b = arr.Select(n => n * n);


var query2a =
```

```
    from n in arr
    where n > 4
    select n * n into x
    where x < 50
    select x * x;

var query2b = arr
    .Where(n => n > 4)
    .Select(n => n * n)
    .Where(x => x < 50)
    .Select(x => x * x);
```

3.9 Aula 15 - Desafio 5

Busque pelo SRAG 2021 (Síndrome Respiratória Aguda Grave). Esses dados trazem todos os casos de entrada no hospital com uma doença respiratória. Olhe o dicionário de dados para identificar cada coluna e poder ler os dados. Filtre os casos de Covid-19 e responda a seguinte pergunta: Onde a mortalidade é maior, nos vacinados ou nos não vacinados? Use consultas LINQ para responder essa pergunta. Dicas:

1. Não acesse os dados de um servidor remoto, caso contrário a aplicação poderá ficar muito lenta. Tenha o .csv dos dados no Desktop.
2. Use um método iterador para ler os dados linha a linha.
3. Ao terminar tudo busque pelo fenômeno de Simpson.

Você pode encontrar os dados em: https://opendatasus.saude.gov.br/dataset/srag-2021-a-2024

3.10 Aula 16 - Pattern Matching e Sobrescrita de Operadores

Em breve...

3.11 Aula 17 - Orientação a Eventos

Em breve...

3.12 Aula 18 - Desafio 6

Em breve...

3.13 Aula 19 - Programação Paralela e Assíncrona

- Sistemas Operacionais, Escalonamento de Tarefas e Preempção
- Deadlocks, Starvation, Priority Inversion e Algoritmo do Avestruz
- Desenvolvimento Paralelo, Threads e Lei de Amdahl
- Thread-Safe, SpinLocks, Mutex, Semaphore Monitor e Lock
- Exemplo: Computando uma gaussiana e avaliando sua qualidade
- Exemplo: Soma de 100 milhões de números
- Async e Await

3.13.1 Sistemas Operacionais, Escalonamento de Tarefas e Preempção

Ao compreender o funcionamento de um processador contemplamos que é impossível um único processador executar duas tarefas ao mesmo tempo. Dito isso, é impossível que em um computador controlemos a interação do cursor e a execução de uma aplicativo qualquer. O que acontece na verdade é que existem sistemas que gerenciam a execução de multiplas aplicações alternando entre elas, esses sistemas são os Sistemas Operacionais. Neles, cada aplicação torna-se uma tarefa e eles trocados rapidamente no processador de forma que se torna impossível que o usuário perceba que na verdade uma tarefa está executando por pouquissímo tempo e então ficando parado no resto do tempo. É imperceptível que uma aplicação fica congelada por algum tempo já que o mesmo trata-se de poucos microssegundos. Lembrando: Em 1 microssegundo, um computador é capaz de executar cerca de poucas milhares de instruções e fazer com que uma tarefa realmente avance em seu processamento. Ao mesmo tempo, 50 microssegundos é tão pouco tempo que em um segundo teríamos 20 mil tarefas colocadas ao processador. Isso significa que mesmo que você possua 1000 tarefas no seu computador, uma aplicação que você está usando entra no processador e executa poucas millhões de instruções.

Para trocar as terefas do processador por outro realizamos uma troca de contexto: Salvamos todas os registradores na memória e compiamos os registradores que estavam salvos da tarefa que irá entrar no processador. Em geral trabalhamos com interrupções: Recursos do processador que "pausam" o processador obrigando a execução de uma determinada secção de código. Note que não tem como o Sistema Operacional agir enquanto outra aplicação está rodando. Por isso, é necessário que o hardware pare uma tarefa e guarde qual é a linha onde o código está para futuro retorno. O nome dessa operação é chamada de Preempção.
Após a preempção o processador é entregue ao Sistema Operacional (que assim não é nada mais nada menos que uma tarefa) que irá realizar o Escalonamento de Tarefas: Escolher uma terefa para entrar no processador. O escalonamento pode ser de várias formas: Escolher qualquer uma; Considerar quanto tempo faz que uma tarefa não é executada; Considerar um nível de prioridade para que uma tarefa ganhe o processador; E assim por diante.
Outra operação que resulta em uma Preempção é o acesso a recursos. Se duas tarefas estão trabalhando com o mesmo recurso na memória, existem alguns cuidados que se necessita ter com a aplicação para que ela não perca dados e tenha comportamente inesperado. Assim quando uma tarefa acessa um recuros já usado ou que demora para ser acessado a tarefa é posta para dormir e aguardar, deixando assim, o processador.

3.13.2 Deadlocks, Starvation, Priority Inversion e Algoritmo do Avestruz

Alguns problemas acontecem quando estamos em sistemas multi-tarefas:

- Starvation: A Inanição acontece quando o Sistema Operacional não percebe que uma tarefa deve executar por causa do algoritmo e de como o algoritmo funciona. Por exemplo existem tarefas com prioridades muito altas que nunca deixam uma tarefa executar e ela acaba nunca executando.
- Priority Inversion: A inversão de prioridade acontece quando uma tarefa de menor prioridade toma o processador e não deixa uma tarefa de maior prioridade executar. Isso acontece pois uma tarefa mais importante pode estar esperando um recurso que a tarefa de baixa prioridade está usando.
- Deadlocks: Deadlocks talvez sejam os problemas mais danosos dessa lista. Um impasse acontece de várias formas, mas a mais básica é quando uma tarefa A e outra B precisam ambas usar recursos R e S. Logo após a tarefa A tomar posse do recurso R ela sofre preempção pelo sistema operacional que agora da o controle para a tarefa B. A tarefa B toma posse do recurso S e então tenta tomar posse do recurso R, mas percebe que já está em uso por outra tarefa e então é preemptada pelo SO que agora devolve o processador para a tarefa A. A tarefa A ao retornar percorre o código e agora chega o momento de torma posse do recurso S, que já está em uso, logo A é preemptada também. Assim A e B estão como inativas: A espera o recurso S que B está usando e B espera o recurso R que A está usando. Logo elas nunca serão executadas e pior: Os recursos R e S ficarão retidos para sempre e qualquer outra tarefa que precise deles também cairá num impasse que não será capaz de executar.

Muitos desses problemas podem ser resolvidos com o Algoritmo do Avestruz mostrado abaixo:

```
1  public void OstrichAlgorithm()
2  {
3
4  }
```

Comicamente, o Algoritmo do Avestruz é basicamente ignorar o problema. Deadlocks são raros e alguns outros ocorrências dos problemas acimas podem ser raros ou até não impactarem o sistema severamente. Então não fazer nada, simplismente pode ser uma boa opção. Outras soluções podem gastar processamento desnecessário em 99.9% dos cenários para resolver um problema que ocorre em 0.1% destes. Mesmo assim são problemas que devemos conhecer porquê nem sempre o algoritmo do avestruz é uma opção. Ao desenvolver aplicações que trabalham com muitas tarefas é importante conhecer os possíveis problemas que a mesma resultará.

3.13.3 Desenvolvimento Paralelo, Threads e Lei de Amdahl

Podemos aumentar o desempenho de um uma aplicação com desenvolvimento paralelo, ou seja, a criação de várias tarefas para a mesma aplicação o que chamamos de Threads. Uma Thread é um código executado em paralelo a aplicação inicial. Note que em um sistema mono-core isso apenas divide o trabalho em vários componentes de código. Porém, em sistemas muilt-core, cada core pode executar uma parte do trabalho em um processador diferente. Deadlocks pode ocorrer aqui e é necessário saber trabalhar com eles, contudo, ainda sim podemos conseguir um grande speedup (ganho de velocidade) separando o código em várias Threads.

É importante perceber que existem processos não paralelizáveis que precisam ser executados sequêncialmente, caso contrário, não seria possível obter o resultado correto. Por exemplo a sequência de Fibonacci, que precisa dos elementos anteriores para que o mesmo funcione. Assim separar o processo em 2 seria complicado para não dizer impossível.

A Lei de Amdahl diz que o aumento do desempenho de um sistema dado a melhora e uma parte do sistema é proporcional ao impacto da parte do sistema no todo. Na programação paralela ela serve para indicar o aumento máximo teórico de uma aplicação cuja parte paralelizável equivale a uma proporção **p** e a quantidade cores é **k** onde o trabalho pode ser dividido.

O ganho de velocidade **S** é escrito como o trabalho inicial **T** sobre o trabalho com as melhorias aplicadas:

$$S = \frac{T}{T'}$$

Com o paralelismo a parte do código paralelisada é dividida entre os processadores:

$$T' = T(\frac{p}{k} + 1 - p)$$

Assim o ganho teórico de velocidade é dado por:

$$S = \frac{T}{T'} = \frac{T}{T(\frac{p}{k}+1-p)} = \frac{1}{\frac{p}{k}+1-p} = \frac{k}{p+k-pk} = \frac{k}{p+k(1-p)}$$

Ou seja, caso uma aplicação possua 50% de código (com 50% de código queremos dizer 50% de trabalho não quantidade de linhas) paralelizável em um sistema com 4 cores teremos o seguinte ganho teórico:

$$S = \frac{4}{0.5+4*0.5} = \frac{4}{2.5} = 1.6$$

Ou seja, o código deve se tornar 60% mais rápido. Se antes ele demorava 16 segundos agora ele deve demorar apenas 10.

3.13.4 Thread-Safe, SpinLocks, Mutex, Semaphore Monitor e Lock

Podemos ganhar muito com Threads, mas precisamos aprender a fazer com que tudo funcione, e para isso temos que garantir que o processo seja Thread-Safe. Isso significa que nenhum dado é perdido e nenhuma condição de corrida (onde você não sabe qual código irá acessar um recurso primeiro) irá ocorrer. Para isso, no exemplo a seguir usaremos muitos recursos de controle de dados e criação de Threads em C# para optimizar a computação de um trabalho.

3.13.5 Exemplo: Computando uma gaussiana e avaliando sua qualidade

Realizaremos o seguinte experimento para compreender a disputa e resolução dessa disputa por recursos. Para isso realizaremos a seguinte computação: Iremos computar uma distribuição Gaussiana que pode ser calculada com a soma de muitos valores aleatórios entre 0 e 1. Depois usaremos conceitos de estatística para apontar se a distribuição tem as qualidades de uma distrubuição gaussiana. Repetiremos o processo várias vezes para tirar uma média do resultado:

```
1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4  using System.Collections.Concurrent;
5
6  int N =
   1000;                                  // Tamanho da amostras
7  int M =
   5000;                                  // Amostras a serem utilizadas
8  int K =
   50;                                    // Vezes que o Monte Carlo será
   executado
```

```
 9
10  var seed = DateTime.Now.Millisecond;
11  var rand = new Random(seed);                    // Objeto para gerar
    números aleatórios
12  float sum = 0f;
13  float sqSum = 0f;
14  float[] dist = new float[M];
15
16  float aval = 0;
17  double time = 0;
18  for (int i = 0; i < K; i++)
19  {
20      for (int j = 0; j < dist.Length; j++)
21          dist[j] = 0;
22      sum = 0f;
23      sqSum = 0f;
24
25      var dt = DateTime.Now;
26
    run();                                          // Código entra aqui
27      var span = DateTime.Now - dt;
28      time += span.TotalMilliseconds;
29      aval += avaliate();
30  }
31
32  Console.WriteLine($"{100 * aval / K}% of precision in {time / K} ms per
    computation.");
33
34  void compute(int index)
35  {
36      var value = 0f;
37      for (int i = 0; i < N; i++)
38          value += rand.NextSingle();             // Geramos muitos valores
    aleatórios e somamos para tirar a média
39      value /= N;
40      dist[index] = value;
41      sum += value;
42      sqSum += value * value;
43  }
44
45  float avaliate()
46  {
47      float avg = sum / M;
48
49      float std = sqSum - sum * sum / M;
50      std /= M;
51      std = MathF.Sqrt(std);
52
53      int count1 = 0;
54      int count2 = 0;
55      int count3 = 0;
56      foreach (var y in dist)
57      {
58          if (y > avg + 2 * std)
59          {
60              count1++;
61              count2++;
62              count3++;
63          }
64          else if (y > avg + std)
```

```
65          {
66              count1++;
67              count2++;
68          }
69          else if (y > avg)
70          {
71              count1++;
72          }
73      }
74
75      var exp1 = M * 0.5f;
76      var exp2 = M * 0.159f;
77      var exp3 = M * 0.022f;
78
79      var err1 = MathF.Abs(count1 - exp1) / exp1;
80      var err2 =  MathF.Abs(count2 - exp2) / exp2;
81      var err3 = MathF.Abs(count3 - exp3) / exp3;
82
83      var prec = 1f - (err1 + err2 + err3) / 3;
84      return prec < 0 ? 0 : prec;
85  }
86
87  void run()
88  {
89      for (int i = 0; i < M; i++)
90          compute(i);
91  }
```

A função *run* executa *compute* M vezes até que a computação esteja concluida. O resultado é >95% de precisão e ~80ms por processamento. Note que estamos em um quad-core onde, teoricamente, cada soma pode ser realizada em paralelo, logo p=1. Assim esperamos aumentar a velocidade da aplicação em 4 vezes. Agora vamos tentar melhorar a execução aprendendo a utilizar Threads:

```
1   void threadWrongRun()
2   {
3       for (int i = 0; i < M; i++)
4       {
5           Thread thread = new Thread(() =>
6           {
7               compute(i);
8           });
9           thread.Start();
10      }
11  }
```

Pelo nome da função você sabe que algo está errado, mas o quê? Ao executar temos o seguinte erro: '*Index was outside the bounds of the array.*' Porquê disso acontecer é simples: Ao criarmos a variável *i* e passarmos ela dentro da função anonima na Thread estamos fazendo com que essa função capture a variável. Isso significa que quando a thread rodar ela usará o valor atual de *i* no momento da execução. O problema é que algumas threads só executarão de fato após o for ser completo. Quando isso acontece *i* já vale *N* (o que ocasionou na quebra do for) e então todas as threads usam o valor atual de *i* que é *N* resultando num estouro de vetor dentro da função *compute*.
Para resolver isso precisamos fazer uma cópia imutável de *i*:

```
1   void threadRun()
2   {
3       for (int i = 0; i < M; i++)
4       {
5           int j = i;
6           Thread thread = new Thread(() =>
```

```
        {
            compute(j);
        });
        thread.Start();
    }
}
```

Agora o código funciona pois cada Thread possui uma cópia da variável *i*. Note que *j* é definido a cada loop, assim existem vários *j's* na memória e não só 1 como poderia se imaginar. Cada um com um valor fixo.
Mas de repente surge um novo problema: Obtivémos <70% e ~680 ms para a execução desta solução. Perdemos tanto precisão quanto desempenho. O motivo é porque a criação e gerenciamento de threads é pesada, criamos *M* threads que no caso são 5000 threads sendo que só temos 4 núcleos. A queda na precisão é interessante de se avaliar também. A classe Random não é Thread-Safe, isso significa que podemos literalmente quebrar a classe se usarmos ela várias vezes ao mesmo tempo. Vamos resolver um problema de cada vez. Primeiramente o desempenho. Para isso vamos utilizar a classe Parallel que usa um *ThreadPool*, ou seja, reuitliza as threads e considera a quantidade de cores na hora de executá-las:

```
void wrongParallelForRun()
{
    Parallel.For(0, M, i =>
    {
        compute(i);
    });
}
```

Um dado importante: Ao executar a função *start* do objeto thread nós iniciamos a thread em segundo plano e continuamos a executar o programa. O Parallel para o programa na linha da chamada de qualquer função e só continua a executar quando o trabalho em paralelo for completamente executado.
Bom, resolvemos noss problema de desemepnho, embora ainda não esteja bom, alcançando ~180 ms de execução. Mas continua errado, visoto que nossa precisão é <1%. Agora entra o trabalho de tentar tornar nossa aplicação Thread-Safe:

```
void spinLockWrongParallelForRun()
{
    SpinLock sl = new SpinLock();
    bool blocked = false;

    Parallel.For(0, M, i =>
    {
        try
        {
            sl.Enter(ref blocked);
            compute(i);
        }
        finally
        {
            if (blocked)
                sl.Exit();
        }
    });
}
```

Novamente, o '*Wrong*' no nome já mostra que algo não está certo. O mesmo erro da primeira tentativa: '*The tookLock argument must be set to false before calling this method.*'. A mensagem é diferente, o erro é o mesmo. Ao deixar a variável *blocked* para fora do Parallel For estamos comprometendo o funcionamento da struct SpinkLock. Vamos corrigir isso antes de falarmos da técnica empregada:

```

void spinLockParallelForRun()
{
    SpinLock sl = new SpinLock();
```

```
4
5       Parallel.For(0, M, i =>
6       {
7           bool blocked = false;
8           try
9           {
10              sl.Enter(ref blocked);
11              compute(i);
12          }
13          finally
14          {
15              if (blocked)
16                  sl.Exit();
17          }
18      });
19  }
```

O SpinLock é uma implementação simples que bloqueia um Thread para permitir que apenas uma thread entre dentro da zona crítica onde usamos o *compute*. Como todas as threads enxergam o mesmo objeto SpinLock, ao chamar a função *Enter*, se alguma thread já chamou essa função e ainda não chamou a função Exit, então isso significa que os recursos estão sendo utilizados e a thread entra em um loop até que outra thread passe na função Exit. Com elas melhoramos em todas as perspectivas tendo >95% de precisão e ~120 ms de execução. Ainda não estamos próximos da precisão do código sequêncial, mas corrigimos a questão da precisão. Vejamos outras opções ainda mais leves que o SpinLock:

```
1   void mutexParallelForRun()
2   {
3       using Mutex mt = new Mutex();
4
5       Parallel.For(0, M, i =>
6       {
7           mt.WaitOne();
8           compute(i);
9           mt.ReleaseMutex();
10      });
11  }
```

O Mutex é como o SpinLock. O que muda é o que acontece com a thread bloqueada, Equanto o SpinLock coloca a thread em um loop a Mutex coloca a tarefa para domir no Sistema Operacional. Isso sigfinica que ela não executará nem entrerá no processador até que seja acordada pela Mutex quando a secção crítica for liberada. Isso evita que ela tome o processador para ficar presa em um loop, dando o processador para threads que realmente precisão dele. Note que colocar uma thread para dormir custa bastante, se a operação dentro da secção crítica for pequena, vale mais a pena usar outra o SpinLock. Com a mutex mantivemos a precisão >95%, mas economizamos um pouco e agora usamos apenas ~110 ms por execução.

```
1   void semaphoreParallelForRun()
2   {
3       int count = 4;
4       using Semaphore sm = new Semaphore(count, count);
5
6       Parallel.For(0, M, i =>
7       {
8           sm.WaitOne();
9           compute(i);
10          sm.Release();
11      });
12  }
```

O semáforo também vale a pena ser comentado. Ele é como um Mutex, porém ele permite a passagem de uma quantidade de *N* threads na secção crítica. Note que um semáforo unitário é o mesmo que um Mutex. O código acima, por permitir que 4 threads adentrem a secção crítica teve o desempemho destruído. Mesmo assim é útil em algumas situações.

```
1  void monitorParallelForRun()
2  {
3      Parallel.For(0, M, i =>
4      {
5          Monitor.Enter(rand);
6          compute(i);
7          Monitor.Exit(rand);
8      });
9  }
```

O monitor é um Mutex que não pode se compartilhado enter várias aplicações não estando no nível do sistema operacional. Mas ele é bem mais leve que um Mutex ou SpinLock. Além disso, ele permite que você associe um objeto que deve ser protegido na secção crítica. No caso, todas as threads que estejam com a mesma referência de *rand*, estarão sujeitas a exclusão mútua, ou seja, serão barradas e colocadas fora de execução até que a área seja liberada. O código acima mantém a precisão e roda em ~90 ms.

```
1  void monitorWithRefParallelForRun()
2  {
3      Parallel.For(0, M, i =>
4      {
5          bool lockTaken = false;
6          try
7          {
8              Monitor.Enter(rand, ref lockTaken);
9              compute(i);
10         }
11         finally
12         {
13             if (lockTaken)
14             {
15                 Monitor.Exit(rand);
16             }
17         }
18     });
19 }
```

Acima você pode observar um Monitor usado de uma forma um pouco mais cuidadosa/profissional evitando alguns errors com um try/finally e usando um booleano para compreender se realmente a computação entrou na secção crítica ou obteve algum erro antes (e assim não deve chamar Exit). O mesmo código acima pode ser reescrito abaixo de forma simples usando um recurso do C# chamado lock:

```
1  void lockParallelForRun()
2  {
3      Parallel.For(0, M, i =>
4      {
5          lock(rand)
6          {
7              compute(i);
8          }
9      });
10 }
```

Nosso último exemplo mostra como pode ser simples escrever códigos Thread-Safe em C#. Note que o código é basicamente sequêncial e nunca vai ter desempenho melhor que o tradicional pois todo trabalho

está dentor de um lock. Além disso, é bom apontar que se deve evitar lock dentro de outro lock para evitar DeadLocks.

3.13.6 Exemplo: Soma de 100 milhões de números

Não iremos sair sem um exemplo real de ganho de performance daqui:

```
1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4
5  var seed = DateTime.Now.Millisecond;
6  var rand = new Random(seed);
7
8  DateTime dt = DateTime.Now;
9  int sum = 0;
10 byte[] arr = new byte[100_000_000];
11 rand.NextBytes(arr);
12 for (int j = 0; j < arr.Length; j++)
13     sum += arr[j];
14 var span = DateTime.Now - dt;
15 Console.WriteLine($"{sum} - {span.TotalMilliseconds}");
16
17 dt = DateTime.Now;
18 sum = 0;
19 int threadCount = 250 * Environment.ProcessorCount; // Obtém o número de
   Processadores Lógicos (no caso 250 threads por processador)
20 Parallel.For(0, threadCount, i =>
21 {
22     int _sum = 0;
23     byte[] arr = new byte[100_000_000 / threadCount];
24     lock (rand)
25     {
26         rand.NextBytes(arr);
27     }
28
29     for (int j = 0; j < arr.Length; j++)
30         _sum += arr[j];
31
32     Interlocked.Add(ref sum, _sum); // Realiza uma Soma Thread-Safe
33 });
34 span = DateTime.Now - dt;
35 Console.WriteLine($"{sum} - {span.TotalMilliseconds}");
```

Enquanto sequêncialmente levamos ~1200 ms, paralelamente levamos ~900ms. Um ganho pequeno mas real.

Ainda existem outros recursos não mostrados, como o ConcurrentQueue e outras coleções ThreadSafe no *namespace System.Collections.Concurrent*.

3.13.7 Async e Await

Outro uso para threads é a programação assíncrona, ou seja, rodar algo em segundo plano para não comprometer o funcionamento da aplicação principal. Observe o seguinte exemplo:

```
1  using System;
2  using System.Threading;
3
4  while (true)
5  {
6      var key = Console.ReadKey(true);
7      if (key.Key == ConsoleKey.Escape)
8          return;
9
```

```
    MostrarNumeroComplicado();
}

long calcularNumeroComplicado()
{
    Thread.Sleep(2000); // Fazendo calculo
    return 3141592653589793238;
}

void MostrarNumeroComplicado()
{
    var resultado = calcularNumeroComplicado();
    Console.WriteLine(resultado);
}
```

Ao clicar em qualquer botão vários caractéres entram no nosso buffer e ao entrar no *calcularNumeroComplicado* a aplicação fica congelada enquanto espera o cálculo (aqui representado pela função Sleep que faz a função dormir por 2000 ms, ou 2 segundos). Ao segurar qualquer tecla exceto o Esc por alguns segundos a aplicação fica trava mostrando o número a cada 2 segundos e não responde mais ao Esc do usuário. Em uma tela normal isso faria com que a aplicação travasse enquanto espera calcular algo, ou conectar ao banco de dados, por exemplo.

```
void MostrarNumeroComplicadoAsync()
{
    var estado = calcularNumeroComplicadoAsync();
    Thread thread = new Thread(() =>
    {
        while (!estado.EstaCompleto)
            Thread.Yield();

        Console.WriteLine(estado.resultado);
    });
    thread.Start();
}

EstadoAssincrono<long> calcularNumeroComplicadoAsync()
{
    EstadoAssincrono<long> estado = new EstadoAssincrono<long>();
    var thread = new Thread(() =>
    {
        var result = calcularNumeroComplicado();
        estado.resultado = result;
        estado.EstaCompleto = true;
    });
    thread.Start();
    return estado;
}

public class EstadoAssincrono<T>
{
    public T resultado { get; set; }
    public bool EstaCompleto { get; set; }
}
```

Neta nova solução ciramos um estádo assincrono que controla se uma dada operação já terminou e seu resultado. Quando *MostrarNumeroComplicadoAsync* é chamada ela chama *calcularNumeroComplicadoAsync* que por sua vez criará uma Thread que interage com um estado assíncrono, Quando o número for calculado, a thread irá modificar o estado. Esse estádo é imediatamente devolvido e *MostrarNumeroComplicadoAsync* verifica continuament em uma segunda thread para ver se ela já terminou. Caso não tenha terminado ela chama a função Thread.Yield que faz a preempção da Thread

atual dando espaço no processador para outra thread que tenha algo a fazer. Quando o calculo termina *EstaCompleto* é definida como verdadeiro, interrompendo o Loop na *MostrarNumeroComplicadoAsync* e fazendo com que a função void seja executada. Ao usar o programa com essa função, todo clique cria uma Thread sendo executada em paralelo mas não deixa de executar o Loop principal, assim, a qualquer momento o usuário pode usar a tecla Esc e fechar a aplicação sem que ela congele.
Felizmente, o C# já tem uma estrutura (que no fundo é bem mais complexa) para trabalhar com esses tipos de sistemas, trata-se do Async e Await. Observe:

```
1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4
5  while (true)
6  {
7      var key = Console.ReadKey(true);
8      if (key.Key == ConsoleKey.Escape)
9          return;
10
11     MostrarNumeroComplicadoAsync();
12 }
13
14 long calcularNumeroComplicado()
15 {
16     Thread.Sleep(2000);
17     return 3141592653589793238;
18 }
19
20 Task<long> calcularNumeroComplicadoAsync()
21     => Task<long>.Factory.StartNew(() => calcularNumeroComplicado());
22
23 async void MostrarNumeroComplicadoAsync()
24 {
25     var estado = await calcularNumeroComplicadoAsync();
26     Console.WriteLine(estado);
27 }
```

Task é a classe que representa uma tarefa, ou seja, uma thread que retorna resultado. Usando o Task.Factory.StartNew podemos criar novas tarefas e aguardá-las em funções assíncronas, isto é, com a palavra reservada async antes do retorno, com a palavra await. Aguardá-las faz com que o código pause naquela linha e retorne ao primeiro chamador não assíncrono. Ou seja, ao usar o await no final do código, voltamos a chamada de método de *MostrarNumeroComplicadoAsync*, e continuamos a executar de lá. Quando o calculo for completo, a execução pausa e volta para linha do await para completar o que estava sendo feito.

```
1                using System;
2                using System.Threading;
3                using System.Threading.Tasks;
4
5  1 13          while (true)
6  2 14          {
7  3 15              var key = Console.ReadKey(true);
8  4 18              if (key.Key == ConsoleKey.Escape)
9  5                     return;
10 6
11 7                 MostrarNumeroComplicadoAsync();
12 12            }
13
14               long calcularNumeroComplicado()
15               {
16 12 13 14           Thread.Sleep(2000);
```

```
17  15              return 3141592653589793238;
18              }
19
20              Task<long> calcularNumeroComplicadoAsync()
21  10              => Task<long>.Factory.StartNew(() =>
    calcularNumeroComplicado());
22
23              async void MostrarNumeroComplicadoAsync()
24  8           {
25  9 11 16         var estado = await calcularNumeroComplicadoAsync();
26  17              Console.WriteLine(estado);
27              }
28
29  1-6         Código executando normalmente
30  7           Chamada de Função
31  8           Código executando normalmente
32  9           Chamada de Função
33  10          Tarefa criada nesta linha
34  11          await chamado para Tarefa
35  12-14       Código retorna ao chamado enquanto thread executada
    paralelamente
36  15          Thread terminou
37  16          Resultado retornado
38  17          Resultado apresentado na tela
39  18          Código executando normalmente
```

Você ainda pode fazer uma sequência de funções assincronas. Quando uma função retorna Task<T> e é assícnrona, retorne T ao invés de uma Task:

```
1   using System;
2   using System.Threading.Tasks;
3
4   var result = await calcularSomaComplicadaAsync();
5   Console.WriteLine(result);
6
7   Task<long> calcularNumeroComplicado()
8       => Task<long>.Factory.StartNew(() => 910);
9
10
11  Task<long> calcularOutroNumeroComplicado()
12      => Task<long>.Factory.StartNew(() => 90);
13
14  // Criando uma Task como chamadas assíncronas de outras Tasks
15  async Task<long> calcularSomaComplicadaAsync()
16  {
17      var x = await calcularNumeroComplicado();
18      var y = await calcularOutroNumeroComplicado();
19      return x + y; // Não retorna Task<long>, mas sim long, pois é
    assíncrona e já é tratada como uma tarefa
20  }
```

Inclusive, na maioria das vezes, criamos funções assíncronas sobre outras funções assíncronas que já existem.

Outra informação importante é que caso você chame uma função assíncrona sem o await você chamará ela sincronamente. Ela retornará o controle para você assim que criada a Thread paralela. Note que se o restorno for uma Task você estará retornando uma tarefa e executando-a em paralelo.

```
1   using System;
2   using System.Threading;
3   using System.Threading.Tasks;
```

```
4
5   var result = await PegarSoma();     // Pegando o número...
6   Console.WriteLine(result);          // 1500
7
8                                       // Retornamos a tarefa e não a
    aguardamos, assim temos um objeto Tarefa e não um número, assim o retorno
    vem antes da tarefa executar
9   var result2 = PegarSoma();          //
    System.Runtime.CompilerServices.AsyncTaskMethodBuilder`1+AsyncStateMachine
    Box`1[System.Int64,Program+<<<Main>$>g__PegarSoma|0_1>d]
10  Console.WriteLine(result2);         // Pegando o número...
11
12  await MostrarSoma();                // Pegando o número...
13                                      // 1500
14
15  // await MostrarSomaVoid();         //não é possível aguardar void
16
17  MostrarSoma();                      // Essa função foi executada
    sincronamente, não aguardando neste ponto e mostrando o Pegando número da
    linha abaixo imediatamente
18  MostrarSomaVoid();                  // Pegando o número...
19                                      // Pegando o número...
20                                      // 1500
21                                      // 1500
22
23  Console.ReadKey(true);
24
25  Task<long> PegarNumero()
26      => Task<long>.Factory.StartNew(() =>
27      {
28          Thread.Sleep(500);
29          Console.WriteLine("Pegando o número...");
30          return 1000;
31      });
32
33  async Task<long> PegarSoma()
34  {
35      var x = await PegarNumero();
36      var y = 500;
37      return x + y;
38  }
39
40  async Task MostrarSoma()
41  {
42      var x = await PegarNumero();
43      var y = 500;
44      Console.WriteLine(x + y);
45  }
46
47  async void MostrarSomaVoid()
48  {
49      var x = await PegarNumero();
50      var y = 500;
51      Console.WriteLine(x + y);
52  }
```

Felizmente, você verá que Tasks é muito mais fácil de usar do dia-a-dia do que entender.

3.14 Aula 20 - Desafio 7

Monte Carlo é um método estatístico onde para obter uma probabilidade, esperança matemática ou outra informação repetimos um experimento computacionalmente (ou não) quantas vezes for necessarário. Por exemplo, para descobrir se a probabilidade de um dado obter 6 é realmente ~16.7%, basta simular um dado com a classe Random milhões devezes e ver quantas delas se obteve um 6:

```
using System;

Random rand = new Random();
int N = 1_000_000;

monteCarlo();

int roll()
    => rand.Next(6) + 1;

void monteCarlo()
{
    int count = 0;
    for (int i = 0; i < N; i++)
    {
        if (roll() == 6)
            count++;
    }
    Console.WriteLine(count / (float)N); // ~0,167
}
```

Seu desafio é criar um programa de computador que para grandes exercitos, determine o vencedor no jogo War. Relembrando as regras:

1. A guerra consiste em várias batalhas de até 3 contra 3. Os exércitos sempre mandam a maior quantidade possível, contudo, o atacante não pode mandar todos os seus soldados, tendo que ficar com ao menos 1no seu território. Asism, com menos de 4 soldados, os atacantes atacam com menos. Os defensores também mandam o máximo possível para defesa, contudo, eles estão lutando na sua casa e não precisam deixar um soladdo ao menos para trás.
2. Para cada soldado na batalha joga-se um dado e orndena-se, assim, por exemplo, os atacantes terão 3 dados e os defensores terão 3 dados também, a não ser que tenham menos que 3 soldados na batalha.
3. Comparando os dados ordenados 1 a 1, o maior da defesa contra o maior do ataque e assim por diante, determina-se um vencedor que é o que possui o maior valor no dado.
4. A defesa ganha todos os empates.
5. Para cada derrota, um time perde um soldado, podendo perder de 0 a 3 de uma vez.
6. A guerra acaba quando ou a defesa for totalmente dizimada, ou reste apenas um atacante.

Simule para grandes exércitos usando um único objeto da classe Random para toda aplicação. Simule 10 mil vezes a luta entre 1000 atacantes e 500 defensores e meça quanto tempo durou para realizar o calculo. Apresente o resultado e compare com o esperado de aproximadamente 50%.
Dica 1: Faça o código não paralelo primeiro.
Dica 2: Este exercício não precisa de Async e Await.
Dica 3: Usar ConcurrentQueue pode ajudar a resolver o problema de forma intelligente.
Dica 4: Na solução do professor, foram usadas menos de 150 linhas, incluindo a implementação sequência, paralela e o teste de velocidade e resultado.

4 3 - C# Avançado

4.1 Aulas

- Aula 21 - SOLID
- Aula 22 - Padrões de Projeto Criacionais
- Aula 23 - Padrões de Projeto Comportamentais
- Aula 24 - Padrões de Projeto Estruturais
- Aula 25 - Padrões de Projeto Adicionais
- Aula 26 - Desafio 8
- Aula 27 - Programação Genérica e Reflexiva, Expressões e Atributos
- Aula 28 - Conexão com Banco de Dados, Object-Relational Mapping e Entity Framework
- Aula 29 - Desafio 9
- Aula 30 - Desafio 10

4.2 Duração Estimada

- 28 horas de conteúdo.
- 12 horas de desafios.
- 4 horas de revisão (opcional).
- 4 horas de avaliação (opcional).
- Total: 40 a 48 horas.

4.3 Overview

Este curso ensinará conceitos complexos de projeto de sistemas, recursos super avançados da linguagem C# e como usar o .NET para conectar de maneira eficiente ao banco de dados.

4.4 Competências

Em definição.

4.5 Aula 21 - SOLID

4.5.1

- Padrões de Projeto e o SOLID
- Princípio da Responsabilidade Única
- Princípio do Aberto-Fechado
- Princípio da Substituição de Liskov
- Princípio da Segregação de Interface
- Princípio da Inversão de Dependência
- Coesão e Acoplamento

4.5.2 Padrões de Projeto e o SOLID

Um Padrão de Projeto é um padrão estrutural onde usamos Orientação a Objetos entre outros aspectos da linguagens para resolver problemas específicos tentando melhorar ao máximo a qualidade de código. Hoje e nas próximas aulas aprenderemos a respeito desses padrões, qual seu objetivo e como implementá-los. Muitos desses padrões ajudam a manter o SOLID, que são 5 princípios de orientação a objetos que, em geral, trazem benefícios se mantidos. Vamos, rapidamente, compreender os 5 princípios a seguir.

4.5.3 Princípio da Responsabilidade Única

Criado por Robert Cecil Martin, o Single-Responsability Principle diz que as classes devem te apenas uma responsabilidade e não pode realizar duas funções. Mas ele vai mais afundo nisso e diz: As classes devem ter apenas um motivo para mudar. Se você tem uma classe que cria e manipula uma tela em uma aplicação ela possui dois motivos para mudar. Primeiro, caso a tela mude colocando-se novos botões ou retirando outros componentes, por exemplo. Segundo, caso você mude a tecnologia da aplicação, antes usando uma biblioteca e agora outra. Sua classe está errada pois tem 2 motivos para mudar e isso significa que você certamente terá que alterar o código por um dos motivos e pode acabar por criar bugs no outro. Com uma única responsabilidade, a classe fica muito mais robusta e resistente a erros.

4.5.4 Princípio do Aberto-Fechado

O Open-Closed Principle diz que uma classe deve estar aberta para extensão e fechada para modificação, isto é, uma classe deve poder ser utilizada por meio de agregação ou herança para ser estendida, ou seja, ter novas implementações em novas classes sem quebrar as antigas bem definidas, e, não deve ser

modificada, ou seja, uma vez pronta e integrada, não devemos alterar seu código para evitar quebrar outros componentes do software. Nas palavras de Bertrand Meyer: Um componente é dito aberto se ele pode ser estendível e seu comportamento pode ser usado e aprimorado em outras classes sem mudar seu código fonte. Um componente é dito fechado se ele já está sendo usado por outros componentes do sistema, logo, não deve ter seu código alterado. Assim, todo componente do sistema deve, em algum momento, ser considerado tanto aberto quanto fechado.

4.5.5 Princípio da Substituição de Liskov

Introduzido por Barbara Liskov, o Liskov-Substitution Principle é um dos princípios mais complexos de se compreender bem. Ele diz que uma classe mãe deve poder ser substituído por uma subclasse sem quebrar o programa. Mas o que seria realmente esse quebrar? Matematicamente o princípio é escrito da seguinte forma:

$$S < T : (\forall x \in T)\phi(x) \rightarrow (\forall y \in S)\phi(y)$$

Ou seja, sendo S subclasse do tipo T, se existe uma propriedade ϕ que é verdade para todos os objetos possíveis de T, ele deve ser verdade para todos os objetos de S.

Parte de atender esse princípio está garantido pelas linguagens de programação: Contravariância nos parâmetros de funções; Covariância nos retornos de funções; Sobrescrita de métodos respeitando as assinaturas; etc. Mas também traz coisas novas, como por exemplo, se uma classe base lança um conjunto de exceções sobre algumas circunstâncias, a classe base não deve lançar nos e diferentes exceções.

Outro conflito que o princípio pode trazer é o clássico exemplo Retângulo-Quadrado. Sabemos que Quadrado é um Retângulo, mas na programação, fazer com que um Quadrado herde de um Retângulo faz com que o mesmo perca um propriedade importante: Podemos definir uma largura e altura e sua área será o produto das duas. Essa é uma propriedade fundamental que faz com que o Quadrado quebre, já que o mesmo em uma restrição mais forte: Altua e Largura devem ser iguais. Adicionalmente, Liskov nos traz os seguintes pontos que devemos cumprir para manter essa relação: Pré-condições não devem ser mais fortes na classe filha (o quadrado tem pré-condições mais fortes). Pós-condições não devem ser menos fortes na classe filha (Ao definir Largura diferente de Altura, temos a pós-condição que a Altura e Largura se mantém diferentes após o calculo da área, já no Quadrado essa pós-condição é mais fraca pois quadrado alterará uma das suas medidas para manter sua condição de quadrado).

4.5.6 Princípio da Segregação de Interface

Ao usa uma interface, se existe alguma possibilidade de que uma classe implemente apenas parte da interface, devemos considerar separar a interface em duas interfaces menores.

4.5.7 Princípio da Inversão de Dependência

O Dependecy Inversion Principle diz que abstrações de alto-nível não devem depender das implementações de baixo-nível. Isso significa que precisamos de uma camada de abstração entre as estruturas. Por exemplo, um navegador Web pode ter vários motores de renderização podendo rodar várias linguagens dele, mas ele não deve ter uma referência direta a esses motores e sim a uma interface *IRenderEngine* desconectando a classe diretamente das implementações. Isso significa que fica fácil criar novas classes que herdam de *IRenderEngine* e você não compromete o funcionamento da classe do Navegador Web.

4.5.8 Coesão e Acoplamento

Um alto grau de Coesão implica que um módulo possui apenas componentes de software que trabalham juntos afim de realizar a mesma função. Muitas vezes é confundido com Princípio da Responsabilidade Única devido ao fato de que um módulo deve ter um objetivo muito bem definido e seus componentes devem contribuir cada um com sua responsabilidade para realizar esse objetivo, como um módulo de conexão com o banco de dados, por exemplo. Deve-se buscar um alto grau de coesão, mas sem exageros.

Um atlo grau de Acoplamento significa que um módulo está fortemente ligado a outro módulo e precisa dele para funcionar. Um erro no segundo módulo gera erros no primeiro. Em geral busca-se baixo grau de acoplamento, ou seja, desacoplamento. Por exemplo, você pode perceber que o Princípio da Inversão de Dependência reduz o acoplamento do sistema uma vez que as classes começam a depender de uma abstração que pode ter implementações trocadas a qualquer momento, não tornando um módulo altamente dependente de outro.

4.6 Aula 22 - Padrões de Projeto Criacionais

- Exercícios

4.6.1 Padrões de Projeto Criacionais

Padrões de Projeto Criacionais permite a criação de estrutura de forma flexível e robusta. Vamos então aprender os nossos primeiros três padrões de projeto.

4.6.2 Singleton

Talvez o padrão mais simples de todos e um dos primeiros a se aprender é o Singleton. Este padrão consiste em criar uma classe que só pode ter uma instância no sistema inteiro. Singletons em geral são mais flexíveis que classes puramente estáticas embora sejam baseadas nelas. Observe a estrutura básica:

```
1   // Usa o Singleton
2   Singleton.Current.SomeProperty = "num1";
3   Singleton.Current.OtherProperty = 4;
4   Singleton.Current.SomeMethod();
5
6   // Reinicia o Singleton
7   Singleton.New("num2");
8   // Teóricamente e possível fazer isso: Colocar o obejto singleton numa
    variável
9   var sin = Singleton.Current;
10  sin.OtherProperty = 10;
11  sin.SomeMethod();
12
13  // Em teoria podemos ter duas instâncias do objeto caso você possa criar
    uma nova instância
14  // Isso não necessariamente é ruim
15  Singleton.New("num3");
16  var newSin = Singleton.Current;
17  if (sin.SomeProperty != newSin.SomeProperty)
18      Console.WriteLine("Difernete!");
19
20  public class Singleton
21  {
22      // Construtores Privados, ou seja, ninguém pode criar uma instâcia de
    Singleton
23      // Isso força que a classe seja usada para seu design proposto:
    Existir apenas um.
24      private Singleton() { }
25
26      // Podem existir sobrecargas para incializações dentro da classe
27      private Singleton(string text)
28          => this.SomeProperty = text;
29
30      // Campo estática e privada que pode iniciar com algum valor ou com
    valor nulo.
31      private static Singleton crr = new Singleton();
32
33      // Propriedade estática publica para acessar a insância atual.
34      public static Singleton Current => crr;
35
36      // Várias Propriedades e métodos não estáticos do objeto Singleton
37      public string SomeProperty { get; set; }
38      public int OtherProperty { get; set; }
39
40      public void SomeMethod()
41          => Console.WriteLine($"{SomeProperty} = {OtherProperty}");
42
43      // Métodos estáticos que permitem recriar/manipular a instância atual
    (totalmente opcional)
44      public static void New()
```

```
45          => crr = new Singleton();
46      public static void New(string text)
47          => crr = new Singleton(text);
48  }
```

Esse padrão é muito útil quando queremos uma classe que controle uma instância muito importante da nossa aplicação. Por exemplo, em um jogo a classe Jogo que controla o Save do usuário é interessante. Singletons para conexões com banco de dados também podem ser uma opção, ou seja, existe apenas uma conexão com banco de dados embora ela possa ser recriada e conectada a outro banco.

4.6.3 Builder

O Builder é um padrão de projeto que facilita criação de objetos complexos e de forma parcelada. Ao invés de usarmos construtores gigantes e confusos podemos usar um Builder para fazer uma construção parcial, e melhor, podemos passar o Builder para outras classes que podem fazer passos da construção para nós.

```
1   // Ao se mudar o Builder, muda-se completamente o funcionamento interno e até mesmo o produto resultante
2   var builder = new ConcreteBuilderB();
3
4   // Bela sintaxe de criação de objeto
5   // Pode ser facilmente modificada com base nas necessidades da aplicação
6   var product = builder
7       .AddNumber(100)
8       .AddStirng("Texto a seguir:\n")
9       .AddManyTimes("\tTexto\n", 3)
10      .AddNumber(3)
11      .Build();
12
13  Console.WriteLine(product.Message);
14
15  // Interface que define o que o Builder precisa ter para construir o objeto, além do método Build que retorna o produto
16  public interface IBuilder
17  {
18      // Os métodos podem ou não retornar IBuilder, se retornarem possibilita sintaxes como no começo do código
19      // que são bem compactas e compreensíveis. Mas não é uma obrigação.
20      IBuilder AddNumber(int num);
21      IBuilder AddStirng(string text);
22      Product Build();
23  }
24
25  // Produto a ser construído pelo Builder
26  public class Product
27  {
28      public Product(string text)
29          => this.Message = text;
30
31      public string Message { get; set; }
32  }
33
34  // Builder Concetro, uma implementação possível do Builder
35  public class ConcreteBuilderA : IBuilder
36  {
37      private string message = string.Empty;
38      public IBuilder AddNumber(int num)
39      {
40          // Cria duas stirngs novas, x = num.Tostring() e y = message + x
41          message += num.ToString();
42          return this;
```

```
43      }
44
45      public IBuilder AddStirng(string text)
46      {
47          // Cria uma stirng nova
48          message += text;
49          return this;
50      }
51
52      public Product Build()
53          => new Product(message);
54  }
55
56  // Builder Concetro, uma implementação possível do Builder
57  // Ele é mais optimizado pois evita a criação de muitas strings criando
    apenas
58  // Uma string nova no método Build.
59  public class ConcreteBuilderB : IBuilder
60  {
61      private List<string> stirngs = new List<string>();
62      public IBuilder AddNumber(int num)
63      {
64          stirngs.Add(num.ToString());
65          return this;
66      }
67
68      public IBuilder AddStirng(string text)
69      {
70          stirngs.Add(text);
71          return this;
72      }
73
74      // Criação inteligente do produto
75      public Product Build()
76      {
77          var strSize = stirngs.Sum(s => s.Length);
78          char[] characters = new char[strSize];
79
80          int i = 0;
81          foreach (var str in stirngs)
82          {
83              foreach (var c in str)
84              {
85                  characters[i] = c;
86                  i++;
87              }
88          }
89
90          var message = new string(characters);
91          return new Product(message);
92      }
93  }
94
95  // Método de Extensão que podem adicionar novas funcionalidades ao Builder
    com base na interface
96  public static class BuilderExtension
97  {
98      public static IBuilder AddManyTimes(this IBuilder builder, string str,
    int times)
99      {
```

```
100             for (int i = 0; i < times; i++)
101                 builder.AddStirng(str);
102             return builder;
103         }
104     }
```

Com o Builder podemos fazer inicializações de projetos ou até mesmo de objetos simples, como o StringBuilder do C# que funciona de forma semelhante ao *ConcreteBuilderB*, que te ajuda criar strings economizando memória.

4.6.4 Abstract Factory

O Abstract Factory é o padrão mais complexo de criação de objetos e permite criar vários produtos diferentes a partir de várias implementações de fábricas. Fábricas são objetos que retornam sempre o mesmo objeto. O que muda o comportamento do software é qual fábrica concreta foi escolhida.

```
1   // Trabalhando com um produto desconhecido
2   int data = int.Parse(Console.ReadLine() ?? "0");
3   IAbstractFactory factory = data % 2 == 0
4       ? new ConcreteFactory1()
5       : new ConcreteFactory2();
6
7   var pA = factory.CreateProductA();
8   var pB = factory.CreateProductB();
9
10
11  // Trabalhamos com Produtos Abstratos
12  public abstract class AbstractProductA
13  {
14      public float PropertyA { get; set; }
15      public abstract float MethodA();
16  }
17
18  public abstract class AbstractProductB
19  {
20      public float PropertyB { get; set; }
21      public abstract float MethodB();
22  }
23
24  // Cada Produto Abstrato pode ter muitos produtos Concretos que funcionam
    de formas diferentes
25  public class ConcreteProductA1 : AbstractProductA
26  {
27      public override float MethodA()
28          => 2 * PropertyA;
29  }
30
31  public class ConcreteProductA2 : AbstractProductA
32  {
33      public override float MethodA()
34          => PropertyA * PropertyA;
35  }
36  public class ConcreteProductB1 : AbstractProductB
37  {
38      public override float MethodB()
39          => PropertyB + 2;
40  }
41
42  public class ConcreteProductB2 : AbstractProductB
43  {
44      public override float MethodB()
```

```
        => PropertyB - 10;
}

// Uma Fábrica pode criar qualquer produto mas você nunca sabe qual
public interface IAbstractFactory
{
    AbstractProductA CreateProductA();
    AbstractProductB CreateProductB();
}

// As implementações das fábricas decidem qual os produtos concretos que você receberá
public class ConcreteFactory1 : IAbstractFactory
{
    public AbstractProductA CreateProductA()
        => new ConcreteProductA1();

    public AbstractProductB CreateProductB()
        => new ConcreteProductB1();
}

public class ConcreteFactory2 : IAbstractFactory
{
    public AbstractProductA CreateProductA()
        => new ConcreteProductA2();

    public AbstractProductB CreateProductB()
        => new ConcreteProductB2();
}
```

Agora vamos a um exemplo usando os três padrões para que fique mais claro como funciona na prática.

4.6.5 Exemplo: Sistema Financerio Adptável a Diversas Leis

Considere que você abre uma empresa tanto no Brasil quanto na Argentina. Como garantir que a implementação respeite as Leis dos diferentes países sem encher de If's por todos os lados. Como permitir que os diferentes processos sejam executados de forma diferente de acordo com simples configurações básicas. Observe os códigos a seguir:

Process.cs

```
public abstract class Process
{
    public abstract string Title { get; }
}
```

Employe,cs

```
public class Employe
{
    public string Name { get; set; }
    public decimal Wage { get; set; }
}
```

DismissalProcess.cs

```
public abstract class DismissalProcess : Process
```

```
{
    public abstract void Apply(DismissalArgs args);
}
```

WagePaymentProcess.cs

```
public abstract class WagePaymentProcess : Process
{
    public abstract void Apply(WagePaymentArgs args);
}
```

Args.cs

```
public class ProcessArgs
{
    private static ProcessArgs empty = new ProcessArgs();
    public static ProcessArgs Empty => empty;
}

public class DismissalArgs : ProcessArgs
{
    public Company Company { get; set; }
    public Employe Employe { get; set; }
}

public class WagePaymentArgs : ProcessArgs
{
    public Company Company { get; set; }
    public Employe Employe { get; set; }
}
```

IProcessFactory.cs

```
public interface IProcessFactory
{
    public WagePaymentProcess CreateWagePaymentProcess();
    public DismissalProcess CreateDismissalProcess();
}
```

Brazil.cs

```
public class BrazilDismissalProcess : DismissalProcess
{
    public override string Title => "Demissão de Funcionário";

    public override void Apply(DismissalArgs args)
    {
        args.Company.Money -= 2 * args.Employe.Wage;
    }
}

public class BrazilWagePaymentProcess : WagePaymentProcess
{
```

```
    public override string Title => "Pagamento de Salário";

    public override void Apply(WagePaymentArgs args)
    {
        args.Company.Money -= 1.45m * args.Employe.Wage + 500;
    }
}

public class BrazilProcessFactory : IProcessFactory
{
    public DismissalProcess CreateDismissalProcess()
        => new BrazilDismissalProcess();

    public WagePaymentProcess CreateWagePaymentProcess()
        => new BrazilWagePaymentProcess();
}
```

Argentina.cs

```
public class ArgentinaDismissalProcess : DismissalProcess
{
    public override string Title => "Despido de Empleados";

    public override void Apply(DismissalArgs args)
    {
        args.Company.Money -= 3 * args.Employe.Wage;
    }
}

public class ArgentinaWagePaymentProcess : WagePaymentProcess
{
    public override string Title => "Pago de salario";

    public override void Apply(WagePaymentArgs args)
    {
        args.Company.Money -= 1.65m * args.Employe.Wage + 700;
    }
}

public class ArgentinaProcessFactory : IProcessFactory
{
    public DismissalProcess CreateDismissalProcess()
        => new ArgentinaDismissalProcess();

    public WagePaymentProcess CreateWagePaymentProcess()
        => new ArgentinaWagePaymentProcess();
}
```

Company.cs

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

public class Company
{
    // Estrutura Singleton
```

```
7       private Company() { }
8       private static Company crr = null;
9       public static Company Current => crr;
10
11      public string Name { get; set; }
12      public decimal Money { get; set; }
13
14      private List<Employe> employes = new List<Employe>();
15      public IEnumerable<Employe> Employes => employes;
16
17      private DismissalProcess dismissalProcess = null;
18      private WagePaymentProcess wagePaymentProcess = null;
19
20      public void Contract(Employe employe)
21      {
22          employes.Add(employe);
23      }
24
25      public void Dismiss(string name)
26      {
27          var employe = this.Employes.FirstOrDefault(x => x.Name == name);
28
29          if (employe == null)
30              return;
31
32          DismissalArgs args = new DismissalArgs();
33          args.Employe = employe;
34          args.Company = this;
35
36          dismissalProcess.Apply(args);
37
38          employes.Remove(employe);
39      }
40
41      public void PayWages()
42      {
43          foreach (var employe in this.Employes)
44          {
45              WagePaymentArgs args = new WagePaymentArgs();
46              args.Employe = employe;
47              args.Company = this;
48
49              wagePaymentProcess.Apply(args);
50          }
51      }
52
53      // Classes Aninhadas: Isso permite que CompanyBuilder veja todos os
    campos privados de Company
54      public class CompanyBuilder
55      {
56          private Company company = new Company();
57
58          public Company Build()
59              => this.company;
60
61          public CompanyBuilder SetName(string name)
62          {
63              company.Name = name;
64              return this;
65          }
```

```
66
67          public CompanyBuilder SetFactory(IProcessFactory factory)
68          {
69              company.dismissalProcess = factory.CreateDismissalProcess();
70              company.wagePaymentProcess =
    factory.CreateWagePaymentProcess();
71              return this;
72          }
73
74          public CompanyBuilder SetInitialCapital(decimal money)
75          {
76              company.Money = money;
77              return this;
78          }
79
80          public CompanyBuilder AddEmploye(string name, decimal wage)
81          {
82              Employe employe = new Employe();
83              employe.Name = name;
84              employe.Wage = wage;
85              company.Contract(employe);
86              return this;
87          }
88      }
89
90      public static CompanyBuilder GetBuilder()
91          => new CompanyBuilder();
92
93      public static void New(CompanyBuilder builder)
94          => crr = builder.Build();
95  }
```

CompanyBuilderExtension.cs

```
1   public static class CompanyBuilderExtension
2   {
3       public static Company.CompanyBuilder InBrazil(this
     Company.CompanyBuilder builder)
4       {
5           builder.SetFactory(new BrazilProcessFactory());
6           return builder;
7       }
8       public static Company.CompanyBuilder InArgentina(this
     Company.CompanyBuilder builder)
9       {
10          builder.SetFactory(new ArgentinaProcessFactory());
11          return builder;
12      }
13  }
```

Program.cs

```
1   var builder = Company.GetBuilder();
2
3   builder
4       .SetName("Mercado Libre")
```

```
    .InArgentina()
    .SetInitialCapital(20_000_000);

builder
    .AddEmploye("Marquitos Guapo", 50_000)
    .AddEmploye("Paulito Pino", 20_000);

Company.New(builder);

// Me rendí, me voy a Brasil
builder = Company.GetBuilder();

builder
    .SetName("Mercado Livre")
    .InBrazil()
    .SetInitialCapital(1_000_000);

builder
    .AddEmploye("Marcos Bonito", 2_500)
    .AddEmploye("Paulo Pinheiro", 1_000);

Company.New(builder);

Employe employe = new Employe();
employe.Name = "Xispita";
employe.Wage = 2_000;
Company.Current.Contract(employe);

Company.Current.Dismiss("Marcos Bonito");

Company.Current.PayWages();
```

4.6.6 Exercícios

1. Modifique a Abstração anterior para possibilitar que Empresa tenha a operação de contratação personalizável a cada país, bem como pagamento de salários e demissão.
2. Modifique a Abstração anterior para possibilitar que Empresa opere nos Estados Unidos da América.

4.7 Aula 23 - Padrões de Projeto Comportamentais

- Proxy
- Padrões de Projeto Comportamentais
- Template Method
- Strategy
- State
- Exemplo: Bot para jogos digitais
- Exercícios

4.7.1 Proxy

Proxy é um Padrão Estrutural para controlar o acesso a recursos que precisam de tempo para serem processados. Ele pode te ajudar a acessar um banco de dados, buscar imagens na Web e mostrar uma imagem mais simples enquanto a mesma carrega, mostrar um calculo simples enquanto um calculo complexo carrega ou carregar dados em Cache. Todas essas operações são feitas com o padrão Proxy:

```
using System;
using System.Threading;

Service service = new Service();
Proxy proxy = new Proxy(service);

```

```
byte[] arr = new byte[1024 * 1024 * 1024];
Random rand = new Random();
rand.NextBytes(arr);

Console.Clear();
while (true)
{
    Console.SetCursorPosition(0, 0);
    Console.WriteLine(proxy.GetSum(arr));
}


public interface IService
{
    string GetSum(byte[] arr);
}

public class Service : IService
{
    public string GetSum(byte[] arr)
    {
        long sum = 0;
        for (int i = 0; i < arr.Length; i++)
            sum += arr[i];
        return $"Total: {sum} (exato)";
    }
}

public class Proxy : IService
{
    private IService realService = null;
    private string realOutput = null;
    private Thread thread = null;

    public Proxy(IService service)
        => this.realService = service;

    public string GetSum(byte[] arr)
    {
        if (realOutput != null)
            return realOutput;

        if (thread == null)
            startService(arr);

        return $"Total: {(128 * arr.LongLength)} (aprox)";
    }

    private void startService(byte[] arr)
    {
        thread = new Thread(() =>
        {
            var result = realService.GetSum(arr);
            this.realOutput = result;
        });
        thread.Start();
    }
}
```

Neste exemplo, apresentamos uma mensagem diferente enquanto esperamos o resultado concreto.

Note que o Proxy lembra muito o Decorator. A diferença fundamental entre eles é o proposito. Muitos padrões são estruturalmente parecidos e o que os diferencia é como você os aplica para resolver os problemas e quais tipos de problemas você quer resolver.

4.7.2 Padrões de Projeto Comportamentais

Nosso último grupo de padrões de projeto tem por objetivo representar comportamentos. Antes criamos objetos complexos, depois construímos estruturas complexas, agora vamos representar comportamentos complexos.

4.7.3 Template Method

Nosso primeiro padrão é outro muito simples de se compreender e usar. O Template Method permite que você faça a implementação parcial de alguma funcionalidade nas classes bases.

```
1  using System;
2
3  BaseClass a = new ImplA();
4  BaseClass b = new ImplB();
5
6  a.Func(); // Hello, World!
7  b.Func(); // Olá... Mundo?
8
9
10 // Pode ou não ser abstrata
11 public class BaseClass
12 {
13     // Método Fixo
14     public void Func()
15     {
16         SubFuncA();
17         SubFuncB();
18         SubFuncC();
19     }
20
21     protected virtual void SubFuncA()
22     {
23         Console.Write("Olá, ");
24     }
25
26     protected virtual void SubFuncB()
27     {
28         Console.Write("Mundo");
29     }
30
31     protected virtual void SubFuncC()
32     {
33         Console.WriteLine("!");
34     }
35 }
36
37 public class ImplA : BaseClass
38 {
39     protected override void SubFuncA()
40     {
41         Console.Write("Hello, ");
42     }
43
44     protected override void SubFuncB()
45     {
46         Console.Write("World");
47     }
48 }
```

```
49
50  public class ImplB : BaseClass
51  {
52      protected override void SubFuncA()
53      {
54          Console.Write("Olá... ");
55      }
56
57      protected override void SubFuncC()
58      {
59          Console.WriteLine("?");
60      }
61  }
```

4.7.4 Strategy

O Strategy permite que criemos um contexto que pode mudar o seu comportamento conforme você define as estratégias dentro da classe.

```
1   using System;
2   using static System.Console;
3
4   Context context = new Context();
5
6   context.SetStrategy(new ConcreteStrategyA());
7   WriteLine(context.Compute(2, 3));
8
9   context.SetStrategy(new ConcreteStrategyB());
10  WriteLine(context.Compute(2, 3));
11
12  context.SetStrategy(new ConcreteStrategyC());
13  WriteLine(context.Compute(2, 3));
14
15  public interface IStrategy
16  {
17      double MakeOperation(double a, double b);
18  }
19
20  public class ConcreteStrategyA : IStrategy
21  {
22      public double MakeOperation(double a, double b)
23          => a + b;
24  }
25
26  public class ConcreteStrategyB : IStrategy
27  {
28      public double MakeOperation(double a, double b)
29          => a * b;
30  }
31
32  public class ConcreteStrategyC : IStrategy
33  {
34      public double MakeOperation(double a, double b)
35          => Math.Pow(a, b);
36  }
37
38  public class Context
39  {
40      private IStrategy strategy;
41
```

```
42      public void SetStrategy(IStrategy strategy)
43          => this.strategy = strategy;
44
45      public double Compute(double a, double b)
46          => this.strategy.MakeOperation(a, b);
47  }
```

4.7.5 State

State é uma evolução do Strategy. A diferença é que o estado, diferente da estratégia, é apenas um estado que controla o contexto e pode decidir os próximos estados.

```
1   using System;
2   using static System.Console;
3
4   Context context = new Context();
5   context.ChangeState(new ConcreteStateA());
6
7   WriteLine(context.Compute(2, 3));
8   WriteLine(context.Compute(2, 3));
9   WriteLine(context.Compute(2, 3));
10
11  public class Context
12  {
13      private State state;
14
15      public void ChangeState(State state)
16      {
17          this.state = state;
18          this.state.SetContext(this);
19      }
20
21      public double Compute(double a, double b)
22          => this.state.MakeOperation(a, b);
23  }
24
25  public abstract class State
26  {
27      protected Context context = null;
28      public void SetContext(Context context)
29          => this.context = context;
30
31      public abstract double MakeOperation(double a, double b);
32  }
33
34  public class ConcreteStateA : State
35  {
36      public override double MakeOperation(double a, double b)
37      {
38          // Muda o Estado para o B
39          this.context.ChangeState(new ConcreteStateB());
40          return a + b;
41      }
42
43  }
44
45  public class ConcreteStateB : State
46  {
47      public override double MakeOperation(double a, double b)
48      {
```

```
49          // Muda o Estado para o C
50          this.context.ChangeState(new ConcreteStateC());
51          return a * b;
52      }
53  }
54
55  public class ConcreteStateC : State
56  {
57      public override double MakeOperation(double a, double b)
58      {
59          // Muda o Estado para o A
60          this.context.ChangeState(new ConcreteStateA());
61          return Math.Pow(a, b);
62      }
63  }
```

4.7.6 Exemplo: Bot para jogos digitais

Vamos praticar o Proxy, Template Method, Strategy e State através de um exemplo. Vamos fazer um Bot que você pode aplicar em muitos contextos como jogos, por exemplo.

4.7.7 Exercícios

4.8 Aula 24 - Padrões de Projeto Estruturais

4.8.1 Flyweigth

Flyweigth é um padrão criacional que nos permite reduzir a repetição de objetos pequenos na memória, por exemplo, exceções. Ele lembra o Singleton na sua estrutura, mas permite que o objeto existe muitas vezes. A ideia é reduzir os gastos de memória.

```
1   using static System.Console;
2   using System.Collections.Generic;
3   using static System.Diagnostics.Process;
4
5   List<MyObject> list = new List<MyObject>();
6
7   list.Add(Flyweigth.ObjectA);
8   list.Add(Flyweigth.ObjectB);
9   list.Add(Flyweigth.ObjectC);
10
11  list.Add(Flyweigth.ObjectA);
12  list.Add(Flyweigth.ObjectA);
13  list.Add(Flyweigth.ObjectB);
14  list.Add(Flyweigth.ObjectB);
15  list.Add(Flyweigth.ObjectC);
16  list.Add(Flyweigth.ObjectC);
17
18  list.Add(Flyweigth.ObjectA);
19  list.Add(Flyweigth.ObjectC);
20  list.Add(Flyweigth.ObjectC);
21  list.Add(Flyweigth.ObjectC);
22  list.Add(Flyweigth.ObjectC);
23
```

```
foreach (var obj in list)
    obj.Show();

// Vê gasto de memória da aplicação
WriteLine(GetCurrentProcess().PrivateMemorySize64);

public static class Flyweigth
{
    private static MyObjectA objA = null;
    public static MyObjectA ObjectA
    {
        get
        {
            if (objA == null)
                objA = new MyObjectA();
            return objA;
        }
    }

    private static MyObjectB objB = null;
    public static MyObjectB ObjectB
    {
        get
        {
            if (objB == null)
                objB = new MyObjectB();
            return objB;
        }
    }

    private static MyObjectC objC = null;
    public static MyObjectC ObjectC
    {
        get
        {
            if (objC == null)
                objC = new MyObjectC();
            return objC;
        }
    }
}

// Classe Base Opcional
public class MyObject
{
    public virtual void Show()
    {
        WriteLine("Olá, Mundo!");
    }
}

public class MyObjectA : MyObject
{
    public override void Show()
    {
        Write("Olá,");
    }
}


```

```
84  public class MyObjectB : MyObject
85  {
86      public override void Show()
87      {
88          Write(" Mun");
89
90      }
91  }
92
93  public class MyObjectC : MyObject
94  {
95      public override void Show()
96      {
97          WriteLine("do!");
98      }
99  }
```

Podemos testar e observar a pequena economia de memória já neste exemplo simples.

4.8.2 Padrões de Projeto Estruturais

Enquanto padrões criacionais nos ajudam a inicializar e criar objetos e a forma como usamos, padrões estruturais permite que criemos estrutura complexas para representar coisas complexas.

4.8.3 Facade

Facade é um padrão simples bem como o Singleton e é um dos primerios a serem aprendidos. Basicamente a ideia dele é junta várias classes e subsistemas complexos em uma única classe que contra uma ou mais operações.

```
1   using static System.Console;
2
3   Facade facade = new Facade();
4   facade.Print();
5
6
7   // Pode, muitas vezes, ser estático
8   public class Facade
9   {
10      // Usa o subsistema para fazer uma tarefa simples e desejada
11      public void Print()
12      {
13          var a = new ClassA();
14          var b = new ClassB();
15          var c = new ClassC();
16          var d = new ClassD();
17          string result = a.Get() + b.Get() + c.Get() + d.Get();
18          WriteLine(result);
19      }
20  }
21
22  // Subsistema complexo abaixo
23  public class ClassA
24  {
25      public string Get()
26      {
27          return "Olá";
28      }
29  }
30
31  public class ClassB
32  {
33      public string Get()
```

```
34      {
35          return ", ";
36      }
37  }
38
39  public class ClassC
40  {
41      public string Get()
42      {
43          return "Mundo";
44      }
45  }
46
47  public class ClassD
48  {
49      public string Get()
50      {
51          return "!";
52      }
53  }
```

4.8.4 Decorator

O Decorator permite decorar um objeto com uma implementação extra. A ideia é usar agregação (e não composição) para que um objeto possa estender o comportamento de outro. Observe:

```
1   using static System.Console;
2
3   var a = new ClassA();
4   var b = new ClassB();
5
6   var d1 = new Decorator();
7   var d2 = new Decorator();
8   var d3 = new Decorator();
9
10  d1.Wrapped = a;
11  WriteLine(d1); //Olá, Mundo!
12
13  d2.Wrapped = b;
14  WriteLine(d2); //Xispita!
15
16  d3.Wrapped = d2;
17  WriteLine(d3); //Xispita!!
18
19  public abstract class BaseClass
20  {
21      public abstract string GetString();
22
23      public override string ToString()
24          => GetString();
25  }
26
27  public class ClassA : BaseClass
28  {
29      public override string GetString()
30      {
31          return "Olá, Mundo";
32      }
33  }
34
```

```
35  public class ClassB : BaseClass
36  {
37      public override string GetString()
38      {
39          return "Xispita";
40      }
41  }
42
43  // Pode ser uma classe base para vários decorators
44  public class Decorator : BaseClass
45  {
46      public BaseClass Wrapped { get; set; }
47      public override string GetString()
48      {
49          return Wrapped.GetString() + "!";
50      }
51  }
```

4.8.5 Composite

Composite é semelhante ao decorator, mas ele agrega uma lista de componentes podendo criar uma relação hierárquica:

```
1   using static System.Console;
2   using System.Collections.Generic;
3   using System.Text;
4
5   var a = new ClassA();
6   var b = new ClassB();
7
8   var c1 = new Composite();
9   var c2 = new Composite();
10
11  c1.Add(a);
12  c1.Add(c2);
13
14  c2.Add(a);
15  c2.Add(b);
16
17  WriteLine(c1); //Olá, MundoOlá, MundoXispita
18  /*
19   *    c1
20   *   /  \
21   *  a   c2
22   *     /  \
23   *    a    b
24   */
25
26  public abstract class BaseClass
27  {
28      public abstract string GetString();
29
30      public override string ToString()
31          => GetString();
32  }
33
34  public class ClassA : BaseClass
35  {
36      public override string GetString()
37      {
```

```
        return "Olá, Mundo";
    }
}

public class ClassB : BaseClass
{
    public override string GetString()
    {
        return "Xispita";
    }
}

public class Composite : BaseClass
{
    private List<BaseClass> list = new List<BaseClass>();
    public IEnumerable<BaseClass> Classes => list;

    public void Add(BaseClass obj)
        => list.Add(obj);

    public override string GetString()
    {
        StringBuilder sb = new StringBuilder();
        foreach (var obj in list)
            sb.Append(obj);
        return sb.ToString();
    }
}
```

4.8.6 Exemplo: Abstração Algébrica de Funções

Vamos agora a um exemplo prático de onde usar os padrões aprendidos no dia de hoje. Vamos implementar um sistema de abstração de funções, isto é, uma abstração em que podemos construir uma biblioteca matemática onde podemos representar funções matemáticas como:

- $f(x) = x$
- $f(x) = cos(x^2)$
- $f(x) = ln(x) + sin(x^3 + 2)$
- $f(x) = \frac{e^x + 3 * x + 1}{sin(x) + cos(x + 0.1)} - 2$

Para isso precisaremos representar várias estruturas complexas que nos ajude a controlar e utilizar dessas funções.

Function.cs (versão 1)

```
// Componente base da estrutura, representa uma função qualquer
public abstract class Function
{
    // Permite usar y = f[10]
    public double this[double x]
        => calcule(x);

    protected abstract double calcule(double x);

    // Bônus para quem sabe derivar
    public abstract Function Derive();
}
```

Constant.cs (versão 1)

```
1  // Componente qualquer, representa função constante, por exemplo, f(x) = 4
2  public class Constant : Function
3  {
4      private double value;
5      public Constant(double value)
6          => this.value = value;
7
8      protected override double calcule(double x) => this.value;
9
10     // Bônus para quem sabe derivar
11     public override Function Derive() => new Constant(0);
12 }
```

Linear.cs (versão 1)

```
1  // Componente qualquer, representa função linear f(x) = x
2  public class Linear : Function
3  {
4      protected override double calcule(double x) => x;
5
6      // Bônus para quem sabe derivar
7      public override Function Derive() => new Constant(1);
8  }
```

Aggregation.cs

```
1  // Classe base para decorators
2  public abstract class Aggregation : Function
3  {
4      public Function InnerFunction { get; set; }
5
6      protected abstract double calcule(Function f, double x);
7
8      protected override double calcule(double x)
9          => calcule(InnerFunction, x);
10 }
```

Cosine.cs

```
1  using System;
2  // Função cosseno que decora uma função interna
3  public class Cosine : Aggregation
4  {
5      protected override double calcule(Function f, double x)
6          => Math.Cos(f[x]);
7
8      public override Function Derive()
9      {
10         // sin(f)' = cos(f) * f'
11         throw new NotImplementedException();
12     }
```

```
}
```

Sine.cs

```
using System;
// Função seno que decora uma função interna
public class Sine : Aggregation
{
    protected override double calcule(Function f, double x)
        => Math.Sin(f[x]);

    public override Function Derive()
    {
        // cos(f)' = -sin(f) * f'
        throw new NotImplementedException();
    }
}
```

Composition.cs

```
using System.Collections.Generic;

// Classe base para composições de funções
public abstract class Composition : Function
{
    protected List<Function> functions = new List<Function>();
    public IEnumerable<Function> InnerFunctions => functions;

    public void Add(Function function)
        => this.functions.Add(function);
}
```

Sum.cs

```
using System.Linq;

// Composição Concreta, no caso representa soma de funções
public class Sum : Composition
{
    protected override double calcule(double x)
        => functions.Select(f => f[x]).Sum();

    public override Function Derive()
    {
        Sum sum = new Sum();

        foreach (var f in this.functions)
            sum.Add(f.Derive());

        return sum;
    }
}
```

FunctionPool.cs

```
1  // Flyweigth
2  public static class FunctionPool
3  {
4      private static Function linearFunction = null;
5      public static Function LinearFunction
6      {
7          get
8          {
9              linearFunction ??= new Linear();
10             return linearFunction;
11         }
12     }
13 }
```

MathLib.cs

```
1  // Facade
2  public static class MathLib
3  {
4      // Geralmente começamos com letras maiúsculas em membros publicos
5      // usando PascalCase, aqui está uma das poucas ou únicas situações
6      // onde é aceitável quebar essa regra: Convenção.
7      // Na matemática é comum usar letras minúsculas para tudo, e como
8      // esse seria um sistema para matemáticos, isso se torna propicio.
9      public static Function x => FunctionPool.LinearFunction;
10     public static Function cos(Function x)
11     {
12         var f = new Cosine();
13         f.InnerFunction = x;
14         return f;
15     }
16     public static Function sin(Function x)
17     {
18         var f = new Sine();
19         f.InnerFunction = x;
20         return f;
21     }
22 }
```

Function.cs

```
1  public abstract class Function
2  {
3      public double this[double x]
4          => calcule(x);
5
6      protected abstract double calcule(double x);
7
8      public abstract Function Derive();
9
10     // Conversões implitas e operações personalizadas para tornar mais
   agradável o uso da biblioteca
11     public static implicit operator Function(double c)
```

```
        => new Constant(c);

    public static Function operator +(Function f, Function g)
    {
        Sum sum = new Sum();
        sum.Add(f);
        sum.Add(g);
        return sum;
    }
}
```

Constant.cs

```
// Classe melhorada graças a conversão implícita
public class Constant : Function
{
    private double value;
    public Constant(double value)
        => this.value = value;

    protected override double calcule(double x) => this.value;

    public override Function Derive() => 0; // Mais simples!
}
```

Linear.cs

```
// Classe melhorada graças a conversão implícita
public class Linear : Function
{
    protected override double calcule(double x) => x;

    public override Function Derive() => 1; // Mais simples!
}
```

Program.cs

```
using static MathLib;

var f = cos(x + 3) + 1;
var y = f[0.14159265359];
Console.WriteLine(y);
```

4.8.7 Exercícios

1. Implemente a função Ln(x) (Log na base e = 2.71828182846...
2. Implemente a multiplicação de funções

4.9 Aula 25 - Padrões de Projeto Adicionais

4.9.1 Bridge

O Bridge é um padrão estrutural que é estrutualmente semelhane a padrões como o Strategy, porém, com uma finalidade e uso totalmente diferente. Sua ideia é reduzir o trabalho e diminuir os possíveis erros e inconsistências ao modificar um sistema. A ideia dele é separar implementação de abstração. A ideia é bem complexa, na verdade, e dificil de compreender muitas vezes, mas observe o exemplo a seguir.

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

List<Product> products = new List<Product>()
{
    new Product()
    {
        Name = "Xispita",
        Price = 10.5f
    },
    new Product()
    {
        Name = "Caixa de Ponteiros",
        Price = 3f
    }
};

Abstraction abstraction = new Abstraction();

abstraction.implementation = new NormalImplementation();
abstraction.DrawMarket("Treviloja normal", products);

Console.WriteLine();
Console.WriteLine();
Console.WriteLine();

abstraction.implementation = new SpecialImplementation();
abstraction.DrawMarket("Treviloja especial", products);

public class Product
{
    public string Name { get; set; }
    public float Price { get; set; }
}

public class Abstraction
{
    public Implementation implementation { get; set; }

    public virtual void DrawMarket(string title, List<Product> products)
    {
        implementation.AddString(title);
        implementation.AddNewLine();

        foreach (var product in products)
        {
            implementation.AddString(product.Name);
            implementation.AddFloat(product.Price);
            implementation.AddNewLine();
        }

        implementation.Print();
    }
}

public interface Implementation
{
    void AddString(string s);
```

```
60      void AddFloat(float f);
61      void AddNewLine();
62      void Print();
63  }
64
65  public class NormalImplementation : Implementation
66  {
67      StringBuilder sb = new StringBuilder();
68
69      public void AddFloat(float s)
70          => sb.Append(s);
71
72      public void AddNewLine()
73          => sb.AppendLine();
74
75      public void AddString(string f)
76          => sb.Append(f);
77
78      public void Print()
79      {
80          Console.Write(sb.ToString());
81          sb.Clear();
82      }
83  }
84
85  public class SpecialImplementation : Implementation
86  {
87      int size = 0;
88      List<string> lines = new List<string>();
89      StringBuilder sb = new StringBuilder();
90
91      public void AddString(string s)
92      {
93          size += s.Length;
94          sb.Append(s);
95      }
96
97      public void AddFloat(float f)
98      {
99          string str = $"{f: 0.00}";
100         AddString(str);
101     }
102
103     public void Print()
104     {
105         foreach (var line in lines)
106             Console.WriteLine(line);
107         lines.Clear();
108     }
109
110     public void AddNewLine()
111     {
112         formatString();
113         lines.Add("");
114     }
115
116     private void formatString()
117     {
118         sb.Insert(0, "\n");
119         sb.AppendLine();
```

```
120             for (int i = 0; i < size; i++)
121             {
122                 sb.Insert(0, "-");
123                 sb.Append("-");
124             }
125             lines.Add(sb.ToString());
126             sb.Clear();
127             size = 0;
128         }
129     }
```

O exemplo acima é muito interessante. A abstração usa as funções da implementação para construir desenhar uma loja com produtos. A depender da implementação, a forma com que a loja será desenhada muda. Isso é perfeito, pois podemos mudar a loja inteira sem alterar a abstração e sem alterar as implementações existentes, basta criar uma nova implementação. Além disso, se modificarmos a abstração ou criarmos classes filhas da abstração que fazem mais coisas, reescrevendo métodos ou adicionando, não quebramos o funcionamento da implementação que fica separada a parte.

4.9.2 Command

Command é um padrão comportamental usado para encapsular e parametrizar comandos que podem ser ativados em vários momentos. Geralmente o Command não é utilizado em linguagens que possuem programação funcional, como C#, mas ainda sim ele possui várias utilizações poderosas.Ele é útil quando queremos permitir alta configuração na funcionalidade dos comandos. Abaixo um simples editor de texto com a possibilidade de gravar macros usando command.

```
1   using System;
2   using System.Linq;
3   using System.Collections.Generic;
4
5   Invoker app = new Invoker();
6
7   while (true)
8   {
9       Console.WriteLine(app.Text);
10      string command = Console.ReadLine();
11      app.UseCommmand(command);
12      Console.Clear();
13  }
14
15  public class Invoker
16  {
17      public string Text { get; set; }
18
19      public Invoker()
20      {
21          commmandDict.Add("add", new AddCommand());
22          commmandDict.Add("rev", new ReverseCommand());
23          commmandDict.Add("sub", new SubStringCommand());
24      }
25
26      public IEnumerable<string> Commands => commmandDict.Keys.Append("macro
    ");
27
28      private Dictionary<string, ICommand> commmandDict = new Dictionary<str
    ing, ICommand>();
29      private Macro macro = null;
30      private bool macroMode = false;
31
32      public void UseCommmand(string comm)
33      {
```

```
34          comm = comm.Trim();
35          var parts = comm.Split(' ',
    StringSplitOptions.RemoveEmptyEntries);
36          if (parts.Length == 0)
37              return;
38
39          if (parts[0] == "help")
40          {
41              Console.WriteLine("Comandos Disponíveis:");
42              foreach (var com in Commands)
43                  Console.WriteLine(com);
44              Console.ReadKey(true);
45              return;
46          }
47
48          if (parts[0] == "macro")
49          {
50              macroMode = !macroMode;
51              if (macroMode)
52                  macro = new Macro(parts[1]);
53              else
54                  commmandDict.Add(macro.Name, macro);
55              return;
56          }
57
58          if (macroMode)
59          {
60              macro.Add(commmandDict[parts[0]], parts.Skip(1).ToArray());
61              return;
62          }
63
64          if (!this.commmandDict.ContainsKey(parts[0]))
65          {
66              Console.WriteLine("Commando não existe!");
67              Console.ReadKey(true);
68              return;
69          }
70
71          this.Text = commmandDict[parts[0]].Apply(this.Text,
    parts.Skip(1).ToArray());
72      }
73  }
74
75  public interface ICommand
76  {
77      string Apply(string text, string[] parameters);
78  }
79
80  public class ReverseCommand : ICommand
81  {
82      public string Apply(string text, string[] parameters)
83          => string.Concat(text.Reverse());
84  }
85
86  public class SubStringCommand : ICommand
87  {
88      public string Apply(string text, string[] parameters)
89      {
90          int size = int.Parse(parameters[0]);
91          int sizeoff = int.Parse(parameters[1]);
```

```
92
93          if (sizeoff > text.Length)
94              return string.Empty;
95
96          if (sizeoff + size >= text.Length)
97              return text.Substring(sizeoff);
98
99          return text.Substring(sizeoff, size);
100     }
101 }
102
103 public class AddCommand : ICommand
104 {
105     public string Apply(string text, string[] parameters)
106     {
107         string addedText = parameters.Aggregate("", (a, s) => a + s + " ")
    ;
108         addedText = addedText.Substring(0, addedText.Length - 1);
109         return text + addedText;
110     }
111 }
112
113 public class Macro : ICommand
114 {
115     private List<ICommand> commands = new List<ICommand>();
116     private List<string[]> parameters = new List<string[]>();
117
118     public string Name { get; set; }
119     public Macro(string name)
120         => this.Name = name;
121
122     public void Add(ICommand command, string[] parameter)
123     {
124         this.commands.Add(command);
125         this.parameters.Add(parameter);
126     }
127
128     public string Apply(string text, string[] parameters)
129     {
130         for (int i = 0; i < this.parameters.Count; i++)
131             text = commands[i].Apply(text, this.parameters[i]);
132         Console.WriteLine(text);
133         return text;
134     }
135 }
```

4.9.3 Memento

Memento é padrão comportamental com o objetivo de criar sistemas com operações de salvar/restaurar:

```
1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4
5  State state = new State();
6  Caretaker history = new Caretaker();
7
8  for (int i = 0; i < 8; i++)
9      state.Paint(i, 0, 4);
10
```

```
11  var mem = new Memento(state);
12  history.Save(mem);
13  Console.WriteLine(state);
14
15  for (int i = 0; i < 8; i++)
16      state.Paint(i, 1, 3);
17
18  mem = new Memento(state);
19  history.Save(mem);
20  Console.WriteLine(state);
21
22  for (int i = 0; i < 8; i++)
23      state.Paint(i, 2, 2);
24
25  mem = new Memento(state);
26  history.Save(mem);
27  Console.WriteLine(state);
28
29  mem = history.Undo();
30  state = mem.GetState();
31  Console.WriteLine(state);
32
33  mem = history.Undo();
34  state = mem.GetState();
35  Console.WriteLine(state);
36
37  public class Caretaker
38  {
39      Stack<Memento> stack = new Stack<Memento>();
40
41      public void Save(Memento memento)
42          => stack.Push(memento);
43
44      public Memento Undo()
45      {
46          stack.Pop();
47          return stack.Peek();
48      }
49  }
50
51  public class Memento
52  {
53      private byte[] state;
54
55      public Memento(State state)
56      {
57          var arr = state.Data;
58          var copy = new byte[arr.Length];
59          arr.CopyTo(copy, 0);
60          this.state = copy;
61      }
62
63      public State GetState()
64          => new State(this.state);
65  }
66
67  public class State
68  {
69      public byte[] Data => data;
70      private byte[] data = new byte[64];
```

```
71
72      public State() { }
73
74      public State(byte[] data)
75          => this.data = data;
76
77      public void Paint(int i, int j, byte value)
78      {
79          this.data[i + 8 * j] = value;
80      }
81
82      public override string ToString()
83      {
84          StringBuilder sb = new StringBuilder();
85
86          for (int j = 0; j < 8; j++)
87          {
88              for (int i = 0; i < 8; i++)
89              {
90                  var value = data[8 * j + i];
91
92                  if (value == 0)
93                      sb.Append(" ");
94                  else if (value == 1)
95                      sb.Append("░");
96                  else if (value == 2)
97                      sb.Append("▒");
98                  else if (value == 3)
99                      sb.Append("▓");
100                 else if (value > 3)
101                     sb.Append("█");
102             }
103             sb.AppendLine();
104         }
105
106         return sb.ToString();
107     }
108 }
```

4.9.4 Prototype

As vezes queremos criar cópias de objetos mas não sabemos exatamente como eles são. Objetos podem ter campos escondidos que desconhecemos. Por exemplo, a classe Random que é bem mais complexa por dentro do que por fora. Para isso, podemos usar o padrão prototype onde definimos um método de clonagem dentro do próprio objeto.

```
1   BoxPrototype box = new BoxPrototype("Olá, Mundo!\nOlá, Mundo!", 2);
2   box.Open();
3
4   var ohter = box.Clone() as BoxPrototype;
5   ohter.Add();
6   ohter.Open();
7
8   box.Open();
9
10  public interface IPrototype
11  {
12      IPrototype Clone();
13  }
14
```

```
15  public class BoxPrototype : IPrototype
16  {
17      private string content;
18      private int quantity;
19
20      public BoxPrototype(string content, int quantity)
21      {
22          this.content = content;
23          this.quantity = quantity;
24      }
25
26      public void Open()
27      {
28          for (int i = 0; i < quantity; i++)
29              Console.WriteLine(content);
30      }
31
32      public void Add()
33          => quantity++;
34
35      public IPrototype Clone()
36      {
37          BoxPrototype copy = new BoxPrototype(this.content, this.quantity);
38          return copy;
39      }
40  }
```

Assim podemos clonar a caixa mesmo sem ter certeza de seu conteúdo. Existe uma interface no C# para esse propósito chamada IClonable.

4.9.5 Exemplo: Draw.io multiplataforma

setup.ps1

```
1   mkdir DrawIoDesktop
2   mkdir DrawIoWeb
3   mkdir DrawIoLib
4
5   cd DrawIoLib
6   dotnet new classlib
7   cd ..
8
9   cd DrawIoDesktop
10  dotnet new winforms
11  dotnet add reference ..\DrawIoLib\DrawIoLib.csproj
12  cd ..
13
14  cd DrawIoWeb
15  dotnet new blazorserver
16  dotnet add reference ..\DrawIoLib\DrawIoLib.csproj
17  dotnet add package Blazor.Extensions.Canvas
18  cd ..
```

IVisualBehavior.cs

```
1   using System.Drawing;
2   using System.Threading.Tasks;
3
```

```
4  namespace DrawIo;
5
6  public interface IVisualBehavior
7  {
8      Task FillRectangle(RectangleF rect, Color color);
9      Task DrawRectangle(RectangleF rect, Color color);
10     Task DrawText(RectangleF rect, string text);
11     Task DrawLine(PointF p, PointF q, Color color, float width);
12     Task DrawDottedLine(PointF p, PointF q, Color color, float width);
13 }
```

IPrototype.cs

```
1  namespace DrawIo;
2
3  public interface IPrototype
4  {
5      VisualObject Clone();
6  }
```

VisualObject.cs

```
1  using System;
2  using System.Threading.Tasks;
3
4  namespace DrawIo;
5
6  public abstract class VisualObject : ICloneable, IPrototype
7  {
8      protected IVisualBehavior g;
9
10     public VisualObject(IVisualBehavior g)
11         => this.g = g;
12
13     public abstract Task Draw(bool selected);
14
15     public abstract VisualObject Clone();
16
17     object ICloneable.Clone()
18         => this.Clone();
19 }
```

ClassBox

```
1  using System.Drawing;
2  using System.Threading.Tasks;
3
4  namespace DrawIo;
5
6  public class ClassBox : VisualObject
7  {
8      public RectangleF Rectangle { get; set; }
9      public Color Color { get; set; }
10     private string text;
11
```

```
12      public ClassBox(IVisualBehavior g, PointF initial) : base(g)
13      {
14          this.text = "Classe";
15          this.Color = Color.White;
16          this.Rectangle = new RectangleF(initial, new SizeF(100, 100));
17      }
18
19      public override VisualObject Clone()
20      {
21          var crrPt = this.Rectangle.Location;
22          var newPt = new PointF(crrPt.X + 20, crrPt.Y + 20);
23
24          ClassBox box = new ClassBox(this.g, PointF.Empty);
25          box.text = this.text;
26          box.Color = this.Color;
27          box.Rectangle = new RectangleF(newPt, this.Rectangle.Size);
28
29          return box;
30      }
31
32      public override async Task Draw(bool selected)
33      {
34          await g.FillRectangle(Rectangle, Color);
35          await g.DrawRectangle(Rectangle, selected ? Color.Red :
    Color.Black);
36          await g.DrawText(Rectangle, text);
37      }
38  }
```

Arrow.cs

```
1   using System.Drawing;
2   using System.Threading.Tasks;
3
4   namespace DrawIo;
5
6   public class Arrow : VisualObject
7   {
8       private bool dotted = false;
9       private ClassBox start;
10      private ClassBox end;
11
12      public Arrow(IVisualBehavior g, ClassBox start, ClassBox end, bool
     dotted) : base(g)
13      {
14          this.start = start;
15          this.end = end;
16          this.dotted = dotted;
17      }
18
19      public override VisualObject Clone()
20      {
21          Arrow arrow = new Arrow(g, start, end, dotted);
22          return arrow;
23      }
24
25      public override async Task Draw(bool selected)
26      {
```

```
        var left = start.Rectangle.Location.X > end.Rectangle.Location.X
            ? end : start;
        var right = left == start ? end : start;

        var leftPt = new PointF(
            left.Rectangle.Location.X + left.Rectangle.Width,
            left.Rectangle.Location.Y + left.Rectangle.Height / 2
        );
        var rightPt = new PointF(
            right.Rectangle.Location.X,
            right.Rectangle.Location.Y + right.Rectangle.Height / 2
        );

        float wid = rightPt.X - leftPt.X;
        float middle = leftPt.X + wid / 2;
        var middleLeftPt = new PointF(middle, leftPt.Y);
        var middleRightPt = new PointF(middle, rightPt.Y);

        var color = selected ? Color.Red : Color.Black;

        if (dotted)
        {
            await g.DrawDottedLine(leftPt, middleLeftPt, color, 2f);
            await g.DrawDottedLine(middleLeftPt, middleRightPt, color, 2f);
            await g.DrawDottedLine(middleRightPt, rightPt, color, 2f);
        }
        else
        {
            await g.DrawLine(leftPt, middleLeftPt, color, 2f);
            await g.DrawLine(middleLeftPt, middleRightPt, color, 2f);
            await g.DrawLine(middleRightPt, rightPt, color, 2f);
        }

        if (end == left)
        {
            await g.DrawLine(leftPt, new PointF(leftPt.X + 20, leftPt.Y + 10), color, 3f);
            await g.DrawLine(leftPt, new PointF(leftPt.X + 20, leftPt.Y - 10), color, 3f);
        }
        else
        {
            await g.DrawLine(rightPt, new PointF(rightPt.X - 20, rightPt.Y + 10), color, 3f);
            await g.DrawLine(rightPt, new PointF(rightPt.X - 20, rightPt.Y - 10), color, 3f);
        }
    }
}
```

ICommand.cs

```
namespace DrawIo;

public interface ICommand
{
    void Execute(Project app);
```

```
    void Undo(Project app);
}
```

Project.cs

```
using System.Collections.Generic;
using System.Threading.Tasks;

namespace DrawIo;

public class Project
{
    public VisualObject Selected { get; set; }
    public IEnumerable<VisualObject> Objects => this.objs;
    private VisualObject copied = null;
    private List<VisualObject> objs = new List<VisualObject>();
    private Stack<ICommand> history = new Stack<ICommand>();
    private Stack<ICommand> undone = new Stack<ICommand>();

    public Project()
    {

    }

    public void Execute(ICommand command)
    {
        command.Execute(this);
        history.Push(command);
        undone.Clear();
    }

    public void Undo()
    {
        if (history.Count < 1)
            return;

        var commmand = history.Pop();
        commmand.Undo(this);
        this.undone.Push(commmand);
    }

    public void Copy()
        => copied = Selected;

    public void Cut()
    {
        copied = Selected;
        Remove(Selected);
        Selected = null;
    }

    public VisualObject Paste()
    {
        if (copied == null)
            return null;

        var copy = copied.Clone();
        this.objs.Add(copy);
```

```
54          copied = copy;
55          return copy;
56      }
57
58      public VisualObject Delete()
59      {
60          Remove(Selected);
61          var removed = Selected;
62          Selected = null;
63          return removed;
64      }
65
66      public void Remove(VisualObject obj)
67          => this.objs.Remove(obj);
68
69      public void Add(VisualObject obj)
70          => this.objs.Add(obj);
71
72      public void Redo()
73      {
74          if (undone.Count < 1)
75              return;
76
77          var commmand = undone.Pop();
78          commmand.Execute(this);
79          this.history.Push(commmand);
80      }
81
82      public async Task Draw()
83      {
84          foreach (var obj in this.objs)
85          {
86              await obj.Draw(obj == Selected);
87          }
88      }
89  }
```

AddCommand.cs

```
1   namespace DrawIo.Commands;
2
3   public class AddCommand : ICommand
4   {
5       public VisualObject Object { get; set; }
6       public void Execute(Project app)
7       {
8           app.Add(Object);
9       }
10
11      public void Undo(Project app)
12      {
13          app.Remove(Object);
14      }
15  }
```

DeleteCommand.cs

```
namespace DrawIo.Commands;

public class DeleteCommand : ICommand
{
    private VisualObject deleted = null;

    public void Execute(Project app)
    {
        this.deleted = app.Delete();
    }

    public void Undo(Project app)
    {
        if (this.deleted == null)
            return;

        app.Add(this.deleted);
    }
}
```

MoveCommand.cs

```
using System.Drawing;

namespace DrawIo.Commands;

public class MoveCommand : ICommand
{
    public ClassBox Object { get; set; }
    public PointF Old { get; set; }
    public PointF New { get; set; }

    public void Execute(Project app)
    {
        this.Old = Object.Rectangle.Location;
        Object.Rectangle = new RectangleF(New, this.Object.Rectangle.Size)
;
    }

    public void Undo(Project app)
    {
        Object.Rectangle = new RectangleF(Old, this.Object.Rectangle.Size)
;
    }
}
```

PasteCommand.cs

```
namespace DrawIo.Commands;

public class PasteCommand : ICommand
{
```

```
    private VisualObject pasted = null;

    public void Execute(Project app)
    {
        this.pasted = app.Paste();
    }

    public void Undo(Project app)
    {
        if (this.pasted == null)
            return;

        app.Remove(this.pasted);
    }
}
```

WebVisualBehavior.cs

```
 1  using System.Drawing;
 2
 3  using DrawIo;
 4  using Blazor.Extensions.Canvas;
 5  using Blazor.Extensions.Canvas.Canvas2D;
 6
 7  public class WebVisualBehavior : IVisualBehavior
 8  {
 9      private Canvas2DContext context = null;
10
11      public WebVisualBehavior(Canvas2DContext context)
12          => this.context = context;
13
14      public Task DrawDottedLine(PointF p, PointF q, Color color, float
     width)
15      {
16          throw new NotImplementedException();
17      }
18
19      public async Task DrawLine(PointF p, PointF q, Color color, float
     width)
20      {
21          await context.SetFillStyleAsync(colorToString(color));
22          await context.BeginPathAsync();
23          await context.MoveToAsync(p.X, p.Y);
24          await context.LineToAsync(q.X, q.Y);
25          await context.StrokeAsync();
26      }
27
28      public async Task DrawRectangle(RectangleF rect, Color color)
29      {
30          await context.BeginPathAsync();
31          await context.SetFillStyleAsync(colorToString(color));
32          await context.RectAsync(rect.X, rect.Y, rect.Width, rect.Height);
33          await context.StrokeAsync();
34      }
35
36      public async Task DrawText(RectangleF rect, string text)
37      {
38          await context.SetFontAsync("20px Calibri");
```

```
        await context.SetStrokeStyleAsync(colorToString(Color.Black));
        await context.StrokeTextAsync(text, rect.X, rect.Y + rect.Height / 2);
    }

    public async Task FillRectangle(RectangleF rect, Color color)
    {
        await context.SetFillStyleAsync(colorToString(color));
        await context.FillRectAsync(rect.X, rect.Y, rect.Width, rect.Height);
    }

    private string colorToString(Color color)
        => $"rgb({color.R}, {color.G}, {color.B})";
}
```

_Layout.cshtml

```
@using Microsoft.AspNetCore.Components.Web
@namespace DrawIoWeb.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <base href="~/" />
    <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
    <link href="css/site.css" rel="stylesheet" />
    <link href="DrawIoWeb.styles.css" rel="stylesheet" />
    <component type="typeof(HeadOutlet)" render-mode="ServerPrerendered" />
    <script src="_content/Blazor.Extensions.Canvas/blazor.extensions.canvas.js"></script>
</head>
<body>
    @RenderBody()

    <div id="blazor-error-ui">
        <environment include="Staging,Production">
            An error has occurred. This application may no longer respond until reloaded.
        </environment>
        <environment include="Development">
            An unhandled exception has occurred. See browser dev tools for details.
        </environment>
        <a href="" class="reload">Reload</a>
        <a class="dismiss">✕</a>
    </div>

    <script src="_framework/blazor.server.js"></script>
</body>
</html>
```

MainLayout.razor

```
@inherits LayoutComponentBase

<PageTitle>DrawIoWeb</PageTitle>

<div class="page">
    @Body
</div>
```

Index.razor

```
@page "/"
@using DrawIo;
@using System.Drawing;
@using Blazor.Extensions;
@using Blazor.Extensions.Canvas
@using Blazor.Extensions.Canvas.Canvas2D;

<BECanvas Width="600" Height="400" @ref="_canvasReference"></BECanvas>

<br/>
<button @onclick="Adicionar">Adicionar</button>

@code
{
    private Canvas2DContext _context;
    protected BECanvasComponent _canvasReference;
    private WebVisualBehavior vb;

    private Project prj;
    int elements = 0;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        this._context = await this._canvasReference.CreateCanvas2DAsync();
        vb = new WebVisualBehavior(_context);

        prj = new Project();
        ClassBox b1 = new ClassBox(vb, new PointF(50, 50));
        ClassBox b2 = new ClassBox(vb, new PointF(300, 300));

        prj.Add(b1);
        prj.Add(b2);
        prj.Add(new Arrow(vb, b1, b2, false));

        await prj.Draw();
    }

    private async void Adicionar()
    {
        elements++;
        ClassBox box = new ClassBox(vb, new PointF(50 + 20 * elements, 50
+ 20 * elements));
        prj.Add(box);
        prj.Selected = box;
```

```
44
45              await prj.Draw();
46          }
47      }
```

4.9.6 Exercícios

Faça um simples editor de texto que possa rodar em multiplas plataformas e implemente na plataforma Console. Use Bridge para reduzir o trabalho a ser feito. Use Memento para permitir que o texto possa ter alterações desfeitas e refeitas.

4.10 Aula 26 - Desafio 8

4.10.1 Introdução

Neste desafio iremos testar suas habilidades de percepção da Orientação a Objeto e apliação de Padrões de Projeto. Assim será dado uma aplicação a ser desenvolvida e posteriormente serão dadas várias alterações secretas. Você não pode ver as alterações antes que elas aconteçam e seu código deve ser flexível, desacoplado, robusto e coeso o suficiente para que as alterações sejam factíveis e não quebrem o seu sistema. Em geral não devemos exagerar nos padrões de projeto a não ser que o sistema precise de grande escalabilidade e tenha muitos motivos para que o mesmo seja alterado, contudo, neste desafio, suponha que seu sistema tenha muitas possibilidades de mudança.

4.10.2 Desafio 2: Super Auto Machines

Super Auto Pets é um jogo popular na internet onde jogadores montam seus times de bichinhos que lutam para ver quem vence. A seguir as regras de negócio de como funciona o jogo:

1. O jogo começa na fase de compras, sendo que o jogador possui 10 moedas de ouro e 5 corações de vida.
2. Na fase de compra o jogador visualiza o seu time, inicialmente vazio com 5 espaços para por suas peças. Além disso, visualiza 3 peças aleatórias no mercado, todas por 3 moedas de ouro.
3. Para comprar a peça, basta arrastá-la para uma posição vazia no seu time. A posição onde estava a peça na loja fica vazia. Não é possível comprar peças sem recurso necessário.
4. Caso o jogador deseje, pode clicar em "atualizar" pagando 1 moeda de ouro para obter uma nova loja com 3 peças aleatórias.
5. Todas as peças possuem nível, experiência, tier, ataque e vida. O nível inicial e experiência inicial de todas as peças é 1. O tier, ataque e vida depende da peça.
6. Os tiers vão de 1 a 6 e indicam o quão forte a peça é.
7. Você pode juntar duas peças iguais arrastando uma para cima da outra. Quando isso acontecer a peça resultante terá a maior das duas vidas + 1, a maior dos dois de ataques + 1 e a soma das experiências. Com 3 de experiência a peça ficará nível 2. Com 6 de experiência a peça ficará nível 3 e não poderá ser fundida com outras peças semelhantes.
8. A qualquer momento o jogador pode finalizar a etapa de compra e iniciar a batalha.
9. Na batalha o jogador joga contra um time aleatório em cada round, com 3 a 5 peças aleatórias.
10. Na batalha, continuamente, as peças mais a esquerda do jogador da direita, e mais a direta do jogador da esquerda se atacam. Ao se atacaram cada peça perde de vida o ataque da peça adversário. Peças morrem ao ficarem sem vida. Se isso acontecer a próxima peça toma seu lugar e a batalha continua até que um dos lados não tenham mais peças.
11. Se o jogador ficar sem peças e o adversário ainda tiver peças, ele perde, perdendo uma vida. Se ambos os jogadores ficarem sem peças resultará em um empate e nada ocorre. Caso contrário, vencendo, ganha um troféu.
12. Se o jogador obter 10 troféis ele ganha o jogo.
13. Se o jogador ficar sem vidas ele perde o jogo.
14. Se o jogador continuar com menos de 10 troféis e com corações disponíveis ele avança de rodada.
15. Na rodada seguinte uma nova loja será gerada, semelhante ao "atualizar" mas gratuíto e o jogador receberá 10 de gold novamente. Além disso, e mais importante, o time do jogado e restaurado da útima loja, e todas as mudanças que ocorreram na batalha são revertidas.

16. Você pode vender peças ganhando 1 de ouro por cada nível da peça.
17. O jogo possui as seguintes peças:

Implemente uma versão industrial do Super Auto Pets chamada Super Auto Machines onde cada peça é um elemento da industrial. As peças disponíveis no jogo são:

1. Tier 1
    a. Martelo: 2/3.
    b. Chave de Fenda: 2/3.
    c. Esteira: 3/1, ao vender ganha um de ouro adicional.
2. Tier 2
    a. Furadeira de Coluna: 3/5.
    b. Forno Industrial a Gás: 1/3, no inicio do turno de compra ganhe 1 de ouro,
    c. Retífica Plana: 4/2.
3. Tier 3
    a. Furadeira de Coordenada: 3/5, ao se machucar (e ficar vivo) da dano em um inimigo aleatório.
    b. Forno Industrial Elétrico: 4/3.
    c. Retífica Cilindrica: 2/6.
4. Tier 4
    a. Fresa: 4/5.
    b. Torno: 5/3, no final do turno de compra, caso você possua uma peça nível 3, recebe 2 de ataque e 2 de vida.
5. Tier 5
    a. Torno CNC: 5/8.
    b. Fresa CNC: 8/4.
6. Tier 6
    a. Corte a Plasma CNC: 6/8.

Você pode fazer a aplicação em Console, mas criamos uma aplicação base que você pode usar para ter recursos gráficos prontos. Ela vem com o desenho da carta, o efeito arrasta e solta e botões. Basta herdar da seguinte classe App:

App.cs

```
1  using System;
2  using System.Drawing;
3  using System.Windows.Forms;
4
5  public abstract class App
6  {
7      Bitmap bmp = null;
8      Graphics g = null;
9      PointF cursor = PointF.Empty;
10     PointF? grabStart = null;
11     PointF? grabDesloc = null;
12     bool isDown = false;
13     int frame = 0;
14
15     public void Run()
16     {
17         ApplicationConfiguration.Initialize();
18
19         var form = new Form();
20         form.WindowState = FormWindowState.Maximized;
21         form.FormBorderStyle = FormBorderStyle.None;
22
23         PictureBox pb = new PictureBox();
24         pb.Dock = DockStyle.Fill;
25         form.Controls.Add(pb);
```

```
26
27          Timer tm = new Timer();
28          tm.Interval = 10;
29
30          pb.MouseDown += (o, e) =>
31          {
32              isDown = true;
33          };
34
35          pb.MouseUp += (o, e) =>
36          {
37              isDown = false;
38              grabDesloc = null;
39              grabStart = null;
40          };
41
42          pb.MouseMove += (o, e) =>
43          {
44              cursor = e.Location;
45          };
46
47          form.Load += delegate
48          {
49              bmp = new Bitmap(pb.Width, pb.Height);
50              pb.Image = bmp;
51
52              g = Graphics.FromImage(bmp);
53              g.Clear(Color.White);
54              pb.Refresh();
55
56              tm.Start();
57          };
58
59          form.KeyDown += (o, e) =>
60          {
61              switch (e.KeyCode)
62              {
63                  case Keys.Escape:
64                      Application.Exit();
65                      break;
66              }
67          };
68
69          tm.Tick += delegate
70          {
71              g.Clear(Color.White);
72
73              frame++;
74              OnFrame(isDown, cursor);
75
76              pb.Refresh();
77          };
78
79          Application.Run(form);
80      }
81
82      public void DrawImage(Bitmap img, RectangleF location)
83      {
84          if (img == null)
85              throw new Exception("Imagem não pode ser nula");
```

```
86
87          g.DrawImage(img, location);
88      }
89
90      public void DrawText(string text, Color color, RectangleF location)
91      {
92          var format = new StringFormat();
93          format.Alignment = StringAlignment.Center;
94          format.LineAlignment = StringAlignment.Center;
95
96          var brush = new SolidBrush(color);
97
98          g.DrawString(text, SystemFonts.MenuFont, brush, location, format);
99      }
100
101     public RectangleF DrawPiece(RectangleF location,
102         int attack, int life, int experience, int tier,
103         bool isGraspable,
104         string name, Bitmap image = null)
105     {
106         float realWidth = .6f * location.Height;
107         var realSize = new SizeF(realWidth, location.Height);
108
109         var deslocX = grabDesloc?.X ?? 0;
110         var deslocY = grabDesloc?.Y ?? 0;
111         var position = new PointF(location.X + deslocX, location.Y +
    deslocY);
112         RectangleF rect = new RectangleF(position, realSize);
113
114         bool cursorIn = rect.Contains(cursor);
115
116         if (!cursorIn && (deslocX != 0 || deslocY != 0))
117             rect = new RectangleF(location.Location, realSize);
118
119         var pen = new Pen(cursorIn ? Color.Cyan : Color.Black, 4f);
120         var yellowPen = new Pen(Color.Yellow, 3f);
121         var whitePen = new Pen(Color.Yellow, 2f);
122
123         g.FillRectangle(Brushes.Brown, rect);
124         g.DrawRectangle(pen, rect.X, rect.Y, realWidth, rect.Height);
125
126         if (image == null)
127             DrawText(name, Color.White, rect);
128         else DrawImage(image, rect);
129
130         var attackRect = new RectangleF(rect.X, rect.Y + .8f *
    rect.Height, realWidth / 3, realWidth / 3);
131         g.FillEllipse(Brushes.Red, attackRect);
132         g.DrawEllipse(yellowPen, attackRect);
133         DrawText(attack.ToString(), Color.White, attackRect);
134
135         var lifeRect = new RectangleF(rect.X + 2 * realWidth / 3, rect.Y +
    .8f * rect.Height, realWidth / 3, realWidth / 3);
136         g.FillEllipse(Brushes.Blue, lifeRect);
137         g.DrawEllipse(yellowPen, lifeRect);
138         DrawText(life.ToString(), Color.White, lifeRect);
139
140         RectangleF expRect = new RectangleF(rect.X + realWidth / 3 - 10,
    rect.Y + 20, 2 * realWidth / 3, realWidth / 6);
141         int level = 1 + experience / 3;
```

```
142             int crrExp = experience % 3;
143             for (int i = 0; i < crrExp; i++)
144                 g.FillRectangle((Brushes.White), expRect.X + i *
    expRect.Width / 3, expRect.Y, expRect.Width / 3, expRect.Height);
145             for (int i = 0; i < 3; i++)
146                 g.DrawRectangle(whitePen, expRect.X + i * expRect.Width / 3,
    expRect.Y, expRect.Width / 3, expRect.Height);
147
148             var levelRect = new RectangleF(rect.X, rect.Y + 10, realWidth / 3,
    realWidth / 3);
149             g.FillEllipse(Brushes.Green, levelRect);
150             g.DrawEllipse(yellowPen, levelRect);
151             DrawText(level.ToString(), Color.White, levelRect);
152
153             var tierRect = new RectangleF(rect.X + realWidth / 3, rect.Y + .8f
    * rect.Height, realWidth / 3, realWidth / 3);
154             DrawText(tier.ToString(), Color.Orange, tierRect);
155
156             if (!cursorIn || !isDown)
157                 return rect;
158
159             if (grabStart == null)
160             {
161                 grabStart = cursor;
162                 return rect;
163             }
164
165             grabDesloc = new PointF(cursor.X - grabStart.Value.X, cursor.Y -
    grabStart.Value.Y);
166
167             return rect;
168         }
169
170         public bool DrawButton(RectangleF location, string text)
171         {
172             if (location.Contains(cursor))
173             {
174                 if (isDown)
175                 {
176                     g.FillRectangle(Brushes.Red, location);
177                     DrawText(text, Color.Black, location);
178                     return true;
179                 }
180                 else
181                 {
182                     g.FillRectangle(Brushes.BlueViolet, location);
183                     DrawText(text, Color.White, location);
184                     return false;
185                 }
186             }
187             else
188             {
189                 g.FillRectangle(Brushes.Blue, location);
190                 DrawText(text, Color.White, location);
191                 return false;
192             }
193         }
194
195         public abstract void OnFrame(bool IsDown, PointF cursor);
```

```
196 }
```

Você não precisa compreender se não quiser, apensar copiar e utilizar. Aqui estão um App de exemplo e como usar ela num projeto windows forms no seu Program.cs:

Program.cs

```
1 App app = new ExampleApp();
2 app.Run();
```

ExampleApp.cs

```
1  using System.Drawing;
2  using System.Windows.Forms;
3
4  public class ExampleApp : App
5  {
6      bool fundiu = false;
7      bool clicked = false;
8      RectangleF rect1 = RectangleF.Empty;
9      RectangleF rect2 = RectangleF.Empty;
10     public override void OnFrame(bool isDown, PointF cursor)
11     {
12         if (rect1.Contains(cursor) && rect2.Contains(cursor) && !isDown)
13             fundiu = true;
14
15         if (!fundiu)
16         {
17             rect1 = DrawPiece(new RectangleF(50, 50, 200, 200), 1, 3, 1,
   1, true, "CNC");
18             rect2 = DrawPiece(new RectangleF(300, 50, 200, 200), 2, 4, 2,
   1, true, "CNC");
19         }
20         else
21         {
22             DrawPiece(new RectangleF(50, 50, 200, 200), 3, 5, 3, 1, true,
   "CNC");
23         }
24
25         if (!clicked)
26         {
27             clicked = DrawButton(new RectangleF(400, 400, 200, 100),
   "Iniciar");
28             if (clicked)
29                 MessageBox.Show("Clicou");
30         }
31     }
32 }
```

4.10.3 Atualização Secreta 1

1. No round 3, caso o jogador tenha perdido vida, ele recupera um único coração.
2. Atualização nas peças:
    a. Materlo: Ao vender da mais um de vida para as peças da loja.
    b. Chave de Fenda: Ao vender da mais um de vida para uma peça aleatória do time.
    c. Furadeira: Tier 1, 2/1, ao morrer da +2/+1 para um aliado aleatório.

d. Jet-cutting: Tier 3, 6/3.

4.10.4 Atualização Secreta 2

1. A partir desta atualização, no inicio do jogo só são disponíveis peças do tier 1. Em todos os rounds impares (3, 5, 7, 9 e 11) será liberado um novo tier na loja.
2. Atualização nas peças:
    a. Furadeira de Coluna: Ao atacar causa 1 de dano ao aliado atrás dele.
    b. Retífica Cilindrica: Ao se machucar (e ficar vivo) da +1/+1 ao aliado atrás.
    c. Retífica Plana: Ao morrer concede +1/+1 a dois aliados atrás

4.10.5 Atualização Secreta 3

1. O jogo agora contará com 2 upgrades na sua loja que atualiza junto com os pets.
2. Atualização nos upgrades:
    a. Óleo: 3 de ouro, +1/+1 a uma peça a escolha.
    b. Reciclgem: 1 de ouro, mata uma peça a sua escolha.
    c. Botão de segurança: 3 de ouro, uma peça a sua escolha ganha o seguinte efeito: Ao sofrer dano, recebe 2 de dano a menos, mas no mínimo 1.
    d. Sensores de segurança: 3 de ouro, uma peça a sua escolha ganha o seguinte efeito: O próximo dano sofrido é ignorado.
3. Os upgrades que dão efeitos não podem ser combados e um substitui o outro.
4. Quaisquer efeitos concedidos durante a fase de compra são permanentes, ou seja, se uma Retífica Plana recebe Reciclagem na fase de compra ela deve dar seu buff para seus alidos e esses buffs devem ser permanentes.

4.10.6 Atualização Secreta 4

1. Ao evoluir uma peça para um nível acima você ganha na sua loja uma peça de um tier superior aleatório. Exemplo, se está no turno 3 e pode comprar peças de tier 1 e 2, se evoluir uma peça qualquer, você recebe uma peça tier 3 na sua loja.
2. Os efeitos de todas as peças melhoram quando elas evoluim de nível. (Em geral, basta dobrar ou repetir no nível 2 e triplicar no nível 3).
3. Atualização nas peças:
    a. Forno Industrial Elétrico: No inicio da batalha causa 3 de dano ao inimigo com a menor vida.
    b. Fresa: Ao matar um inimigo ganha +3/+3.
    c. Jet-cutting: 1/3, se o aliado da frente morrer, recebe +1/+0 por nível e Sensores de Segurança.
    d. Torno CNC: Ao matar um inimigo, causa 4 vezes o nível de dano ao próximo inimigo.
    e. Fresa CNC: No inicio do combate, causa 8 de dano, uma vez por nível, no inimigo mais atrás do adversáro.

4.10.7 Atualização Secreta 5

1. Atualização nas peças:
    a. Impressora 3D: Tier 4, 4/2, No final do turno de compra copia a habilidade do aliado a sua frente como se ele estivesse no mesmo nível da impressora 3D.
    b. Forno Industrial Combinado: Tier 5, 4/6, Ao ser comprado substitui os upgrades da loja por 2 engrenagens que dão +1/+1 por nível a um aliado qualquer custando 0 de ouro.
    c. Jet-cutting CNC: Tier 6, 5/5, Se um aliado morrer em batalha, invoca uma mini Jet-cutting CNC 5/5 vezes o nível. Acontece até 3 vezes.
    d. EDM: Tier 6, 4/3, Faz com que o aliado da frente repita sua habilidade como se estivesse no nível da EDM.

4.11 Aula 27 - Programação Genérica e Reflexiva, Expressões e Atributos

- Restrições Genéricas
- Variância Genérica

- Reflexão
- Reflexão Genérica
- Expressions
- Atributos
- Exemplo: Removendo Prints de Objetos específicos
- Exercícios

4.11.1 Restrições Genéricas

Em C# básico foram mostrados os tipos genéricos e suas capacidades. Agora vamos ir um pouco mais afundo nas capacidades desta feature do C#. Primeiramente, considere o seguinte código:

```
1  public class A
2  {
3
4  }
5
6  public class A1 : A
7  {
8
9  }
10
11 public class A2 : A
12 {
13
14
15 }
16
17 public class B<T>
18     where T : A
19 {
20     public T Result { get; set; }
21 }
```

A classe genérica B contém agora uma restrição genérica definido com a palavra 'where'. Ela garante que o tipo T será/herdará de A. Assim, B<A>, B<A1> e B<A2> são tipos possíveis para B, enquanto isso, B<int> resultará em um erro. Além de um tipo você pode especificar vários tipos de outras restrições:

```
// C1<int> c1; Error
C1<C3<int>> c2; // Ok
C4<int> c3; // Ok
C5<int> c4; // Ok
C8<MemoryStream, Stream> c5; // Ok

public class C1<T> where T : class { }
public class C2<T> where T : class? { }
public class C3<T> where T : struct { }
public class C4<T> where T : notnull { } // Não deve ser anulável
public class C5<T> where T : unmanaged { } // Deve ser um tipo gerenciável, structs
puras
public class C6<T, U> where T : U { }
public class C7<T> where T : new()
{
    public T Get()
    {
        return new T();
    }
}
public class C8<T, U> where T : U, new() where U : class
{
    public U Value { get; set; } = new T();
```

```
}
```

Logo veremos alguns exemplos interessantes e complexos onde podemos aplicar restrições genéricas.

4.11.2 Variância Genérica

Sabemos que podemos colocar um objeto em uma variável do tipo da classe mãe. Isso é comum em Orientação a Objetos. Mas e com genéricos? Poderíamos por uma lista de int em uma variável de coleção de objetos? Afinal Lista é uma coleção e int é um objeto. Na verdade isso funciona em partes. A conversão entre object e int não é tão direta. Isso se dá graças ao boxing do int, mas de outros tipos de referência isso é totalmente possível. Observe:

```
object obj = 8; // Variância comum

var cobj = new C<Funcionario>();

A<Funcionario> a1 = cobj; // Esperado
B<Funcionario> b1 = cobj; // Esperado

B<Pessoa> b2 = cobj; // Covariância genérica
A<Aprendiz> a2 = cobj; // Contravariância genérica

a2.Value = new Aprendiz() { Nome = "Xipsita" };
// a2.Value = new Mae(); // Erro: a2 espera objeto do tipo filha
// Avo avo = a2.Value; // Erro: a2 não possui Get
Pessoa avo = b2.Value;
Console.WriteLine(avo.Nome);

var list = new List<Funcionario>();
IEnumerable<Pessoa> collection = list;

Func<Funcionario, Funcionario> func = x => x;
Func<Aprendiz, Pessoa> func2 = func;
Pessoa res = func2(new Aprendiz());

public class Pessoa
{
    public string Nome { get; set; }
}
public class Funcionario : Pessoa { }
public class Aprendiz : Funcionario { }

// in e out válidos a interfaces e delegados
public interface A<in T> // T só pode ser usado em parâmetros
{
    // T Get(); Erro
    T Value { set; }
}

public interface B<out T> // T só pode ser usado em retornos
{
    // void Set(T value); Erro
    T Value { get; }
}

public class C<T> : A<T>, B<T>
{
    public T Value { get; set; }
}
```

4.11.3 Reflexão

Reflexão é a capacidade das linguagens de programação de ler sua própria estrutura e até modificá-la. No C# a programação reflexiva é possível e extremamente poderosa.

```
1  using System.Reflection; // Necessário apenas para o GetRuntimeFields
2
3  var type = typeof(Funcionario);
4
5  Console.WriteLine($"Class {type.Name}:");
6  foreach (var prop in type.GetMembers()) // Obtém membros publicos!
7  {
8      Console.WriteLine($"\t{prop.MemberType} {prop.Name} :
   {prop.DeclaringType}");
9  }
10
11 foreach (var field in type.GetRuntimeFields()) // Obtém campos privados!
12 {
13     Console.WriteLine($"\t{field.MemberType} {field.Name} :
   {field.DeclaringType}");
14 }
15
16 Console.WriteLine();
17 Funcionario funcionario = new Funcionario("Xispita", "12345678", 10);
18 var nameProp = type.GetProperty("Nome");
19 var name = nameProp.GetValue(funcionario) as string;
20 nameProp.SetValue(funcionario, name + "!");
21 Console.WriteLine(funcionario.Nome);
22
23 var calcMethod = type.GetMethod("CalcularSalario");
24 var wage = calcMethod.Invoke(funcionario, new object[] { 200 });
25 Console.WriteLine(wage);
26
27 Console.WriteLine();
28 var assembly = Assembly.GetExecutingAssembly();
29 foreach (var x in assembly.GetTypes())
30     Console.WriteLine(x.Name);
31
32 /**
33 Saída:
34 Class Funcionario:
35         Method get_Nome : Funcionario
36         Method set_Nome : Funcionario
37         Method get_EDV : Funcionario
38         Method set_EDV : Funcionario
39         Method get_SalarioHora : Funcionario
40         Method CalcularSalario : Funcionario
41         Method GetType : System.Object
42         Method ToString : System.Object
43         Method Equals : System.Object
44         Method GetHashCode : System.Object
45         Constructor .ctor : Funcionario
46         Property Nome : Funcionario
47         Property EDV : Funcionario
48         Property SalarioHora : Funcionario
49         Field salarioHora : Funcionario
50         Field <Nome>k__BackingField : Funcionario
51         Field <EDV>k__BackingField : Funcionario
52
53 Xispita!
```

```
2000

EmbeddedAttribute
NullableAttribute
NullableContextAttribute
Program
Funcionario
 **/

public class Funcionario
{
    private double salarioHora;

    public string Nome { get; set; }
    public string EDV { get; set; }
    public double SalarioHora => salarioHora;

    public Funcionario(string nome, string edv, double salarioHora)
    {
        this.Nome = nome;
        this.EDV = edv;
        this.salarioHora = salarioHora;
    }

    public double CalcularSalario(double horas)
        => horas * salarioHora;
}
```

4.11.4 Reflexão Genérica

Assim como podemos fazer reflexão sobre um tipo qualquer, podemos fazer reflexão sobre um tipo que não conhecemos:

```

Creator<int> creator = new Creator<int>();
int i = creator.Create();
int j = creator.Create();

public class Creator<T>
{
    public T Create(params object[] arr)
    {
        var type = typeof(T);
        foreach (var constructor in type.GetConstructors())
        {
            var parameters = constructor.GetParameters();
            if (parameters.Length == arr.Length)
                return (T)constructor.Invoke(arr);
        }
        return default(T);
    }
}
```

4.11.5 Expressions

Expressões são outra forma do C# de compreender o próprio código. Com elas o C# é capaz de ler uma função Lambda e gerar uma árvore de expressão e reconhecer sua estrutura.

```

using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
```

```
5
6   Expression<Func<float, float>> exp = x => MathF.Sqrt(x * x + 3) +
    MathF.Sin(MathF.PI * x) - 2;
7   analyzeNode(exp.Body);
8
9   /**
10      ((Sqrt(((x * x) + 3)) + Sin((3,1415927 * x))) - 2)
11      ├(Sqrt(((x * x) + 3)) + Sin((3,1415927 * x)))
12      │├Sqrt(((x * x) + 3))
13      ││└((x * x) + 3)
14      ||  ├(x * x)
15      ||  │├x
16      ||  │└x
17      ||  └3
18      │└Sin((3,1415927 * x))
19      |  └(3,1415927 * x)
20      |   ├3,1415927
21      |   └x
22      └2
23   **/
24
25  void analyzeNode(Expression exp)
26  {
27      Stack<string> stack = new Stack<string>();
28      _analyzeNode(exp);
29
30      void _analyzeNode(Expression exp, string newStr = null)
31      {
32          if (newStr != null)
33              stack.Push(newStr);
34
35          var levelInfo = string.Concat(stack.Reverse());
36          if (levelInfo.Length > 0)
37              levelInfo =
38                  levelInfo.Substring(0, levelInfo.Length - 1) +
39                  (levelInfo.Last() == ' ' ? '└' : '├');
40
41          switch (exp)
42          {
43              case MethodCallExpression call:
44                  Console.Write(levelInfo);
45                  Console.WriteLine(call);
46
47                  var args = call.Arguments;
48                  for (int i = 0; i < args.Count; i++)
49                      _analyzeNode(args[i], i == args.Count - 1 ? " " : "|")
    ;
50                  break;
51
52              case BinaryExpression bin:
53                  Console.Write(levelInfo);
54                  Console.WriteLine(bin);
55
56                  _analyzeNode(bin.Left, "|");
57                  _analyzeNode(bin.Right," ");
58                  break;
59
60              case ConstantExpression con:
61                  Console.Write(levelInfo);
62                  Console.WriteLine(con);
```

```
                break;

            case ParameterExpression par:
                Console.Write(levelInfo);
                Console.WriteLine(par);
                break;

            default:
                Console.WriteLine(exp.GetType());
                break;
        }

        if (newStr != null)
            stack.Pop();
    }
}
```

Expressions podem ser usadas para muitas coisas, em especial, representar código em diversas plataformas, visto que, é possível traduzir código C# para outras linguagens mais facilmente. Podemos criar nossas funções em tempo de execução, compila-las e executa-las:

```
using System;
using System.Linq.Expressions;

var parameter = Expression.Parameter(typeof(float));                    // x
var sqrt = typeof(MathF).GetMethod("Sqrt");                             // Sqrt
var exp = Expression.Call(sqrt, parameter);                             // Sqrt(x)
var lambda = Expression.Lambda<Func<float, float>>(exp, parameter);     // x => Sqrt(x)
var f = lambda.Compile();
var result = f(4);
Console.WriteLine(result);
```

As possibilidades são infinitas e você pode escrever qualquer função usando isso:

```
using System;
using System.Linq.Expressions;
using static System.Linq.Expressions.Expression;

int[] data = new int[] { 5, 2, 1, 3, 4 };

// int low = int.MaxValue;
// int value = 0;
// for (int i = 0; i < arr.Length; i++)
// {
//      value = arr[i];
//      if (value < low)
//          low = value;
// }

var arr = Parameter(typeof(int[]), "arr");
var len = typeof(int[]).GetProperty("Length");
var print = typeof(Console).GetMethod("WriteLine", 0, new Type[] { typeof(int) });
var low = Parameter(typeof(int), "low");
var value = Parameter(typeof(int), "value");
var i = Parameter(typeof(int), "i");
```

```
var label = Label(typeof(int));
var block = Block(
    new[] { low, i, value },
    Assign(low, Constant(int.MaxValue)),
    Assign(i, Constant(0)),
    Loop(
        Block(
            Assign(value, ArrayAccess(arr, i)),
            IfThen(LessThan(value, low),
                Assign(low, value)
            ),
            Assign(i, Add(i, Constant(1))),
            IfThen(GreaterThanOrEqual(i, Property(arr, len)),
                Break(label, low)
            )
        ), label
    )
);

var lambda = Expression.Lambda<Func<int[], int>>(block, arr);
var f = lambda.Compile();
var result = f(data);
Console.WriteLine(result);
```

Ainda sim, você não estará apto a modificar métodos aos quais você não criou inicialmente como uma Expression.

4.11.6 Atributos

Atributos são modificadores que podem ser criados e lidos com reflection. Eles são úteis nas mais diversas situações e é muito comum que entremos em contato com eles antes mesmo deles serem compreensíveis a nós:

```
using System;
using System.Reflection;

foreach (var type in Assembly.GetExecutingAssembly().GetTypes())
{
    var att = type.GetCustomAttribute<MyAttribute>();

    if (att == null)
        continue;

    if (att.Data < 15)
        continue;

    Console.WriteLine(type.Name);
}
// Output: B


public class MyAttribute : Attribute
{
    public int Data { get; set; }

    public MyAttribute(int data)
        => this.Data = data;
}

[MyAttribute(10)]
public class A { }
```

```
29
30  [My(20)] // A palavra Attribute pode ser omotida
31  public class B { }
32
33  public class C { }
```

4.11.7 Exemplo: Removendo Prints de Objetos específicos

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq.Expressions;
4   using System.Reflection;
5   using static System.Linq.Expressions.Expression;
6
7   A[] arrA = new[] { new A(), new A(), new A(), new A() };
8   B[] arrB = new[] { new B(), new B(), new B(), new B() };
9
10  arrA.Act(x => Console.WriteLine(x));
11  arrB.Act(x => Console.WriteLine(x));
12  arrB.Act(x => Console.WriteLine("Xispita"));
13
14  public static class ShowData
15  {
16      public static void Act<T>(this IEnumerable<T> coll,
    Expression<Action<T>> action)
17          where T : InfoElement
18      {
19          var type = typeof(T);
20          var att = type.GetCustomAttribute<NotPrintableAttribute>();
21          if (att == null)
22          {
23              act(coll, action.Compile());
24              return;
25          }
26
27          action = removePrint(action);
28          act(coll, action.Compile());
29      }
30
31      private static void act<T>(IEnumerable<T> coll, Action<T> action)
32          where T : InfoElement
33      {
34          var it = coll.GetEnumerator();
35          while (it.MoveNext())
36              action(it.Current);
37      }
38
39      private static Expression<Action<T>>
    removePrint<T>(Expression<Action<T>> action)
40          where T : InfoElement
41      {
42          switch (action.Body)
43          {
44              case MethodCallExpression call:
45                  bool isWrite = call.Method.Name == "WriteLine" ||
46                      call.Method.Name == "Write";
47                  bool isConsole = call.Method.DeclaringType == typeof(Conso
    le);
48                  bool isPrint = isConsole && isWrite;
```

```
                if (!isPrint)
                    return action;

                var obj = call.Arguments.Count > 0 ? call.Arguments[0] :
null;
                bool isNotPrintable =
obj.Type.GetCustomAttribute<NotPrintableAttribute>() != null;

                if (!isNotPrintable)
                    return action;

                var t = Parameter(typeof(T));
                return Lambda<Action<T>>(Empty(), t);

            default:
                return action;
        }
    }
}

public class NotPrintableAttribute : Attribute { }

public class InfoElement
{
    public override string ToString()
        => "Info: " + getInfo();

    protected virtual string getInfo()
        => null;
}

public class A : InfoElement
{
    protected override string getInfo()
        => "Olá Mundo!";
}

[NotPrintable]
public class B : InfoElement
{
    protected override string getInfo()
        => "Hello World!";
}
```

4.11.8 Exercícios

1. Printe todas as classes de um projeto que possuem métodos com um atributo chamado 'ImportantAttribute'.
2. Crie um método de extensão chamado Copy. Este método estende um tipo T qualquer com construtor vazio e copia suas propriedades e campos privados para um novo objeto, copiando-o.
3. Construa um conversor de código C# para Python usando Expressions. Considere que não existem vetores e que todas as variáveis são do tipo int.
4. Faça uma função que conte quantas vezes uma variável de entrada é usada numa função lambda recebida.

4.12 Aula 28 - Conexão com Banco de Dados, Object-Relational Mapping e Entity Framework

4.12.1 Conectando o C# ao SQLServer

Existem várias maneiras de conectar o C# ao banco de dados, vamos a mais clássica delas. Inicialmente, podemos executar o seguinte script que cria um novo projeto e instá-la uma biblioteca para tal. Essa biblioteca é antiga e antigamente vinha junto com o .NET. Hoje em dia, em versões novas, é necessário usar o nuget para obtê-la:

```
mkdir BancoExemplo
cd BancoExemplo
dotnet new console
dotnet add package System.Data.SqlClient
```

Agora vamos a utilização dela. Inicialmente não criaremos nenhuma classe, vamos ao seu uso direto. Abaixo o script do SQl que usaremos:

```
use master
go

if exists(select * from sys.databases where name = 'example')
    drop database example

create database example
go

use example
go

create table Cliente(
    ID int identity primary key,
    Nome varchar(100) not null,
    Senha varchar(100) not null,
    DataNasc date not null
);
go
```

```
using System.Data.SqlClient;

SqlConnectionStringBuilder stringConnectionBuilder = new
 SqlConnectionStringBuilder();
stringConnectionBuilder.DataSource = @"JVLPC0480\SQLEXPRESS"; // Nome do
servidor
stringConnectionBuilder.InitialCatalog = "example"; // Nome do banco
stringConnectionBuilder.IntegratedSecurity = true;
string stringConnection = stringConnectionBuilder.ConnectionString;

SqlConnection conn = new SqlConnection(stringConnection);
conn.Open();

```

```
12  SqlCommand comm = new SqlCommand("insert Cliente values ('Pamella', '123',
    CONVERT(DATETIME, '03/27/2023'));");
13  comm.Connection = conn;
14  comm.ExecuteNonQuery();
15
16  conn.Close();
```

Ao executarmos um select podemos ver que o cliente foi criado com sucesso. Podemos usar a mesma ideai junto de um DataTable para ler dados.

```
1   using System;
2   using System.Data;
3   using System.Data.SqlClient;
4
5   SqlConnectionStringBuilder stringConnectionBuilder = new
     SqlConnectionStringBuilder();
6   stringConnectionBuilder.DataSource = @"JVLPC0480\SQLEXPRESS";
7   stringConnectionBuilder.InitialCatalog = "example";
8   stringConnectionBuilder.IntegratedSecurity = true;
9   string stringConnection = stringConnectionBuilder.ConnectionString;
10
11  SqlConnection conn = new SqlConnection(stringConnection);
12  conn.Open();
13
14  string nome = Console.ReadLine();
15  string senha = Console.ReadLine();
16
17  SqlCommand comm = new SqlCommand($"select * from Cliente where Nome =
    '{nome}' and Senha = '{senha}'");
18  comm.Connection = conn;
19  var reader = comm.ExecuteReader();
20
21  DataTable dt = new DataTable();
22  dt.Load(reader);
23
24  if (dt.Rows.Count > 0)
25      Console.WriteLine($"Usuário {dt.Rows[0].ItemArray[0]} Logado");
26  else
27      Console.WriteLine("Conta inexistente");
28
29  conn.Close();
```

Ao executarmos podemos fazer uma espécie de Login.

4.12.2 Sql Injection

Temos que tomar cuidado com alguns tipos de abordagens ao usarmos bibliotecas de acesso ao banco de dados. Uma delas é o Sql Injection. No Sql Injection, uma pessoa mal intencionada pode inserir SQL em nossa query fechando as aspas da linha 17 e colocando código malicioso.

```
1   dotnet run
2   ' or 1 = 1 --
3
4   Usuário 3 Logado
```

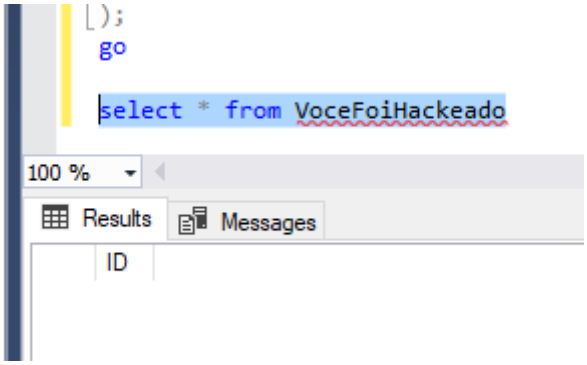
Explicando, ao digitarmos aquele código na primeira linha fazemos a seguinte query SQL na linha 17:

```
select * from Cliente where Nome = '' or 1 = 1 --' and Senha = '{senha}'
```

Basicamente ele selecionará todos os clientes onde o nome for vazio ou 1 for igual a 1, o que é sempre verdadeiro. O resto da query será ignorada como temos um comentário. Assim o hacker invade o sistema. Mas isso não é tudo de ruim que pode ser feito. Observe:

```
dotnet run
'; create table VoceFoiHackeado ( ID int ); --

Conta inexistente
```



Além disso podemos excluir dados também ou qualquer coisa que você imaginar:

```
 dotnet run
'; delete Cliente; --

Conta inexistente
```

Por isso podemos melhorar o uso do nosso sistema para evitar tais debilidades. O código abaixo bloqueia várias possibilidades de se afetar o software com sql injection.

```
1  using System;
2  using System.Data;
3  using System.Data.SqlClient;
4
5  SqlConnectionStringBuilder stringConnectionBuilder = new
    SqlConnectionStringBuilder();
6  stringConnectionBuilder.DataSource = @"JVLPC0480\SQLEXPRESS";
7  stringConnectionBuilder.InitialCatalog = "example";
8  stringConnectionBuilder.IntegratedSecurity = true;
9  string stringConnection = stringConnectionBuilder.ConnectionString;
10
11 SqlConnection conn = new SqlConnection(stringConnection);
12 conn.Open();
13
14 string nome = Console.ReadLine();
15 string senha = Console.ReadLine();
16
17 SqlCommand comm = new SqlCommand($"select * from Cliente where Nome =
   @Nome and Senha = @Senha");
18 comm.Connection = conn;
19
20 comm.Parameters.Add(new SqlParameter("@Nome", nome));
21 comm.Parameters.Add(new SqlParameter("@Senha", senha));
22
23 var reader = comm.ExecuteReader();
24
25 comm.Parameters.Clear(); // Limpar depois de usar permite reutilizar o
   SqlCommand
26
27 DataTable dt = new DataTable();
28 dt.Load(reader);
```

```
29
30  if (dt.Rows.Count > 0)
31      Console.WriteLine($"Usuário {dt.Rows[0].ItemArray[0]} Logado");
32  else
33      Console.WriteLine("Conta inexistente");
34
35  conn.Close();
```

4.12.3 Exemplo: Criando um ORM

ORMLib/DataAnnotations/ForeignKeyAttribute.cs

```
1   using System;
2
3   namespace ORMLib.DataAnnotations;
4
5   // No C# 11.0 em diante poderemos usar Atributos Genéricos
6   public class ForeignKeyAttribute : Attribute
7   {
8       public Type ForeignTable { get; set; }
9       public ForeignKeyAttribute(Type foreignTable)
10          => this.ForeignTable = foreignTable;
11  }
```

ORMLib/DataAnnotations/NotNullAttribute.cs

```
1   using System;
2
3   namespace ORMLib.DataAnnotations;
4
5   public class NotNullAttribute : Attribute { }
```

ORMLib/DataAnnotations/PrimaryKeyAttribute.cs

```
1   using System;
2
3   namespace ORMLib.DataAnnotations;
4
5   public class PrimaryKeyAttribute : Attribute { }
```

ORMLib/Exceptions/ConfigNotInitializedException.cs

```
1   using System;
2
3   namespace ORMLib.Exceptions;
4
5   public class ConfigNotInitializedException : Exception
6   {
7       public override string Message => "ORM config is not initializated";
8   }
```

ORMLib/Exceptions/InvalidColumnType.cs

```cs
using System;

namespace ORMLib.Exceptions;

public class InvalidColumnType : Exception
{
    string type;

    public InvalidColumnType(Type type)
        => this.type = type.Name;

    public override string Message => $"O tipo {type} não é valido para uma coluna no banco de dados.";
}
```

ORMLib/ObjectRelationalMappingConfig.cs

```cs
#pragma warning disable CS1998

using System.Threading.Tasks;

namespace ORMLib;

using Exceptions;
using Providers;

public class ObjectRelationalMappingConfig
{
    private static ObjectRelationalMappingConfig config = null;
    public static ObjectRelationalMappingConfig Config
    {
        get
        {
            if (config == null)
                throw new ConfigNotInitializedException();

            return config;
        }
    }

    public static ObjectRelationalMappingConfigBuilder GetBuilder()
        => new ObjectRelationalMappingConfigBuilder();

    public virtual DataBaseSystem DataBaseSystem { get; set; }
    public virtual string StringConnection { get; set; }
    public virtual string InitialCatalog { get; set; }
    public IQueryProvider QueryProvider { get; set; }
    public AccessProvider AccessProvider { get; set; }

    public ObjectRelationalMappingConfig(
        DataBaseSystem dataBaseSystem,
        string stringConnection,
        string initialCatalog,
        IQueryProvider queryProvider,
```

```
        AccessProvider accessProvider
    )
    {
        this.DataBaseSystem = dataBaseSystem;
        this.StringConnection = stringConnection;
        this.InitialCatalog = initialCatalog;
        this.QueryProvider = queryProvider;
        this.AccessProvider = accessProvider;
    }

    public void Use()
        => config = this;

    public virtual async Task UseAsync()
        => Use();
}
```

ORMLib/ObjectRelationalMappingConfigBuilder.cs

```
using System.Data.SqlClient;

namespace ORMLib;

using Providers;

public class ObjectRelationalMappingConfigBuilder
{
    private DataBaseSystem dataBaseSystem;
    private SqlConnectionStringBuilder stringConnectionBuilder = new SqlConnectionStringBuilder();
    private IQueryProvider queryProvider;
    private AccessProvider accessProvider;

    public ObjectRelationalMappingConfigBuilder SetDataBaseSystem(DataBaseSystem sys)
    {
        this.dataBaseSystem = sys;
        return this;
    }

    public ObjectRelationalMappingConfigBuilder SetDataSource(string serverName)
    {
        stringConnectionBuilder.DataSource = serverName;
        return this;
    }

    public ObjectRelationalMappingConfigBuilder SetInitialCatalog(string initialCatalog)
    {
        stringConnectionBuilder.InitialCatalog = initialCatalog;
        return this;
    }

    public ObjectRelationalMappingConfigBuilder SetIntegratedSecurity(bool integratedSecurity)
    {
        stringConnectionBuilder.IntegratedSecurity = integratedSecurity;
```

```
35          return this;
36      }
37
38      public ObjectRelationalMappingConfigBuilder SetStringConnection(string
     strConn)
39      {
40          stringConnectionBuilder.ConnectionString = strConn;
41          return this;
42      }
43
44      public ObjectRelationalMappingConfigBuilder
    SetQueryProvider(IQueryProvider provider)
45      {
46          this.queryProvider = provider;
47          return this;
48      }
49
50      public ObjectRelationalMappingConfigBuilder
    SetAccessProvider(AccessProvider provider)
51      {
52          this.accessProvider = provider;
53          return this;
54      }
55
56      public ObjectRelationalMappingConfig Build()
57      {
58          ObjectRelationalMappingConfig config = new
     ObjectRelationalMappingConfig(
59              this.dataBaseSystem,
60              this.stringConnectionBuilder.ConnectionString,
61              this.stringConnectionBuilder.InitialCatalog,
62              this.queryProvider,
63              this.accessProvider
64          );
65          return config;
66      }
67  }
```

ORMLib/ObjectRelationalMappingConfigBuilderExtension.cs

```
1   namespace ORMLib;
2
3   using MSSql;
4
5   public static class ObjectRelationalMappingConfigBuilderExtension
6   {
7       public static ObjectRelationalMappingConfigBuilder UseMSSqlServer(this
     ObjectRelationalMappingConfigBuilder builder)
8       {
9           builder.SetDataBaseSystem(DataBaseSystem.SqlServer);
10          builder.SetQueryProvider(new MSSqlQueryProvider());
11          builder.SetAccessProvider(new MSSqlProvider());
12          return builder;
13      }
14  }
```

ORMLib/DataBaseSystem.cs

```
namespace ORMLib;

public enum DataBaseSystem
{
    SqlServer,
    Oracle,
    MariaDB,
    MySql
}
```

ORMLib/Access.cs

```
using System.Threading.Tasks;

namespace ORMLib;

public abstract class Access
{
    private static Access instance = null;
    public static Access Instance
    {
        get
        {
            if (instance is not null)
                return instance;

            var provider =
ObjectRelationalMappingConfig.Config.AccessProvider;
            instance = provider.Provide();

            return instance;
        }
    }

    public abstract Task Insert<T>(T obj);
    public abstract Task Delete<T>(T obj);
    public abstract Task Update<T>(T obj);
}
```

ORMLib/Table.cs

```
using System.Threading.Tasks;
using System.Linq.Expressions;

namespace ORMLib;

using Linq;

public abstract class Table<T>
    where T : class, new()
{
    private bool exist = false;

```

```
    public async Task Save()
    {
        var obj = this as T;

        if (this.exist)
            await Access.Instance.Update(obj);
        else await Access.Instance.Insert(obj);

        this.exist = true;
    }

    public async Task Delete()
    {
        var obj = this as T;
        await Access.Instance.Delete(obj);
        this.exist = false;
    }

    public static IQueryable<T> All
    {
        get
        {
            var provider =
ObjectRelationalMappingConfig.Config.QueryProvider;
            var empty = provider.CreateQuery<T>(null);
            return provider.CreateQuery<T>(Expression.Constant(empty));
        }
    }
}
```

ORMLib/MSSql/MSSqlProvider.cs

```
namespace ORMLib.MSSql;

using Providers;

public class MSSqlProvider : AccessProvider
{
    public override Access Provide()
        => new SqlAccess();
}
```

ORMLib/MSSql/SqlAccess.cs

```
using System;
using System.Linq;
using System.Data;
using System.Reflection;
using System.Data.SqlClient;
using System.Threading.Tasks;
using System.Collections.Generic;

namespace ORMLib.MSSql;

using Exceptions;
using DataAnnotations;
```

```
13  using System.Collections;
14
15  internal class SqlAccess : Access
16  {
17      private SqlCommand comm;
18      private SqlConnection conn;
19      private bool loaded = false;
20      private bool exist = false;
21
22      public async Task CreateDataBaseIfNotExistAsync()
23      {
24          if (exist)
25              return;
26          exist = true;
27
28          var config = ObjectRelationalMappingConfig.Config;
29          var masterStrConn = config.StringConnection.Replace(
30              config.InitialCatalog,
31              "master"
32          );
33          var conn = new SqlConnection(masterStrConn);
34          await conn.OpenAsync();
35
36          var comm = new SqlCommand();
37          comm.CommandText = $"select * from sys.databases where name =
    '{config.InitialCatalog}'";
38          comm.Connection = conn;
39          comm.CommandType = CommandType.Text;
40
41          var reader = await comm.ExecuteReaderAsync();
42          var dt = new DataTable();
43          dt.Load(reader);
44
45          if (dt.Rows.Count > 0)
46          {
47              await conn.CloseAsync();
48              return;
49          }
50
51          comm.CommandText = $"create database {config.InitialCatalog}";
52          await comm.ExecuteNonQueryAsync();
53          await conn.CloseAsync();
54      }
55
56      public async Task LoadAsync()
57      {
58          if (loaded)
59              return;
60
61          await CreateDataBaseIfNotExistAsync();
62
63          var config = ObjectRelationalMappingConfig.Config;
64          conn = new SqlConnection(config.StringConnection);
65          await conn.OpenAsync();
66
67          comm = new SqlCommand();
68          comm.Connection = conn;
69          comm.CommandType = CommandType.Text;
70
71          loaded = true;
```

```
72      }
73
74      public async Task CreateIfNotExistAsync(Type type)
75      {
76          await LoadAsync();
77
78          if (await TestExistenceAsync(type))
79              return;
80
81          comm.CommandText = $"create table {type.Name} (";
82
83          foreach (var prop in type.GetProperties())
84          {
85              string column = $"{prop.Name}
    {ConvertToSqlType(prop.PropertyType)}";
86
87              if (prop.Name == "ID")
88                  column += " identity primary key";
89
90              var foreignKeyAtt =
    prop.GetCustomAttribute<ForeignKeyAttribute>();
91              if (foreignKeyAtt != null)
92              {
93                  string temp = comm.CommandText;
94                  await CreateIfNotExistAsync(foreignKeyAtt.ForeignTable);
95                  comm.CommandText = temp;
96                  column += $" references {foreignKeyAtt.ForeignTable.Name}
    (ID)";
97              }
98
99              var notnullAtt = prop.GetCustomAttribute<NotNullAttribute>();
100
101             if (notnullAtt != null)
102                 column += $" not null";
103
104             comm.CommandText += $"{column}, ";
105         }
106         comm.CommandText = comm.CommandText.Substring(0,
    comm.CommandText.Length - 1) + " )";
107
108         await comm.ExecuteNonQueryAsync();
109     }
110
111     public string ConvertToSqlType(Type type)
112     {
113         if (type == typeof(int))
114             return "int";
115
116         if (type == typeof(string))
117             return "varchar(MAX)";
118
119         if (type == typeof(byte[]))
120             return "varbinary";
121
122         if (type == typeof(decimal))
123             return "decimal";
124
125         if (type == typeof(long))
126             return "bigint";
127
```

```
128             if (type == typeof(DateTime))
129                 return "datetime";
130 
131             throw new InvalidColumnType(type);
132         }
133 
134         public async Task<bool> TestExistenceAsync(Type type)
135         {
136             DataTable dt = await ReadTableAsync($"select * from sys.tables
    where name = '{type.Name}'");
137             return dt.Rows.Count > 0;
138         }
139 
140         public async Task<DataTable> ReadTableAsync(string query, params
     SqlParameter[] parameters)
141         {
142             await LoadAsync();
143             comm.CommandText = query;
144             comm.Parameters.Clear();
145             comm.Parameters.AddRange(parameters);
146 
147             var reader = await comm.ExecuteReaderAsync();
148 
149             DataTable dt = new DataTable();
150             dt.Load(reader);
151 
152             return dt;
153         }
154 
155         public async Task ExecuteNonQueryAsync(string query, params
     SqlParameter[] parameters)
156         {
157             comm.CommandText = query;
158             comm.Parameters.Clear();
159             comm.Parameters.AddRange(parameters);
160 
161             await comm.ExecuteNonQueryAsync();
162         }
163 
164         public async Task<T> RunQuery<T>(string query, params SqlParameter[]
    parameters)
165         {
166             var type = typeof(T);
167             var dt = await ReadTableAsync(query, parameters);
168             var isCollection = typeof(IEnumerable).IsAssignableFrom(type);
169 
170             if (isCollection)
171             {
172                 var args = type.GetGenericArguments();
173                 if (args.Length > 0)
174                     type = args[0];
175             }
176 
177             int i = 0;
178             object[] data = new object[dt.Rows.Count];
179 
180             foreach (DataRow row in dt.Rows)
181             {
182                 if (row.ItemArray.Length == 1)
183                 {
```

```
                data[i++] = row.ItemArray[0];
                continue;
            }

            var obj = Activator.CreateInstance(type);

            foreach (var prop in type.GetProperties())
                prop.SetValue(obj, row[prop.Name]);

            data[i++] = obj;
        }

        if (isCollection)
        {
            var listType = typeof(List<>).MakeGenericType(type);
            var list = (IList)Activator.CreateInstance(listType);

            foreach (var x in data)
                list.Add(Convert.ChangeType(x, type));

            return (T)list;
        }

        return (T)data[0];
    }

    public override async Task Insert<T>(T obj)
    {
        await CreateIfNotExistAsync(typeof(T));

        var id = getID<T>();

        List<SqlParameter> parameters = new List<SqlParameter>();
        string query = $"insert {typeof(T).Name} values (";

        foreach (var prop in typeof(T).GetProperties())
        {
            if (prop.Name == id?.Name)
                continue;

            var paramName =  "@" + prop.Name;
            query += paramName + ",";
            parameters.Add(new SqlParameter(paramName,
prop.GetValue(obj)));
        }

        query = query.Substring(0, query.Length - 1) + ")";
        await ExecuteNonQueryAsync(query, parameters.ToArray());
    }

    public override async Task Delete<T>(T obj)
    {
        await CreateIfNotExistAsync(typeof(T));

        var id = getID<T>();

        await ExecuteNonQueryAsync(
            $"delete {typeof(T).Name} where {id.Name} == @{id.Name}",
            new SqlParameter($"@{id.Name}", id.GetValue(obj))
        );
```

```
    }

    public override async Task Update<T>(T obj)
    {
        await CreateIfNotExistAsync(typeof(T));

        var id = getID<T>();

        List<SqlParameter> parameters = new List<SqlParameter>();
        string query = $"update {typeof(T).Name} set \n";

        foreach (var prop in typeof(T).GetProperties())
        {
            if (prop.Name == id?.Name)
                continue;

            var paramName =  "@" + prop.Name;
            query += paramName + ",";
            parameters.Add(new SqlParameter(paramName,
prop.GetValue(obj)));
        }

        query += $" where {id.Name} == @{id.Name}";
        parameters.Add(new SqlParameter($"@{id.Name}", id.GetValue(obj)));

        await ExecuteNonQueryAsync(query, parameters.ToArray());
    }

    private PropertyInfo getID<T>()
    {
        foreach (var prop in typeof(T).GetProperties())
        {
            if (prop.GetCustomAttribute<PrimaryKeyAttribute>() is null)
                continue;

            return prop;
        }

        return null;
    }
}
```

ORMLib/MSSqlQueryable.cs

```
using System;
using System.Linq.Expressions;
using System.Collections.Generic;

namespace ORMLib.MSSql;

using Providers;
using Linq;

public class MSSqlQueryable<T> : IQueryable<T>
{
    public MSSqlQueryable(Expression exp, MSSqlQueryProvider provider)
    {
        this.ElementType = typeof(IEnumerable<T>);
```

```
        this.Expression = exp;
        this.Provider = provider;
    }

    public Type ElementType { get; private set; }

    public Expression Expression { get; private set; }

    public IQueryProvider Provider { get; private set; }
}
```

ORMLib/MSSql/MSSqlQueryProvider.cs

```
using System.Linq;
using System.Threading.Tasks;
using System.Linq.Expressions;
using System.Collections.Generic;

namespace ORMLib.MSSql;

using System.Data.SqlClient;
using Linq;
using Providers;

public class MSSqlQueryProvider : IQueryProvider
{
    public IQueryable<T> CreateQuery<T>(Expression expression)
        => new MSSqlQueryable<T>(expression, this);

    public async Task<T> Execute<T>(Expression expression)
    {
        SqlAccess access = new SqlAccess();
        List<SqlParameter> list = new List<SqlParameter>();
        var query = buildQuery<T>(expression, list);
        // System.Console.WriteLine(query);
        var data = await access.RunQuery<T>(query, list.ToArray());
        return data;
    }

    private string buildQuery<T>(Expression expression, List<SqlParameter>
parameters)
    {
        string query = "";

        if (expression is MethodCallExpression call)
        {
            if (call.Method.Name == "Select" && call.Method.DeclaringType
== typeof(Queryable))
            {
                query = buildQuery<T>(call.Arguments[0], parameters);
                if (call.Arguments[1] is UnaryExpression unary)
                {
                    var exp = unary.Operand;
                    var paramter = exp.ToString().Split(' ')[0];
                    var str = string.Concat(
                        exp.ToString()
                        .SkipWhile(c => c != '>')
                        .Skip(1)
```

```
                    );
                    query = query.Replace("*", str) + " " + paramter;
                }
            }

            if (call.Method.Name == "Where" && call.Method.DeclaringType
== typeof(Queryable))
            {
                query = buildQuery<T>(call.Arguments[0], parameters);
                query = buildQuery<T>(call.Arguments[0], parameters);
                if (call.Arguments[1] is UnaryExpression unary)
                {
                    var exp = unary.Operand;
                    var paramter = exp.ToString().Split(' ')[0];
                    var str = string.Concat(
                        exp.ToString()
                        .SkipWhile(c => c != '>')
                        .Skip(1)
                    );
                    str = str.Replace("==", "=");
                    str = str.Replace("\"", "'");
                    query = $"{query} {paramter} where {str}";
                }
            }
        }
        else if (expression is ConstantExpression constExp)
        {
            var type = constExp.Type;
            if (typeof(IQueryable).IsAssignableFrom(type))
            {
                var queryType = type.GenericTypeArguments[0];
                query = $"select * from {queryType.Name}";
            }
        }

        return query;
    }
}
```

ORMLib/Linq/IQueryable.cs

```
using System;
using System.Linq.Expressions;

namespace ORMLib.Linq;

using Providers;

public interface IQueryable
{
    Type ElementType { get; }
    Expression Expression { get; }
    IQueryProvider Provider { get; }
}

public interface IQueryable<out T> : IQueryable
{

```

```
}
```

ORMLib/Linq/Queryable.cs

```
using System;
using System.Linq;
using System.Threading.Tasks;
using System.Linq.Expressions;
using System.Collections.Generic;

namespace ORMLib.Linq;

public static class Queryable
{
    public static IQueryable<R> Select<T, R>(this IQueryable<T> source,
Expression<Func<T, R>> selector)
        where T : new()
    {
        if (source is null)
            throw new ArgumentNullException("source");

        if (selector is null)
            throw new ArgumentNullException("selector");

        var provider = source.Provider;

        var self = new Func<IQueryable<T>, Expression<Func<T, R>>,
IQueryable<R>>(Select);

        var query = provider.CreateQuery<R>(
            Expression.Call(
                null,
                self.Method,
                source.Expression,
                Expression.Quote(selector)
            )
        );

        return query;
    }

    public static IQueryable<T> Where<T>(this IQueryable<T> source,
Expression<Func<T, bool>> predicate)
        where T : new()
    {
        if (source is null)
            throw new ArgumentNullException("source");

        if (predicate is null)
            throw new ArgumentNullException("predicate");

        var provider = source.Provider;

        var self = new Func<IQueryable<T>, Expression<Func<T, bool>>,
IQueryable<T>>(Where);

        var query = provider.CreateQuery<T>(
            Expression.Call(
```

```
                null,
                self.Method,
                source.Expression,
                Expression.Quote(predicate)
            )
        );

        return query;
    }

    public static async Task<List<T>> ToListAsync<T>(this IQueryable<T>
source)
    {
        if (source is null)
            throw new ArgumentNullException("source");

        var exp = source.Expression;
        var provider = source.Provider;

        var collection = await provider.Execute<IEnumerable<T>>(exp);

        return collection.ToList();
    }
}
```

ORMLib/ORMLib.csproj

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

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="System.Data.SqlClient" Version="4.8.5" />
  </ItemGroup>

</Project>
```

Test/Program.cs

```
using static System.Console;

using ORMLib;
using ORMLib.Linq;
using ORMLib.DataAnnotations;

var builder = ObjectRelationalMappingConfig.GetBuilder();

builder
    .UseMSSqlServer()
    .SetDataSource(@"JVLPC0480\SQLEXPRESS")
    .SetInitialCatalog("MyDatabaseTest")
    .SetIntegratedSecurity(true)
    .Build()
    .Use();
```

```
16
17  while (true)
18  {
19      string command = ReadLine();
20
21      switch (command.ToLower())
22      {
23          case "add marca":
24              Marca marca = new Marca();
25
26              Write("Nome: ");
27              marca.Nome = ReadLine();
28
29              await marca.Save();
30              break;
31
32          case "add produto":
33              Produto produto = new Produto();
34
35              Write("Nome: ");
36              produto.Nome = ReadLine();
37
38              Write("Marca: ");
39              var marcaNome = ReadLine();
40              var marcasSelecionadas = await Marca.All
41                  .Where(m => m.Nome == "Bosch")
42                  .ToListAsync();
43              var marcaSelecionada = marcasSelecionadas[0];
44              produto.MarcaID = marcaSelecionada.ID;
45
46              await produto.Save();
47              break;
48
49          case "view produto":
50              var produtos = await Produto.All
51                  .Select(m => m.Nome)
52                  .ToListAsync();
53              foreach (var nome in produtos)
54              {
55                  WriteLine(nome);
56              }
57              break;
58
59
60          case "view marca":
61              var marcas = await Marca.All
62                  .Select(m => m.Nome)
63                  .ToListAsync();
64              foreach (var nome in marcas)
65              {
66                  WriteLine(nome);
67              }
68              break;
69
70          case "exit":
71              return;
72      }
73  }
74
75  public class Anuncio : Table<Anuncio>
```

```
{
    [PrimaryKey]
    public int ID { get; set; }

    [NotNull]
    public string Titulo { get; set; }

    [NotNull]
    public decimal Preco { get; set; }

    [NotNull]
    [ForeignKey(typeof(Vendedor))]
    public int VendedorID { get; set; }

    [NotNull]
    [ForeignKey(typeof(Produto))]
    public int ProdutoID { get; set; }
}

public class Marca : Table<Marca>
{
    [PrimaryKey]
    public int ID { get; set; }

    [NotNull]
    public string Nome { get; set; }
}

public class Vendedor : Table<Vendedor>
{
    [PrimaryKey]
    public int ID { get; set; }

    public string Nome { get; set; }

    [NotNull]
    public string Login { get; set; }

    [NotNull]
    public string Senha { get; set; }

    public string CPF { get; set; }

    public string PIX { get; set; }
}

public class Produto : Table<Produto>
{
    [PrimaryKey]
    public int ID { get; set; }

    [NotNull]
    public string Nome { get; set; }

    [ForeignKey(typeof(Marca))]
    public int MarcaID { get; set; }
}
```

4.12.4 Entity Framework

O Entity Framework é o ORM padrão do ,NET. É o framework que usaremos para fazer conexão com o banco de dados. Abaixo um exemplo de como utilizá-lo.
A estrutura é semelhante ao que fizermos acima, porém bem mais complexa.

4.12.5 DBFirst, CodeFirst e Scaffolding

Para inicio precisamos falar da diferença entre DBFirst e CodeFirst.
Para utilizar o Entity Framework você precisa definir o seu banco através de código manualmente. Indicar as tabelas e as relações. Isso é chamado de CodeFirst. O Banco de dados será gerado com base no seu modelo. Por outro lado podemos fazer Scaffolding e realizar o DBFirst: Escrever o SQL do banco de dados e gerar o código C# que atende os modelos especificados no banco. Caso seja seu desejo, e nesse tutorial, aqui está um script para a instalação do entity, ferramentas de Scaffolding e ainda a utilização no projeto atual:

createModel.ps1

```
1  $strconn = "Data Source=" + $args[0] + ";Initial Catalog=" + $args[1] +
   ";Integrated Security=True;TrustServerCertificate=true"
2  dotnet add package Microsoft.EntityFrameworkCore.Design
3  dotnet add package Microsoft.EntityFrameworkCore.SqlServer
4  dotnet tool install --global dotnet-ef
5  dotnet ef dbcontext scaffold $strconn
    Microsoft.EntityFrameworkCore.SqlServer --force -o Model
```

Note que você precisa executar passando logo em seguida uma string com o nome do servidor e depois uma string com o nome do banco. Caso tenha problemas com o proxy, você pode solucionar altearando as configurações do Nuget em %appdata%/Nuget/NuGet.config:

NuGet.config

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <configuration>
3      <config>
4          <add key="http_proxy" value="http://BWSOA@rb-proxy-
   de.bosch.com:8080" />
5          <add key="https_proxy" value="http://BWSOA@rb-proxy-
   de.bosch.com:8080" />
6      </config>
7
8    <packageSources>
9      <add key="nuget.org" value="https://api.nuget.org/v3/index.json"
   protocolVersion="3" />
10   </packageSources>
11 </configuration>
```

Vamos, antes de tudo criar o seguinte banco de dados para testes:

```
1  use master
2  go
3
4  if exists(select * from sys.databases where name = 'testentity')
5      drop database testentity
6  go
7
8  create database testentity
9  go
10
11 use testentity
```

```
go

create table Produto(
    ID int identity primary key,
    Nome varchar(100) not null,
    Descricao varchar(MAX) not null,
    Foto image null
);
go

create table Usuario(
    ID int identity primary key,
    Nome varchar(120) not null,
    DataNascimento date not null,
    Foto image null
);
go

create table Oferta(
    ID int identity primary key,
    Produto int references Produto(ID) not null,
    Usuario int references Usuario(ID) not null,
    Preco decimal not null
);
go
```

Após executar o script esperamos o seguinte resultado em arquivos gerados numa pasta Model:

Model/Ofertum.cs

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

namespace entityTest.Model;

public partial class Ofertum
{
    public int Id { get; set; }

    public int Produto { get; set; }

    public int Usuario { get; set; }

    public decimal Preco { get; set; }

    public virtual Produto ProdutoNavigation { get; set; }

    public virtual Usuario UsuarioNavigation { get; set; }
}
```

Model/Produto.cs

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

namespace entityTest.Model;

```

```
6   public partial class Produto
7   {
8       public int Id { get; set; }
9
10      public string Nome { get; set; }
11
12      public string Descricao { get; set; }
13
14      public byte[] Foto { get; set; }
15
16      public virtual ICollection<Ofertum> Oferta { get; set; } = new
        List<Ofertum>();
17  }
```

Model/Usuario.cs

```
1   using System;
2   using System.Collections.Generic;
3
4   namespace entityTest.Model;
5
6   public partial class Usuario
7   {
8       public int Id { get; set; }
9
10      public string Nome { get; set; }
11
12      public DateTime DataNascimento { get; set; }
13
14      public byte[] Foto { get; set; }
15
16      public virtual ICollection<Ofertum> Oferta { get; set; } = new
        List<Ofertum>();
17  }
```

Model/TestentityContext.cs

```
1   using System;
2   using System.Collections.Generic;
3   using Microsoft.EntityFrameworkCore;
4
5   namespace entityTest.Model;
6
7   public partial class TestentityContext : DbContext
8   {
9       public TestentityContext()
10      {
11      }
12
13      public TestentityContext(DbContextOptions<TestentityContext> options)
14          : base(options)
15      {
16      }
17
18      public virtual DbSet<Ofertum> Oferta { get; set; }
19
```

```
20      public virtual DbSet<Produto> Produtos { get; set; }
21
22      public virtual DbSet<Usuario> Usuarios { get; set; }
23
24      protected override void OnConfiguring(DbContextOptionsBuilder
        optionsBuilder)
25      #warning To protect potentially sensitive information in your connection
        string, you should move it out of source code. You can avoid scaffolding
        the connection string by using the Name= syntax to read it from
        configuration - see https://go.microsoft.com/fwlink/?linkid=2131148. For
        more guidance on storing connection strings, see http://go.microsoft.com/
        fwlink/?LinkId=723263.
26          => optionsBuilder.UseSqlServer("Data Source=CT-C-00189\
        \SQLEXPRESS01;Initial Catalog=testentity;Integrated
        Security=True;TrustServerCertificate=true");
27
28      protected override void OnModelCreating(ModelBuilder modelBuilder)
29      {
30          modelBuilder.Entity<Ofertum>(entity =>
31          {
32              entity.HasKey(e =>
        e.Id).HasName("PK__Oferta__3214EC27449381AC");
33
34              entity.Property(e => e.Id).HasColumnName("ID");
35              entity.Property(e => e.Preco).HasColumnType("decimal(18, 0)");
36
37              entity.HasOne(d => d.ProdutoNavigation).WithMany(p =>
        p.Oferta)
38                  .HasForeignKey(d => d.Produto)
39                  .OnDelete(DeleteBehavior.ClientSetNull)
40                  .HasConstraintName("FK__Oferta__Produto__286302EC");
41
42              entity.HasOne(d => d.UsuarioNavigation).WithMany(p =>
        p.Oferta)
43                  .HasForeignKey(d => d.Usuario)
44                  .OnDelete(DeleteBehavior.ClientSetNull)
45                  .HasConstraintName("FK__Oferta__Usuario__29572725");
46          });
47
48          modelBuilder.Entity<Produto>(entity =>
49          {
50              entity.HasKey(e =>
        e.Id).HasName("PK__Produto__3214EC271CBD74B4");
51
52              entity.ToTable("Produto");
53
54              entity.Property(e => e.Id).HasColumnName("ID");
55              entity.Property(e => e.Descricao)
56                  .IsRequired()
57                  .IsUnicode(false);
58              entity.Property(e => e.Foto).HasColumnType("image");
59              entity.Property(e => e.Nome)
60                  .IsRequired()
61                  .HasMaxLength(100)
62                  .IsUnicode(false);
63          });
64
65          modelBuilder.Entity<Usuario>(entity =>
66          {
```

```
            entity.HasKey(e =>
e.Id).HasName("PK__Usuario__3214EC27A5C9C1B3");

            entity.ToTable("Usuario");

            entity.Property(e => e.Id).HasColumnName("ID");
            entity.Property(e => e.DataNascimento).HasColumnType("date");
            entity.Property(e => e.Foto).HasColumnType("image");
            entity.Property(e => e.Nome)
                .IsRequired()
                .HasMaxLength(120)
                .IsUnicode(false);
        });

        OnModelCreatingPartial(modelBuilder);
    }

    partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
}
```

Note que a classe TestentityContext gerá todo código que poderia ser feito a mão. Abaixo um exemplo de como usar a estrutura criada:

Program.cs

```
using System;
using System.Linq;
using System.Threading.Tasks;

using entityTest.Model;

TestentityContext context = new TestentityContext();

await createUser("Don");
await createUser("Marcão");
await createUser("Queila Lima");
await createUser("Pamella");

await createProduct("Bico", "O máximo do avanço tecnológico em mecânica.")
;
await createProduct("Pudim", "O máximo do avanço tecnológico em
gastronomia.");

var users =
    from user in context.Usuarios
    where user.Nome == "Queila Lima"
    select user;

var queila = users.FirstOrDefault();

var produtos =
    from product in context.Produtos
    where product.Nome == "Pudim"
    select product;

var pudim = produtos.FirstOrDefault();

await addOfertum(pudim, queila, 10);

```

```
printData();

async Task addOfertum(Produto produto, Usuario usuario, decimal value)
{
    Ofertum oferta = new Ofertum();
    oferta.Usuario = usuario.Id;
    oferta.Produto = produto.Id;
    oferta.Preco = value;

    context.Oferta.Add(oferta);
    await context.SaveChangesAsync();

}

async Task createProduct(string nome, string desc)
{
    Produto produto = new Produto();
    produto.Nome = nome;
    produto.Descricao = desc;

    context.Produtos.Add(produto);
    await context.SaveChangesAsync();
}

void printData()
{
    var query =
        from user in context.Usuarios
        join ofer in context.Oferta
        on user.Id equals ofer.Usuario
        select new {
            nome = user.Nome,
            produto = ofer.Produto,
            valor = ofer.Preco
        } into x
        join prod in context.Produtos
        on x.produto equals prod.Id
        select new {
            nome = x.nome,
            produto = prod.Nome,
            valor = x.valor
        };
    foreach (var y in query)
        Console.WriteLine($"{y.nome} está vendendo um {y.produto} por
{y.valor} R$.");
}

async Task createUser(string nome)
{
    Usuario usuario = new Usuario();
    usuario.Nome = nome;
    usuario.DataNascimento = DateTime.Now;
    usuario.Foto = null;

    context.Usuarios.Add(usuario);
    await context.SaveChangesAsync();
}
```

Um fato interessante é, que assim como na nossa biblioteca, métodos LINQ são convertidos para SQL e executados com máximo desempenho no banco de dados sem precisar buscar grandes quantidades de

dados para o Front-End. Além disso, queries usando select, where, join e etc, como apresentados neste exemplo são convertidos para LINQ e, consequentemente, reconvertidos para SQL. Desta forma, escrever "from user in context.Usuarios select user" é o mesmo que escrever no SQL "select * from Usuarios".

4.12.6 Exercícios

Adicione uma tabela a escolha com algum relacionamento, gere o modelo novamente, e faça uma inserção e consulta ao dados. Tente modificar os dados e veja o que acontece.

4.13 Aula 29 - Desafio 9

4.13.1 Enterprise Resource Planning

ERPs são sistemas que conectam vários subsistemas de uma empresa em um único ambiente com a intenção de permitir o gerenciamento da equipe, dos gastos e dos processos em um único lugar. Além disso, ele pode reunir vários tipos de sistemas como sistemas de apoio a decisão e visualização de dados.

4.13.2 Robotic Process Automation

RPAs são programas de computador com o foco em automatizar processos. ERPs e RPAs podem interagir fortemente, visto que automatizar processos em ERPs com bots pode aumentar muito a produtividade dos setores administrativos de uma empresa.

4.13.3 Extract, Transform, Load

ETLs são programas de computador que tem a capacidade de extrair dados de diversos lugares, transformá-los e carregá-los em outro lugar como, por exemplo, um banco de dados. ETLs podem ser processos realizados por RPAs ainda mais dentro de ERPs. Desta forma esses são três assuntos que se interconectam e conversam bastante.

4.13.4 Ferramentas para ETL e RPA no C#

Vamos agora ver algumas ferramentas que possibilitam o ETL e o RPA no C#. Primeiramente podemos realizar a instalação do PowerShell no seu projeto:

```
dotnet add package Microsoft.PowerShell.SDK
dotnet add package System.Management.Automation
```

Isso nos possibilitará criar uma instância do PowerShell e executá-lo. Qualquer coisa que funciona no PowerShell funciona aqui. Abaixo você pode ver um bot clonando um repositório para você.

```
using System;
using System.Management.Automation;

using var ps = PowerShell.Create();

ps
    .AddCommand("git")
    .AddArgument("clone")
    .AddArgument("https://github.com/trevisharp/pamella");

var result = ps.Invoke();

foreach (var line in result)
    Console.WriteLine(line);
```

Você também pode usar a System.IO para acessar arquivos e procurar pastas no seu computador. Até mesmo pastas normalmente invisíveis como a .git:

```
using System;
using System.IO;

var ls = Directory.EnumerateDirectories("..//orkestra-lang");

foreach (var dir in ls)
    Console.WriteLine(dir);
```

Para ETL utilize o Entity Framework.

4.13.5 Desafio

Crie uma aplicação console ou desktop que ira gerenciar os projetos da sua equipe. Nele você poderá registrar pastas no seu computador. Um bot, periodicamente ou ao ser acionado numa opção atualizar, irá buscar nas pastas registradas e em todas as pastas filhas procurando pastas .git que irão indicar um projeto git. Após isso, o Bot ira realizar uma operação de ETL carregando no banco de dados o registro de todos os repositórios nas pastas escolhidas. Os repositórios encontrados serão posteriormente apresentados no banco de dados. Você ainda pode escolher a opção de realizar um pull automático em todos os repositórios encontrados.

4.14 Aula 30 - Desafio 10

Em breve...

5 4 - Desenvolvimento Web com .NET

5.1 Aulas

- Aula 31 - Angular Básico
- Aula 32 - Componentes, Templates e Binding em Angular
- Aula 33 - SPA, Roteamento avançado e Injeção de Dependência
- Aula 34 - Storage, Forms e Design Materials
- Aula 35 - Introdução a Desenvolvimento Backend
- Aula 36 - Conectando Back e Front
- Aula 37 - Um exemplo completo com salvamento de imagens
- Aula 38 - Segurança de Sistemas
- Aula 39 - Sistemas de Autentificação
- Aula 40 - Desafio 12

5.2 Duração Estimada

- 32 horas de conteúdo.
- 8 horas de desafios.
- 4 horas de revisão (opcional).
- 4 horas de avaliação (opcional).
- Total: 40 a 48 horas.

5.3 Overview

Este curso ensinará como utilizar conceitos avançados do C# para criar aplicações web modernas, seguras e rápidas com um frontend escrito no framework Angular.

5.4 Competências

Em definição.

5.5 Aula 31 - Angular Básico

- Iniciando no Angular
- O básico sobre Componentes
- O básico sobre Rotas
- O básico sobre Binding e Templates
- Próximos passos
- Exercícios

5.5.1 Iniciando no Angular

Para começar vamos a instalação do Angular. Você só precisa do **npm** instalado no seu computador para que possa executar o seguinte comando:

```
npm install -g @angular/cli
```

Em alguns computadores é necessário permitir a execução de scripts PowerShell não assinados, você pode fazer isso executando:

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned
```

E então, dentro de uma pasta qualquer inicializar o seu projeto da seguinte forma:

```
ng new example-app
```

Existe uma grande possibilidade do comando ng não ser encontrado. Ele é instalado junto ao primeiro comando na pasta %AppData%, mas pode não ser reconhecível pelo PowerShell em um primeiro momento. Você pode executá-lo através do **npm** para que ele seja reconhecível da seguinte forma:

```
npm run ng new example-app
```

Note que é comum, no Angular, usar um convensão para nomes exatamente iguail ao padrão de nome de repositórios no GitHub.

Nosso último comando irá criar um projeto Angular vazio, muito embora o projeto base seja extremamente carregado de arquivos. A estrutura incial do projeto é semelhante a abaixo:

```
example-app
    .angular/
    .vscode/
    node_modules/
    src
        app
            app-routing.module.ts
            app.component.css
            app.component.html
            app.component.spec.ts
            app.component.ts
            app.module.ts

        assets/
        index.html
        main.ts
        style.css

    .editorconfig
    .gitignore
    angular.json
    package-lock.json
    package.json
    README.md
    tsconfig.app.json
    tsconfig.json
    tsconfig.spec.json
```

Enquanto alguns arquivos são mais utilizados para configuração e geração de código, como index.html e o styles.css que muitas vezes não são alterados por serem apenas templates para o projeto, a pasta app é onde o trabalho em geral acontece. Para iniciar a execução do projeto você pode utilizar um comando que fica definido na package.json:

```
npm start
```

Ele irá compilar todo código e criar um servidor web que responderá em um link que será mostrado no terminal no seguinte modelo: http://localhost:port/. Comumente a porta será 4200. Ao acessar, o servidor nos dará uma página 100% Html, Css e JavaScript algumas vezes irreconhecível ao código original.
Agora vamos compreender como um projeto angular está estrututrado.

5.5.2 O básico sobre Componentes

O Angular é estruturado em componentes, isto é, uma estrutura que pode ser replicada como se fosse uma tag própria. Ela tem seu prórpio html, cdd e comportamento, que por sua vez não é definido com JavaScript (.js) no Angular, mas sim TypeScript (.ts), uma tecnologia que se converte em JavaScript ao ser executado. Assim, todo componente terá 4 arquivos associados a ele:

- x.component.css: O css que é aplicado apenas a este componente.
- x.component.html: O html a ser reproduzido ao utilizarmos este componente.
- x.cmponent.spec.ts: Especificação de testes do componente.
- x.component.ts: Classe TypeScirpt que define comportamento do componente.

Como você pode perceber, você inicia sua aplicação com um componente chamado app. Note que existem ainda dois arquivos não mensionados acima, o 'app-routing.module.ts' e o 'app.module.ts', mas esses não fazem parte do componente app e sim arquivos importantes de configuração do projeto em si.
Toda vez que você inicia uma aplicação o componente app será renderizado para você. Note que existe um grane comentário neste componente apontando que todo conteúdo nele é apenas de exemplo. Por isso, no component.html iremos simplificá-lo deixando-o apenas assim:

app.component.html

```
1  <router-outlet></router-outlet>
```

A tag 'router-outlet' será subsittuido por um código presente em alguma rota específica (depende do URL acessado). Mais tarde veremos como isso funciona, mas antes, vamos a uma definição básica para compreendermos como tudo funciona. Vamos criar nosso próprio componente. Você poderia fazer isso apenas criando arquvios e digitando código, mas também pode usar o seguinte código em um terminal PowerShell qualquer:

```
npm run ng generate component Main
```

Uma pasta dentro de app será criada com o nome 'main'. Assim você cria seus próprios componentes. A ideia dos componentes é utilizá-los em outros componentes e usar como telas existentes em sua aplicação. Ambos os casos são possíveis. A seguir veremos como usar estes componentes como nossas telas. Para isso criaremos um outro componente:

```
npm run ng generate component Second
```

5.5.3 O básico sobre Rotas

O arquivo app-routing.module.ts indica qual componente deve ser renderizado em um 'router-outlet' com base no URL da página. É simples trabalhar com ele, observe:

app-routing.module.ts

```
1  import { NgModule } from '@angular/core';
2  import { RouterModule, Routes } from '@angular/router';
3  import { MainComponent } from './main/main.component';
4  import { SecondComponent } from './second/second.component';
5
6  const routes: Routes = [
7    { path: '', component: MainComponent },
8    { path: 'second-component', component: SecondComponent }
9  ];
10
11 @NgModule({
12   imports: [RouterModule.forRoot(routes)],
13   exports: [RouterModule]
14 })
15 export class AppRoutingModule { }
```

Nas linhas 7 e 8 eu defino que o MainComponent deve ser renderizado na página principal, ou seja, na http://localhost:port/, e a segunda página http://localhost:port/second-component. Para criar uma navegabilidade iremos estruturar as páginas da seguinte forma:

main.component.html

```
1  <p>main works!</p>
2  <a routerLink="/second-component" routerLinkActive="active"
   ariaCurrentWhenActive="page">Second Component</a>
```

second.component.html

```
1  <p>second works!</p>
```

```
2   <a routerLink="" routerLinkActive="active" ariaCurrentWhenActive="page">Fi
    rst Component</a>
```

Assim podemos fazer uma navegabilidade simples mas funcional.
Note que tudo está sendo renderizado no app.component.html. Se alteramos esse aquivo colocando alguma informação como, por exemplo, passando as tags 'a' de seus componentes para o appcomponent.html, poderemos criar uma guia de navegação:

app.component.html

```
1   <nav>
2       <ul>
3           <li>
4               <a routerLink="" routerLinkActive="active"
    ariaCurrentWhenActive="page">First Component</a>
5           </li>
6           <li>
7               <a routerLink="/second-component" routerLinkActive="active"
    ariaCurrentWhenActive="page">Second Component</a>
8           </li>
9       </ul>
10  </nav>
11
12  <router-outlet></router-outlet>
```

Note que este 'nav' ficará em todas as páginas pois ela está no app. Tudo será renderizado logo abaixo da navegação.
Por organização, você ainda poderia separar a navegação em mais um componente:

```
npm run ng generate component Nav
```

nav.component.html

```
1   <ul>
2       <li>
3           <a routerLink="" routerLinkActive="active" ariaCurrentWhenActive="
    page">First Component</a>
4       </li>
5       <li>
6           <a routerLink="/second-component" routerLinkActive="active"
    ariaCurrentWhenActive="page">Second Component</a>
7       </li>
8   </ul>
```

app.component.html

```
1   <header>
2       Meu site
3   </header>
4
5   <nav>
6       <app-nav></app-nav>
7   </nav>
8
9   <main>
10      <router-outlet></router-outlet>
```

```
</main>

<footer>
    Todos os direitos reservados
</footer>
```

Note que o nome do componente é app seguido do nome do componente.

5.5.4 O básico sobre Binding e Templates

A comunicação do Angular com a tela e os eventos pode ser complexa de se compreender. É possível associar valores dentro da classe TypeScript e a tela. Enquanto o Template permite a construção de telas mais facilmente e o Binding a conectar eventos e informações ao código de comportamento do componente.

main.component.html

```
<section>
    <p>Bem-vindo ao meu sistema!</p>
</section>

<section>
    <p>
        <input (input)="update($event)">
    </p>
    <p>
        <button (click)="onClick()">Salvar</button>
    </p>
</section>

<section>
    <p>
        Texto Salvo: {{savedText}}
    </p>
</section>
```

main.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-main',
  templateUrl: './main.component.html',
  styleUrls: ['./main.component.css']
})
export class MainComponent {
  text = "Altere aqui..."
  savedText = ""

  onClick()
  {
    this.savedText = this.text
  }

  update(event:any)
  {
      this.text = event.target.value
  }
}
```

Os parêntesis ao redor de 'input' apontam um biding que levará dados da tela (view) para o código (data source). Assim quando um texto é digitado a função update é chamada e atualiza a propriedade text que o componente tem internamente. Ao clicar em Salvar, temos outro biding semelhante que chamará o onClick no componente colocando o texto no savedText. Quando savedText é atualizado, a última secção que utiliza chaves duplas atualiza seu valor. Essas chaves duplas é uma interpolação de texto, isso significa que o que estiver na variável savedText é apresentado na tela. Note que .html e o .ts conversam de diversas formas. Note que dentro das funções TypeScript podemos utilizar JavaScript puro.

5.5.5 Próximos passos

Nas próximas aulas iremos ir a fundo nos tópicos que apresentamos nessa aula. Componentes, Rotas, Templates, Binding além de outros tópicos como formulários e bilbiotecas de componentes. Num primeiro momento, este é o overview necessário para se compreender como o Angular conversa e atual sobre os dados.

5.5.6 Exercícios

Faça um sistema de utilidades com uma calculadora, um conversor de temperaturas e um conversor de binário para decimal. Utilize as mesmas estruturas e recursos vistas nessa aula introdutória e crie um css para deixar seu sistema apresentável.

5.6 Aula 32 - Componentes, Templates e Binding em Angular

5.6.1 Fazendo um projeto um pouco mais complexo

Nesta aula compreenderemos profundamente os componentes, templates e binding. Para isso vamos mostrar um projeto simples e ainda incompleto onde usamos tudo que já sabemos e nos aprofundamos no que tange a componentes.

app.component.html

```
<header>
    <h1>Rede Social Minimalista</h1>
</header>

<nav>
    <app-nav></app-nav>
</nav>

<main>
    <router-outlet></router-outlet>
</main>

<footer>
    Todos os direitos reservados
</footer>
```

app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ComunityPageComponent } from './comunity-page/comunity-page.component';
import { FeedPageComponent } from './feed-page/feed-page.component';
import { HomePageComponent } from './home-page/home-page.component';
import { LoginPageComponent } from './login-page/login-page.component';
```

```
import { NewAccountPageComponent } from './new-account-page/new-account-page.component';
import { NotFoundPageComponent } from './not-found-page/not-found-page.component';
import { RecoverPageComponent } from './recover-page/recover-page.component';
import { UserPageComponent } from './user-page/user-page.component';

const routes: Routes = [
  { path: "", component: HomePageComponent },
  { path: "login", component: LoginPageComponent },
  { path: "feed", component: FeedPageComponent },
  { path: "comunity", component: ComunityPageComponent },
  { path: "newaccount", component: NewAccountPageComponent},
  { path: "recover", component: RecoverPageComponent},
  { path: "user", component: UserPageComponent},
  { path: "**", component: NotFoundPageComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

nav.component.html

```
<ul class="loginBox">
    <li>
        <a href="/">Home</a>
    </li>
    <li>
        <a href="/">Feed</a>
    </li>
    <li>
        <a href="/">Comunidades</a>
    </li>
    <li>
        <a href="/">Login</a>
    </li>
</ul>
```

nav.component.css

```
.loginBox {
    width: 100%;
    display: flex;
    flex-direction: row;
    list-style-type: none;
    align-content: center;
    align-items: center;
    justify-content: space-evenly;
    background-color: lightgray;
    height: 2rem;
}

```

```
.loginBox a {
    padding: 0.5rem;
}

.loginBox a:hover {
    background-color: gray;
    color: white;
}
```

not-found-page.component.html

```
<div class="not-found-box">
    <h1>Not Found</h1>
    <img src="https://media.makeameme.org/created/file-not-found-c17b083c9c.jpg">
</div>
```

not-found-page.component.css

```
.not-found-box {
    display: flex;
    flex-direction: column;
    align-items: center;
}
```

home-page.component.html

```
<marquee>
    <h2>
        <a href="/newaccount">Venha fazer parte a nossa Comunidade</a>
    </h2>
</marquee>
```

Essa simples rede social mostrará como os componentes podem ser utilizados. Vamos priorizar a estrutura antes de começar a usar Bootstrap entre outras tecnologias para tornar nosso sistema bonito. Assim implementaremos, nessa aula, as seguintes páginas:

- Login
- NewAccount
- Comunity
- Feed
- Recover (recuperar senha)

Iniciando pelo Login, podemos propor um componente que controle o fluxo de senha.

5.6.2 Fazendo um componente complexo

Agora vamos fazer um componente complexo com vários tipos de mecânicas do Angular como two-way binding entre outros tipos, eventos de clico de vida como ngOnInit, Inputs e Outputs e trabalhar mais com eventos.

password.component.html

```
<div>

    <p>
        <label>Senha</label>
```

```
        <!-- A linha só será pulada se breakLineOnInput for verdadeiro -->
        <br *ngIf="breakLineOnInput">
        <!--
            ngModel permite que façamos um two-way bind. Isso é, ao alterar uma variável
            na classe alteramos na tela e ao alterarmos na tela (por meio da ação do usuário)
            alteramos na classe também
            Outro fator importante é que chamamos eventos como click e change ao invés de onclick ou onchange.
            <input type={{inputType}}> é, em geral equivalente a <input [type]="inputType">
        -->
        <input type={{inputType}} [style]=inputStyle (focusout)="passwordFocusout()"
[(ngModel)]="inputText" (keydown)="passwordChanged()" (change)="passwordChanged()" (click)="passwordClick()">
    </p>

    <!-- Este p e tudo que está dentro dele só aparecerá na tela se canSeePassword for verdadeiro -->
    <p *ngIf="canSeePassword">
        <!--
            Porquê não usar click aqui? Bem, o evento click é chamado antes que o Model seja atualizado.
            Assim seePassword apresentará o valor incorreto!
         -->
        <input type="checkbox" [(ngModel)]="seePassword" (change)="checkBoxToogle($event)">
        <label>Mostrar Senha</label>
    </p>

</div>
```

password.component.ts

```
import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';

@Component({
  selector: 'app-password',
  templateUrl: './password.component.html',
  styleUrls: ['./password.component.css']
})
export class PasswordComponent implements OnInit {

  // Inputs podem ser acessados de fora do componente como propriedades HTML
  // Outputs podem ser acessados de fora do componente como eventos no estilo onclick
  @Output() valueChanged = new EventEmitter<string>();

  @Input() breakLineOnInput = true;
  @Input() canSeePassword = true;

  @Input() seePassword = false
```

```
18    @Output() seePasswordChanged = new EventEmitter<boolean>();
19
20    // Membros protegidos não podem ser usados fora da classe, apenas no
      html
21    protected inputType = "text";
22    protected inputStyle = "color: black;"
23    protected inputText = "";
24    protected initialState = true;
25
26    // Implmentamos OnInit para executar algum comportamento quando o
      componente inicializa
27    ngOnInit(): void
28    {
29      // Atualizamos o inputType que aparece na tela
30      this.updateInput()
31    }
32
33    /*  Aqui o evento foi pedido usando checkBoxToogle($event) no html e foi
      recuperado aqui.
34      *  Isso não necessariamente precisa ser feito. Neste caso poderiamos
      usar apenas checkBoxToogle()
35      *  e deixar essa função sem parâmetros já que não usamos o resultado do
      evento 'newValue'.
36      */
37    protected checkBoxToogle(newValue: any)
38    {
39      this.updateInput()
40      this.seePasswordChanged.emit(this.seePassword);
41    }
42
43    protected updateInput()
44    {
45      if (this.initialState)
46      {
47        this.inputText = "Escreva sua senha..."
48        this.inputType = "text"
49        this.inputStyle = "color: gray;"
50        return
51      }
52
53      this.inputStyle = "color: black;"
54      this.inputType = this.seePassword ? "text" :  "password";
55    }
56
57    protected passwordChanged()
58    {
59      this.updateInput()
60      this.valueChanged.emit(this.inputText)
61    }
62
63    protected passwordClick()
64    {
65      if (!this.initialState)
66        return
67
68      this.initialState = false;
69      this.inputText = "";
70      this.updateInput();
71    }
72
```

```
  protected passwordFocusout()
  {
    if (this.inputText !== "")
      return

    this.initialState = true
    this.updateInput()
  }
}
```

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { NavComponent } from './nav/nav.component';
import { LoginPageComponent } from './login-page/login-page.component';
import { HomePageComponent } from './home-page/home-page.component';
import { NotFoundPageComponent } from './not-found-page/not-found-page.component';
import { FeedPageComponent } from './feed-page/feed-page.component';
import { ComunityPageComponent } from './comunity-page/comunity-page.component';
import { NewAccountPageComponent } from './new-account-page/new-account-page.component';
import { RecoverPageComponent } from './recover-page/recover-page.component';
import { UserPageComponent } from './user-page/user-page.component';
import { PasswordComponent } from './password/password.component';
import { FormsModule } from '@angular/forms'; // Adicionado para poder usar o ngModel

@NgModule({
  declarations: [
    AppComponent,
    NavComponent,
    LoginPageComponent,
    HomePageComponent,
    NotFoundPageComponent,
    FeedPageComponent,
    ComunityPageComponent,
    NewAccountPageComponent,
    RecoverPageComponent,
    UserPageComponent,
    PasswordComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule  // Adicionado para poder usar o ngModel
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Agora temos um interessante componente para colocar a senha com comportamento mais complexo. Note que nesta aula iremos apenas construir uma tela, não nos preocuparemos em adicionar reais funcionalidades como Login funcional, por exemplo. Assim nossa tela de Login é bem simples e com pouco comportamento:

login-page.component.html

```
<h1>
    Login
</h1>

<p>
    <label>Email/Username</label>
    <br>
    <input>
</p>

<app-password [seePassword]="false" [breakLineOnInput]="true" />

<p>
    <button>Logar</button>
</p>

<p>
    Não possui conta? <a href="/newaccount">Crie uma agora mesmo!</a>
</p>

<p>
    Esqueceu sua senha? <a href="/recover">Recupere agora!</a>
</p>
```

5.6.3 Reutilização de Componentes, Gerênciamento de dados e ngFor

Agora vamos fazer a tela de criação de conta para termos uma ideia do que mais poderíamos fazer:

create-password.component.html

```
<app-password (valueChanged)="passwordChanged($event)" />

<!-- Para cada elemento em passStrong coloca uma div a mais -->
<div class="strongBox">
    <div *ngFor="let x of passStrong">
        <div class="strongPass"></div>
    </div>
    {{passClassify}}
</div>

<p>
    <label>Repetir Senha</label>
    <br>
    <input type="password"
 [(ngModel)]="repeat" (change)="updateRepeatCondition()">
</p>

<p style="color: red;">
    {{ repeatEqualToPass ? "" : "As senhas devem ser iguais"}}
```

```
</p>
```

create-password.component.ts

```
import { Component, OnChanges, SimpleChanges } from '@angular/core';

@Component({
  selector: 'app-create-password',
  templateUrl: './create-password.component.html',
  styleUrls: ['./create-password.component.css']
})
export class CreatePasswordComponent {

  protected passStrong = Array(1);
  protected password = "";
  protected repeat = "";
  protected passClassify = "";
  protected repeatEqualToPass = true;

  protected updateStrongBar()
  {
    let finalStrong = 1;

    if (this.password.length > 3)
      finalStrong++;

    if (this.password.length > 5)
      finalStrong++;

    if (this.password.length > 7)
      finalStrong++;

    if (this.password.length > 9)
      finalStrong++;

    if (this.password.match("[a-z]") != null)
      finalStrong++;

    if (this.password.match("[A-Z]") != null)
      finalStrong++;

    if (this.password.match("[0-9]") != null)
      finalStrong++;

    if (this.password.match("[\W]") != null)
      finalStrong++;

    this.passStrong = Array(finalStrong);

    if (finalStrong < 3)
    {
      this.passClassify = "Senha muito fraca";
    }
    else if (finalStrong < 5)
    {
      this.passClassify = "Senha fraca"
    }
    else if (finalStrong < 7)
```

```
    {
      this.passClassify = "Senha mediana"
    }
    else if (finalStrong < 9)
    {
      this.passClassify = "Senha forte"
    }
    else
    {
      this.passClassify = "Senha muito forte"
    }
  }

  protected updateRepeatCondition()
  {
    this.repeatEqualToPass = this.password === this.repeat
  }

  protected passwordChanged(event: any)
  {
    this.password = event;
    this.updateStrongBar()
    this.updateRepeatCondition()
  }
}
```

create-password.component.css

```
.strongPass {
  width: 20px;
  height: 20px;
  margin: 2px;
  background: green;
}

.strongBox {
  display: flex;
  flex-direction: row;
}
```

new-account-page.component.html

```
<h1>
    Nova Conta
</h1>

<p>
    <label>Email</label>
    <br>
    <input>
</p>

<p>
    <label>Username</label>
    <br>
    <input>
```

```
15  </p>
16
17  <app-create-password/>
18
19  <p>
20      <button>Criar Conta</button>
21  </p>
```

5.6.4 Exercícios

1. Faça um componente personalizado para armazenar e validar CPF.
2. Adicione-o a tela de criação de contas.
3. Faça um componente de card que possa armazenar uma imagem e um texto.
4. Tente criar vários posts na tela de feed com base nesses cads.

5.7 Aula 33 - SPA, Roteamento avançado e Injeção de Dependência

- Roteamento Avançado e SPA
- Roteamento Aninhado
- Titulos das Páginas
- Usando injeção de dependência
- ActivatedRoute
- Objetos injetáveis costumizáveis

5.7.1 Roteamento Avançado e SPA

SPA ou Single-Page Application é uma aplicação Web que possui apenas uma única página. Isso signiifca que tudo que acontece ocorre sobre um único HTML. O Angular em si é um SPA. Isso significa que todas as páginas renderizadas são uma única página que altera seu conteúdo constantemente. Para isso serve o sistemas de rotas: Ao alterar a URL podemos mudar a renderização e se basear em uma aplicação de única página.
Já vimos antes como escolher o componente renderizado baseado na rota na Aula 11 e usamos bastante na aula 12:

app-routing.module.ts

```
1   const routes: Routes = [
2     { path: "", component: HomePageComponent },
3     { path: "login", component: LoginPageComponent },
4     { path: "feed", component: FeedPageComponent },
5     { path: "comunity", component: ComunityPageComponent },
6     { path: "newaccount", component: NewAccountPageComponent},
7     { path: "recover", component: RecoverPageComponent},
8     { path: "user", component: UserPageComponent},
9     { path: "**", component: NotFoundPageComponent }
10  ];
```

Vimos que o "**" permite que nós criemos uma página de NotFound e ainda aprendemos a usar tags 'a' para redirecionar para outras páginas. Vamos fazer nossas SPA usando apenas o roteamento do Angular.

5.7.2 Roteamento Aninhado

No Angular podemos usar roteamento aninhado:

app-routing.module.ts

```
1   const routes: Routes = [
2     { path: "", component: HomePageComponent },
3     {
4       path: "login",
```

```
    component: LoginPageComponent,
    children: [
      { path: "newaccount", component: NewAccountPageComponent }
    ]
  },
  { path: "feed", component: FeedPageComponent },
  { path: "comunity", component: ComunityPageComponent },
  { path: "recover", component: RecoverPageComponent },
  { path: "user", component: UserPageComponent },
  { path: "**", component: NotFoundPageComponent }
];
```

Aqui dizemos que a nova conta é uma rota interna do login. Isso nos permite fazer o seguinte:

login-page.component.html

```
<h1>
    Login
</h1>

<p>
    <label>Email/Username</label>
    <br>
    <input>
</p>

<app-password [seePassword]="false" [breakLineOnInput]="true" />

<p>
    <button>Logar</button>
</p>

<p>
    Não possui conta? <a routerLink="newaccount">Crie uma agora mesmo!</a>
</p>

<p>
    Esqueceu sua senha? <a href="/recover">Recupere agora!</a>
</p>

<router-outlet></router-outlet>
```

Usando routerLink ao invés de href sem a barra indicamos querer acessar a url /login/newaccount que renderizará no lugar do "router-outlet". Isso é prefeito para fazer as mais variáveis páginas.

5.7.3 Titulos das Páginas

É possível também

app-routing.module.ts

```
const routes: Routes = [
  { path: "", title: "Rede Social Minimalista", component:
HomePageComponent },
  {
    path: "login",
    title: "Autentificação",
    component: LoginPageComponent,
    children: [
      { path: "newaccount", component: NewAccountPageComponent }
```

```
    ]
  },
  { path: "feed", title: "Feed", component: FeedPageComponent },
  { path: "comunity", title: "Comunidades", component:
ComunityPageComponent },
  { path: "recover", title: "Recuperar Senha", component:
RecoverPageComponent },
  { path: "user", title: "Página de Usuário", component:
UserPageComponent },
  { path: "**", title: "Not Found", component: NotFoundPageComponent }
];
```

5.7.4 Usando injeção de dependência

Injeção de dependência acontece, no Angular, quando o framework cria um objeto para nós que podemos requisitar no construtor de um componente. Ou seja, não criamos e configuramos um objeto, mas esperamos recebê-lo no construtor. Você verá um exemplo abaixo, onde usamos a injeção de dependência para conseguir um objeto que controla as rotas no angular.

5.7.5 ActivatedRoute

Desejamos que, caso o usuário decida recuperar sua senha e tenha deixado o email na sua tentativa de login, a página de recuperar senha abra com o email já definido para, supostamente, enviar um email de recuperação de senha. Podemos passar informações entre as telas da seguinte forma:

app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ComunityPageComponent } from './comunity-page/comunity-
page.component';
import { FeedPageComponent } from './feed-page/feed-page.component';
import { HomePageComponent } from './home-page/home-page.component';
import { LoginPageComponent } from './login-page/login-page.component';
import { NewAccountPageComponent } from './new-account-page/new-account-
page.component';
import { NotFoundPageComponent } from './not-found-page/not-found-
page.component';
import { RecoverPageComponent } from './recover-page/recover-
page.component';
import { UserPageComponent } from './user-page/user-page.component';

const routes: Routes = [
  { path: "", title: "Rede Social Minimalista", component:
HomePageComponent },
  {
    path: "login",
    title: "Autentificação",
    component: LoginPageComponent,
    children: [
      { path: "newaccount", component: NewAccountPageComponent }
    ]
  },
  { path: "feed", title: "Feed", component: FeedPageComponent },
  { path: "comunity", title: "Comunidades", component:
ComunityPageComponent },
  { path: "recover/:email", title: "Recuperar Senha", component:
RecoverPageComponent }, // Agora podemos mandar recover/valor para mandar
um parâmetro para rota
  { path: "recover", title: "Recuperar Senha", component:
RecoverPageComponent }, // Podemos fazer rotas com e sem parâmetros
```

```
  { path: "user", title: "Página de Usuário", component: UserPageComponent },
  { path: "**", title: "Not Found", component: NotFoundPageComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Agora vamos alterar a tela de login para permitir que essa rota seja atingida:

login-page.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-login-page',
  templateUrl: './login-page.component.html',
  styleUrls: ['./login-page.component.css']
})
export class LoginPageComponent {
  email = ""
  link = ''
}
```

login-page.component.html

```
<h1>
    Login
</h1>

<p>
    <label>Email/Username</label>
    <br>
    <input [(ngModel)]="email">
</p>

<app-password [seePassword]="false" [breakLineOnInput]="true" />

<p>
    <button>Logar</button>
</p>

<p>
    Não possui conta? <a routerLink="newaccount">Crie uma agora mesmo!</a>
</p>

<p>
    Esqueceu sua senha? <a href="{{'/recover/' + email}}">Recupere agora!
</a>
</p>

<router-outlet></router-outlet>
```

recover-page.component.html

```
<p>
    Digite seu email para recuperar a senha:
</p>

<input value="{{email}}"/>

<button>Enviar Email</button>
```

recover-page.component.ts

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-recover-page',
  templateUrl: './recover-page.component.html',
  styleUrls: ['./recover-page.component.css']
})
export class RecoverPageComponent implements OnInit, OnDestroy {
  email = "";
  subscription: any;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    this.subscription = this.route.params.subscribe(params => {
       this.email = params['email'];
    });
  }

  ngOnDestroy() {
    this.subscription.unsubscribe();
  }
}
```

Outro objeto interessante que podemos obter com a injeção de dependências é o Router. Ele permite que controlemos a rota ao invés de apenas obter dados dela:

recover-page.component.html

```
<p>
    Digite seu email para recuperar a senha:
</p>

<input value="{{email}}"/>

<button (click)="send()">Enviar Email</button>
```

recover-page.component.ts

```
import { Component, OnInit, OnDestroy } from '@angular/core';
```

```
import { ActivatedRoute, Router } from '@angular/router'; // Adicionamos o Router aqui

@Component({
  selector: 'app-recover-page',
  templateUrl: './recover-page.component.html',
  styleUrls: ['./recover-page.component.css']
})
export class RecoverPageComponent implements OnInit, OnDestroy {
  email = "";
  subscription: any;

  constructor(private route: ActivatedRoute, private router: Router) { } // E aqui

  ngOnInit() {
    this.subscription = this.route.params.subscribe(params => {
       this.email = params['email'];
    });
  }

  ngOnDestroy() {
    this.subscription.unsubscribe();
  }

  send() {
    // Send Email Here
    this.router.navigate(["/login"]) // E usamos para redirecionar aqui
  }
}
```

5.7.6 Objetos injetáveis costumizáveis

Podemos fazer nossos próprios objetos injetáveis para criar serviços. Isso é importante e será muito usado no desenvolvimento backend também. Vamos criar uma pasta de serviços e então trabalhar em cima disso:

```
cd .\src\
cd .\app\
mkdir cep-service
ni cep-data.ts
cd ..
cd ..
npm run ng generate service cep-service/cep
```

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { NavComponent } from './nav/nav.component';
import { LoginPageComponent } from './login-page/login-page.component';
import { HomePageComponent } from './home-page/home-page.component';
import { NotFoundPageComponent } from './not-found-page/not-found-page.component';
import { FeedPageComponent } from './feed-page/feed-page.component';
import { ComunityPageComponent } from './comunity-page/comunity-page.component';
```

```
import { NewAccountPageComponent } from './new-account-page/new-account-page.component';
import { RecoverPageComponent } from './recover-page/recover-page.component';
import { UserPageComponent } from './user-page/user-page.component';
import { PasswordComponent } from './password/password.component';
import { FormsModule } from '@angular/forms';
import { CreatePasswordComponent } from './create-password/create-password.component'; // Added for use ngModel
import { HttpClientModule } from '@angular/common/http'; // Added for use HttpClient

@NgModule({
  declarations: [
    AppComponent,
    NavComponent,
    LoginPageComponent,
    HomePageComponent,
    NotFoundPageComponent,
    FeedPageComponent,
    ComunityPageComponent,
    NewAccountPageComponent,
    RecoverPageComponent,
    UserPageComponent,
    PasswordComponent,
    CreatePasswordComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule, // Added for use ngModel
    HttpClientModule // Added for use HttpClient
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

cep-data.ts

```

export interface CepData
{
    cep: string;
    logradouro: string;
    complemento: string;
    bairro: string;
    localidade: string;
    uf: string;
    ibge: string;
    gia: string;
    ddd: string;
    siafi: string;
}
```

cep.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { CepData } from './cep-data'

@Injectable({
  providedIn: 'root'
})
export class CepService {

  constructor(private http: HttpClient) { }

  getStreet(cep: string)
  {
    return this.http.get<CepData>("https://viacep.com.br/ws/" + cep + "/
json/");
  }
}
```

new-account-page.component.html

```
<h1>
    Nova Conta
</h1>

<p>
    <label>Email</label>
    <br>
    <input>
</p>

<p>
    <label>CEP</label>
    <br>
    <input [(ngModel)]="cepvalue" (change)="cepAdded()">
</p>

<p>
    <label>Rua</label>
    <br>
    <input [(ngModel)]="ruavalue">
</p>

<p>
    <label>Username</label>
    <br>
    <input>
</p>

<app-create-password/>

<p>
    <button>Criar Conta</button>
</p>
```

new-account-page.component.ts

```
import { Component } from '@angular/core';
import { CepService } from '../services/cep.service';

@Component({
  selector: 'app-new-account-page',
  templateUrl: './new-account-page.component.html',
  styleUrls: ['./new-account-page.component.css']
})
export class NewAccountPageComponent {
  cepvalue = ""
  ruavalue = ""

  constructor(private cep: CepService) { }

  cepAdded()
  {
    this.cep.getStreet(this.cepvalue)
      .subscribe(x =>
      {
        this.ruavalue = x.logradouro
      })
  }
}
```

Com esse serviço criamos um sistema de acesso ao CEP. Usamos a classe HttpClient para acessar um serviço de CEP para obter a rua de um CEP.

5.8 Aula 34 - Storage, Forms e Design Materials

- Session e Local Storage
- Forms
- Design Materials
- Exercícios

5.8.1 Session e Local Storage

Todo navegador trás a possibilidade de se armazenar dados localmente. Isto é, no cliente. Isso significa que podemos guardar dados no computador do usuário para uso futuro e que seja disponível de forma global na aplicação. O Local Storage é mantido mesm que o navegador é fechado. Sistemas como Termo, por exemplo, guardam resultados do usuário no local estorage. O Session Storage é limpo ao encerrarmos a sessão, isto é, está associado a uma aba do navegador.
Para acessar essa storage no angular é relativamente simples:

login-page.component.html

```
<h1>
    Login
</h1>

<p>
    <label>Email/Username</label>
    <br>
    <input [(ngModel)]="email">
</p>

<app-password
[seePassword]="false" [breakLineOnInput]="true" (valueChanged)="passwordCh
anged($event)" />
```

```
12
13  <p>
14      <button (click)="login()">Logar</button>
15  </p>
16
17  <p>
18      Não possui conta? <a routerLink="newaccount">Crie uma agora mesmo!</a>
19  </p>
20
21  <p>
22      Esqueceu sua senha? <a href="{{'/recover/' + email}}">Recupere agora!
    </a>
23  </p>
24
25  <router-outlet></router-outlet>
```

login-page.component.ts

```
1   import { Component } from '@angular/core';
2   import { Router } from '@angular/router';
3
4   @Component({
5     selector: 'app-login-page',
6     templateUrl: './login-page.component.html',
7     styleUrls: ['./login-page.component.css']
8   })
9   export class LoginPageComponent {
10    email = ""
11    link = ""
12    password = ""
13
14    constructor(private router: Router) { }
15
16    passwordChanged(event : any)
17    {
18      this.password = event
19    }
20
21    login()
22    {
23      // Aqui precisariamos fazer essa verificação no banco de dados
24      if (this.email == "email@email.com" && this.password == "123")
25      {
26        // Isso evidentemente não é seguro, mas a ideia é bom e será
    melhorada no futuro
27        sessionStorage.setItem('user', 'pamella');
28        this.router.navigate(["/feed"])
29      }
30    }
31  }
```

feed-page.component.ts

```
1   import { Component, OnInit } from '@angular/core';
2
3   @Component({
```

```
  selector: 'app-feed-page',
  templateUrl: './feed-page.component.html',
  styleUrls: ['./feed-page.component.css']
})
export class FeedPageComponent implements OnInit {
  user: string | null = null

  ngOnInit(): void
  {
    this.user = sessionStorage.getItem("user")
  }
}
```

feed-page.component.html

```
{{user != null ? "Bem-vind@, " + user : "Logue para ver ser feed"}}
```

Todas as outras interações de dados são feitas em servidores feitos futuramente por nós.

5.8.2 Forms

Você pode usar formulários reativos para criar cadastros mais simples e poderosos. O assunto é longo mas o básico é um sistema de validação usando o FormControl:

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { NavComponent } from './nav/nav.component';
import { LoginPageComponent } from './login-page/login-page.component';
import { HomePageComponent } from './home-page/home-page.component';
import { NotFoundPageComponent } from './not-found-page/not-found-page.component';
import { FeedPageComponent } from './feed-page/feed-page.component';
import { ComunityPageComponent } from './comunity-page/comunity-page.component';
import { NewAccountPageComponent } from './new-account-page/new-account-page.component';
import { RecoverPageComponent } from './recover-page/recover-page.component';
import { UserPageComponent } from './user-page/user-page.component';
import { PasswordComponent } from './password/password.component';
import { FormsModule } from '@angular/forms';
import { CreatePasswordComponent } from './create-password/create-password.component';
import { HttpClientModule } from '@angular/common/http';
import { ReactiveFormsModule } from '@angular/forms'; // Added for use ReactiveForms

@NgModule({
  declarations: [
    AppComponent,
    NavComponent,
    LoginPageComponent,
    HomePageComponent,
    NotFoundPageComponent,
```

```
    FeedPageComponent,
    ComunityPageComponent,
    NewAccountPageComponent,
    RecoverPageComponent,
    UserPageComponent,
    PasswordComponent,
    CreatePasswordComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    HttpClientModule,
    ReactiveFormsModule // Added for use ReactiveForms
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

new-account-page.component.ts

```
import { Component } from '@angular/core';
import { FormControl, Validators } from '@angular/forms';
import { CpfService } from '../services/cpf.service';

@Component({
  selector: 'app-new-account-page',
  templateUrl: './new-account-page.component.html',
  styleUrls: ['./new-account-page.component.css']
})
export class NewAccountPageComponent {
  cepvalue = ""
  ruavalue = ""

  email = new FormControl('', [
    Validators.required,
    Validators.email,
    Validators.minLength(4)
  ]);

  constructor(private cep: CpfService) { }

  cepAdded()
  {
    this.cep.getStreet(this.cepvalue)
      .subscribe(x =>
      {
        this.ruavalue = x.logradouro
      })
  }
}
```

new-account-page.component.html

```
<h1>
```

```
    Nova Conta
</h1>

<label>Email</label>
<br>
<input type="text" [formControl]="email">

<div *ngIf="email.invalid" style="color: red;">
    <div *ngIf="email.errors?.['required']">
        Name is required.
    </div>
    <div *ngIf="email.errors?.['minlength']">
    Name must be at least 4 characters long.
    </div>
</div>

<p>
    <label>CEP</label>
    <br>
    <input [(ngModel)]="cepvalue" (change)="cepAdded()">
</p>

<p>
    <label>Rua</label>
    <br>
    <input [(ngModel)]="ruavalue">
</p>

<p>
    <label>Username</label>
    <br>
    <input>
</p>

<app-create-password/>

<p>
    <button>Criar Conta</button>
</p>
```

5.8.3 Design Materials

Para começar a deixar a aplicação mais bonita podemos, ao invés de investir pesadamente em html e css, podemos usar componentes feitos por outras pessoas com css feito por outras pessoas, como se fosse um bootstrep construído sobre o Angular. Um das bilbiotecas de componentes que existem é o Angular Material. A instalação pode ser feita da seguinte maneira:

```
npm run ng add @angular/material
```

Algumas perguntas são feitas no terminal como tipografia e paleta de cores. Após isso você pode usar todos os componentes do Angular Material, basta adicionar no app module, como cita a documentação do Angular Material:

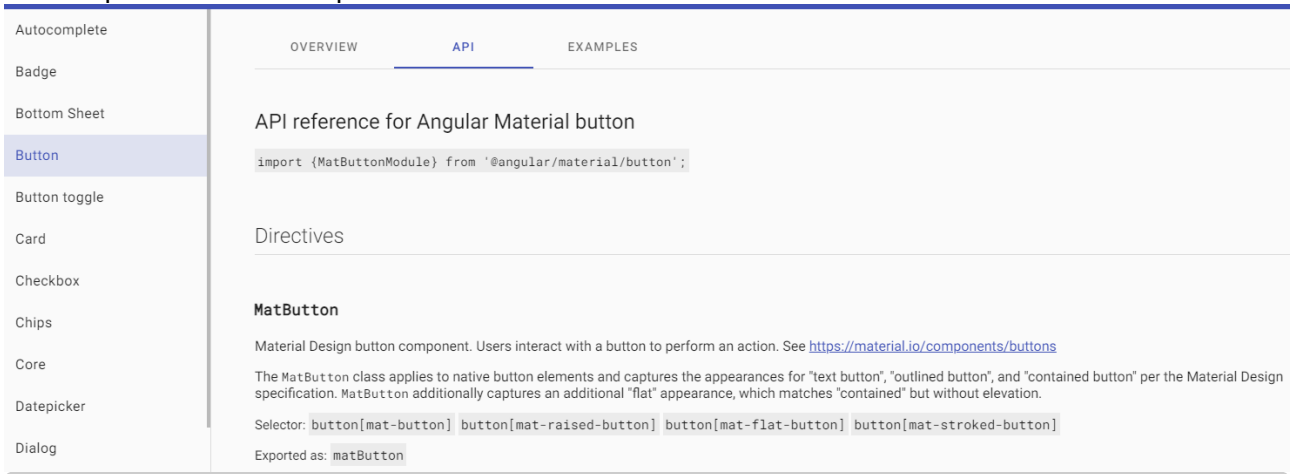
app.module.ts

```
// ...
import { MatSlideToggleModule } from '@angular/material/slide-toggle';

@NgModule ({
```

```
// ...
  imports: [
    // ...
    MatSlideToggleModule
  ]
})
class AppModule {}
```

Vamos fazer um exemplo básico adicionando um botão mais bonito, mas você pode olhar mais componentes em: https://material.angular.io/components. Em cada componente, na aba API podemos ver como importar e usar o componente:



app.module.ts

```
1  import { NgModule } from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3
4  import { AppRoutingModule } from './app-routing.module';
5  import { AppComponent } from './app.component';
6  import { NavComponent } from './nav/nav.component';
7  import { LoginPageComponent } from './login-page/login-page.component';
8  import { HomePageComponent } from './home-page/home-page.component';
9  import { NotFoundPageComponent } from './not-found-page/not-found-
   page.component';
10 import { FeedPageComponent } from './feed-page/feed-page.component';
11 import { ComunityPageComponent } from './comunity-page/comunity-
   page.component';
12 import { NewAccountPageComponent } from './new-account-page/new-account-
   page.component';
13 import { RecoverPageComponent } from './recover-page/recover-
   page.component';
14 import { UserPageComponent } from './user-page/user-page.component';
15 import { PasswordComponent } from './password/password.component';
16 import { FormsModule } from '@angular/forms';
17 import { CreatePasswordComponent } from './create-password/create-
   password.component';
18 import { HttpClientModule } from '@angular/common/http';
19 import { ReactiveFormsModule } from '@angular/forms';
20 import { BrowserAnimationsModule } from '@angular/platform-browser/
   animations';
21 import { MatButtonModule } from '@angular/material/button'; // Added for
   use Angular Material Button
22
23
```

```
@NgModule({
  declarations: [
    AppComponent,
    NavComponent,
    LoginPageComponent,
    HomePageComponent,
    NotFoundPageComponent,
    FeedPageComponent,
    ComunityPageComponent,
    NewAccountPageComponent,
    RecoverPageComponent,
    UserPageComponent,
    PasswordComponent,
    CreatePasswordComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    HttpClientModule,
    ReactiveFormsModule, BrowserAnimationsModule,
    MatButtonModule // Added for use Angular Material Button
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

login-page.component.html

```
<h1>
    Login
</h1>

<p>
    <label>Email/Username</label>
    <br>
    <input [(ngModel)]="email">
</p>

<app-password
[seePassword]="false" [breakLineOnInput]="true" (valueChanged)="passwordCh
anged($event)" />

<p>
    <button mat-button color="primary" (click)="login()">Logar</button>
</p>

<p>
    Não possui conta? <a routerLink="newaccount">Crie uma agora mesmo!</a>
</p>

<p>
    Esqueceu sua senha? <a href="{{'/recover/' + email}}">Recupere agora!
</a>
</p>

```

```
25  <router-outlet></router-outlet>
```

5.8.4 Exercícios

Explore o Angular Material e os Forms e aplique-os sobre o nosso sistema.

5.9 Aula 35 - Introdução a Desenvolvimento Backend

- Fundamentos em Arquitetura Web
- Requisições HTTP, REST e JSON
- C# WebAPI
- Rotas e Parâmetros
- Injeção de Dependência e Serviços
- Padrão Repository
- Classe HttpClient

5.9.1 Fundamentos em Arquitetura Web

O desenvolvimento Web pode ter muitas abordagens. Uma abordagem antiga porém ainda muito usada são os monólitos. Monólitos são aplicações que não separam seus componentes em mais de um sistema. Isso significa que a aplicação se conectará no banco de dados, fará todo o processamento e, ainda, disponibilizará o front-end de alguma forma. Por exemplo, poderá retornar código CSS/HTML para o cliente com os dados de uma dada página. Aplicações monolíticas rodam sobre uma única máquina e estão sujeitos aos limites de desempenho deste servidor.
Podemos também usar a arquitetura de Microsserviços para melhorar o desempenho. Ela consiste em separar o projeto em vários pequenos programas que podem rodar em diversos computadores ou na nuvem. Esses serviços se comunicam e podem ser acessados por um API Gateway.
Neste curso consideraremos diversas abordagens para resolução dos problemas, muito embora, certamente teremos ao menos 2 serviços: O Angular para frontend e o C# para backend.
Diferentemente de outras oportunidades, não utilizaremos exatamente do MVC para criar aplicações. O MVC geralmente funciona como aplicações como monolíticas que cuidam de todos os processos do software como o ciclo de renderização. Aqui trabalharemos com front e back desacoplados, isto é, duas aplicações independentes que se comunicam. É importante notar que em diferentes projetos podemos ter diferentes abordagens e padrões arquitetônicos.

5.9.2 Requisições HTTP, REST e JSON

A comunicação entre processos se da por requisições HTTP. Essas são requisições TCP, ou seja, sem perdas de dados. Como toda requisição HTTP temos componentes em comum, mas isso não é suficiente para caracterizar a forma como vamos trabalhar. Iremos trabalhar sobre o padrão REST. O REST (Representational State Transfer) é um padrão de comunicação que envolve o uso de método, cabeçalho e corpo. Quando uma aplicação segue o REST chamamos ela de RESTful. Abaixo um exemplo de requisição REST:

```
https://myservice.com/endpoint

GET

content-type: application/json

{
    "mydata": 4
}
```

O request tem seu alvo (uma URL). Note que isso caracteriza o REST como uma comunicação unidimensional. Evidentemente, nada impede que 2 sistemas usem o REST e tentem conversar, contudo, a priori, a estruturas não facilitaram uma conversa sequencial e é ineficaz para implementar coisas como jogos online. Apesar disso, ainda sim existe uma, e apenas uma, resposta de toda requisição REST.
Logo depois vem o método. A principio o método só tem sentido semântico. Isso significa ele não afeta em nada o comportamento da Request, porém, os padrões REST apontam métodos válidos e quando usamos cada um deles. Os mais importantes são: GET, usado ao ler dados; POST, usado ao salvar dados; PUT, usado ao alterar os dados; e DELETE, usado ao deletar dados. Muito embora sejam semânticos é importante seguir as recomendações. Além disso, algumas bibliotecas podem bloquear algumas operações inconsistentes com seus métodos.

Ai temos o header. No header traremos várias configurações e definições sobre a requisição em si.
Por último o corpo. Nem todas as requisições devem ter corpo. Por exemplo, em geral requisições GET não possuem (mas seu retorno contém). O corpo são os dados e podem ser estruturados em XML, JSON ou até mesmo HTML. O mais usado será o JSON (JavaScript Object Notation). Basicamente a notação usada para definir objetos no JavaScript como dicionários é extremamente utilizada para enviar dados pela rede em requisições REST.
Ao acessar uma página na web é comum que estejamos realizando um GET e obtendo um HTML da página. Nosso serviço Angular fará isso o tempo todo.

5.9.3 C# WebAPI

Para criar uma WEP API Rest em C# que está apta a receber e responder requisições REST. Para isso podemos usar o seguinte comando:

```
dotnet new webapi
```

Ele irá inicializar um projeto bem mais complexo que uma aplicação console. Inclusive alguns arquivos de exemplo que podem ser removidos. São eles o WeatherForecast e Controllers/WeatherForecastController. Na pasta de Controllers ficarão os controladores. Um controlador é uma classe que possui endpoints que, por sua vez, são códigos executáveis que podem receber requisições REST. Após isso faramos um arquivos TestController.cs na pasta de Controllers para criar nosso primeiro endpoint.

```
1  using Microsoft.AspNetCore.Mvc;
2
3  namespace ProjetoWeb.Controllers;
4
5  [ApiController]
6  [Route("test")]
7  public class TestController : ControllerBase
8  {
9      [HttpGet]
10     public string Get()
11     {
12         return "Server is running...";
13     }
14 }
```

Como você pode ver ele será acessado da seguinte forma: um requisição de GET para https://nomedoservidor/test e será apresentando "Server is running..." na tela do navegador, ou seja, na resposta da requisição. A seguir vamos explorar um pouco como configuramos tudo. Por enquanto é bom atentar-se a simplicidade da estrutura: Trata-se de uma função que retorna exatamente o que será recebido do outro lado. Você ainda pode retornar objetos complexos que serão convertidos automaticamente em um JSON para consumir em outros lugares.

5.9.4 Rotas e Parâmetros

```
1  public class CpfService
2  {
3      private int getVerificationDigits(Cpf cpf)
4      {
5          var str = cpf.Value;
6
7          int sum = 0;
8          for (int i = 0; i < 8; i++)
9          {
10             int digit = str[i] - '0';
11             sum += (i + 2) * digit;
12         }
13         int verifier1 = sum % 11;
14
15         for (int i = 0; i < 8; i++)
16         {
```

```
            int digit = str[i] - '0';
            sum -= digit;
        }
        sum += 9 * verifier1;
        int verifier2 = sum % 11;

        return 10 * verifier1 + verifier2;
    }

    public void Validate(Cpf cpf)
    {
        if (cpf is null)
            throw new ArgumentNullException("cpf");

        int verifier = getVerificationDigits(cpf);
        bool isValid = verifier == cpf.VerificationDigit;

        cpf.Verified = true;
        cpf.Validated = isValid;
    }

    public Cpf Generate(int region = -1)
    {
        Cpf cpf = new Cpf();

        if (region == -1)
            region = Random.Shared.Next(10);

        cpf.FiscalRegionDigit = region;
        cpf.RandomDigits = Random.Shared.Next(100_000_000);
        cpf.VerificationDigit = getVerificationDigits(cpf);

        cpf.Verified = true;
        cpf.Validated = true;
        return cpf;
    }
}

public class Cpf
{
    public string Value
    {
        get => $"{RandomDigits:000.000.00}{FiscalRegionDigit}-
{VerificationDigit:00}";
        set
        {
            if (value is null)
                throw new InvalidCastException("Value can't be null.");

            value = value
                .Replace("-", "")
                .Replace(".", "");

            if (value.Length != 11)
                throw new InvalidCastException("Invalid number of digits.")
;

            RandomDigits = int.Parse(value.Substring(0, 8));
            FiscalRegionDigit = int.Parse(value.Substring(8, 1));
            VerificationDigit = int.Parse(value.Substring(9, 2));
```

```
        }
    }

    public string Region
    {
        get => FiscalRegionDigit switch
        {
            1 => "DF, GO, MT, MS e TO",
            2 => "AC, AP, AM, PA, RO e RR",
            3 => "CE, MA e PI",
            4 => "AL, PB, PE e RN",
            5 => "BA e SE",
            6 => "MG",
            7 => "ES e RJ",
            8 => "SP",
            9 => "PR e SC",
            10 => "RS",
            _ => "Região desconhecida"
        };
    }

    public int RandomDigits { get; set; }
    public int FiscalRegionDigit { get; set; }
    public int VerificationDigit { get; set; }
    public bool Verified { get; set; } = false;
    public bool Validated { get; set; } = false;
}
```

Controllers/CpfController.cs

```
using Microsoft.AspNetCore.Mvc;

namespace ProjetoWeb.Controllers;

[ApiController]
[Route("cpf")]
public class CpfController : ControllerBase
{
    [HttpGet("{cpf}")]
    public ActionResult<Cpf> Get(string cpf)
    {
        Cpf result = new Cpf();
        try
        {
            result.Value = cpf;
        }
        catch (Exception ex)
        {
            return BadRequest(ex.Message);
        }

        return result;
    }
}
```

5.9.5 Injeção de Dependência e Serviços

Podemos permitir através de injeção de dependência que o .NET gerencie quais serviços, como o CpfService, serão entregues a uma função. Veja abaixo como cadastrar um serviço e como usá-lo:

Program.cs

```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
builder.Services.AddTransient<CpfService>();

var app = builder.Build();
```

CpfController.cs

```
    }

    // Usa um serviço usando o atributo FromServices que foi cadastrado no Program.cs
    [HttpGet("generate/{region}")]
    public ActionResult<Cpf> Generate([FromServices]CpfService cpf, int region)
    {
        var result = cpf.Generate(region);

        return result;
    }
}
```

AddTransient criará um objeto sempre que necessário que o objeto seja recriado toda vez que precisar ser utilizado. Ainda existem o AddScoped que cria o objeto que é reutilizado para o mesmo escopo, isso é confuso e facilmente confundível com o AddTransient, mas construiremos um exemplo para podermos diferenciá-los. Ainda existe o AddSingleton que criará um objeto único usado por toda a aplicação.
Além disso, como serviços e controladores são constantemente criados pelo Framework, você pode pedir um tipo no construtor que será injetado com base nas suas configurações e você pode salvar este objeto na classe como se fosse global. Note que a cada requisição recebida os controladores e os serviços não singleton serão recriados com base na demanda.
Outro fator interessante é que podemos criar dependências baseadas em interfaces, isso significa que podemos criar uma estrutura completa baseando-se apenas nas interfaces dos serviços e mudar suas implementações como quisermos. Faremos um exemplo interessante logo a seguir junto de um interessante e novo padrão de projetos.

5.9.6 Padrão Repository

Vamos supor que queremos testar uma aplicação criada sem afetar os dados reais que já estão no sistema. Isso pode ser uma tarefa chata ou até mesmo difícil. Vamos aplicar na conexão com o seguinte banco de dados:

```
create database repoExemplo
go

use repoExemplo
go

create table Mensagem(
    ID int identity primary key,
    Texto varchar(MAX) not null,
    Horario datetime not null
)
go
```

Vamos criar uma conexão com o banco de dados com o Entity framework e vamos definir a seguinte interface para representar um repositório, ou seja uma estrutura de acesso a um conjunto de dados.

Isso nos dará um RepoExemploContext e usaremos ele como um serviço no nosso sistema.
Vamos agora implementar o padrão repository baseado na seguinte interface:

IRepository.cs

```
using System.Linq.Expressions;

public interface IRepository<T>
{
    Task<List<T>> Filter(Expression<Func<T, bool>> exp);
    void Add(T obj);
    void Delete(T obj);
    void Update(T obj);
}
```

Um repositório então, seria um objeto capaz de adicionar, alterar, deletar e ler um objeto de um tipo T.
Podemos fazer implementações concretas como o repositório de mensagem abaixo:

MessageRepository.cs

```
using backend.Model;

public class MessageRepository : IRepository<Mensagem>
{
    private RepoExemploContext entity;
    public MessageRepository(RepoExemploContext service)
        => this.entity = service;

    public async Task<List<Mensagem>> Filter(Expression<Func<Mensagem,
bool>> exp)
    {
        return await entity.Mensagems
            .Where(exp)
            .ToListAsync();
    }

    public void Add(Mensagem obj)
    {
        entity.Add(obj);
        entity.SaveChanges();
    }

    public void Delete(Mensagem obj)
    {
        entity.Remove(obj);
        entity.SaveChanges();
    }

    public void Update(Mensagem obj)
    {
        entity.Update(obj);
        entity.SaveChanges();
    }
}
```

Note que essa classe pede no construtor um RepoExemploContrext, ou seja, só poderá ser instanciado por injeção de dependência se for configurado um RepoExemploContext na nossa Program.cs. Observe o que será feita neste arquivo abaixo:

Program.cs

```
builder.Services.AddSwaggerGen();

builder.Services.AddScoped<RepoExemploContext>();
builder.Services.AddTransient<IRepository<Mensagem>, MessageRepository>();
builder.Services.AddTransient<CpfService>();

var app = builder.Build();
```

Observe que interessante: Ao se pedir um repositório de mensagens você receberá a implementação do MessageRepository. Essa implementação é transiente, ou seja, será instanciado toda que vez que necessário. Já o RepoExemploContext é baseado em escopo, ou seja, para cada requisição será criado uma única instância. Isso significa que se tivermos 2 serviços que utilizam banco de dados sendo usados e ambos precisam usar o contexto do banco de dados o mesmo contexto do banco de dados será utilizado. Isso é ótimo pois impede que duas conexões diferentes sejam usadas em cada repositório (isso pode causar problemas ao fazer joins, já que duas conexões diferentes não podem montar e executar a mesma query).
Podemos fazer outros repositórios que conectam com outros bancos ou tem dados falsos apenas para testes, por exemplo, um repositório fake, observe:

FakeMessageRepository.cs

```
using backend.Model;

public class FakeMessageRepository : IRepository<Mensagem>
{
    private List<Mensagem> fakeData = new List<Mensagem>()
    {
        new Mensagem()
        {
            Id = 1,
            Horario = DateTime.Now.AddDays(-1),
            Texto = "Don, platina o cabelo por favor"
        },
        new Mensagem()
        {
            Id = 2,
            Horario = DateTime.Now.AddDays(-.5),
            Texto = "Se alguém perguntar, você não sabe o que aconteceu com a robodrill"
        },
        new Mensagem()
        {
            Id = 2,
            Horario = DateTime.Now,
            Texto = "VOCÊ SABE QUEM QUEBROU A ROBODRILL?"
        },
    };

    public void Add(Mensagem obj)
        => fakeData.Add(obj);

    public void Delete(Mensagem obj)
        => fakeData.Remove(obj);

    public void Update(Mensagem obj)
```

```
    {
        var old = fakeData
            .FirstOrDefault(m => m.Id == obj.Id);
        if (obj is null)
            return;

        fakeData.Remove(old);
        fakeData.Add(obj);
    }
}
```

E no Program.cs poderemos fazer o seguinte:

Program.cs

```
builder.Services.AddSwaggerGen();

var env = builder.Environment;

builder.Services.AddScoped<RepoExemploContext>();
if (env.IsDevelopment())
    builder.Services.AddTransient<IRepository<Mensagem>,
FakeMessageRepository>();
else if (env.IsProduction())
    builder.Services.AddTransient<IRepository<Mensagem>,
MessageRepository>();

builder.Services.AddTransient<CpfService>();

var app = builder.Build();
```

Se executar com um "**dotnet run**" executaremos como desenvolvimento usando um repositório fake, se usarmos "**dotnet run --environment Production**" estaremos iniciando em produção e conectando-se a um banco real. Então um controlador pode ter comportamento diferente a depender do environment definido no projeto.

RepositoryController.cs

```
using backend.Model;
using Microsoft.AspNetCore.Mvc;

namespace ProjetoWeb.Controllers;

[ApiController]
[Route("message")]
public class MesssageController : ControllerBase
{
    [HttpGet]
    public async Task<ActionResult<IEnumerable<Mensagem>>> GetAll(
        [FromServices]IRepository<Mensagem> repo
    )
    {
        var result = await repo.Filter(x => true);
        return result;
    }
}
```

5.9.7 Classe HttpClient

Podemos fazer requisições a outros serviços pré-existente para realizar tarefas. Vamos então definir um projeto para buscar dados a respeito de um CEP inserido no sistema. Iremos utilizar o Via Cep para descobrirmos a rua, cidade e estado de um Cep inserido no sistema. Para isso usaremos a classe HttpClient:

```
1  public class CepData
2  {
3      public string Cep { get; set; }
4      public string Logradouro { get; set; }
5      public string Complemento { get; set; }
6      public string Bairro { get; set; }
7      public string Uf { get; set; }
8      public string Ibge { get; set; }
9      public string Gia { get; set; }
10     public string Ddd { get; set; }
11     public string Siafi { get; set; }
12 }
```

ICepService.cs

```
1  public interface ICepService
2  {
3      Task<CepData> Get(string cep);
4  }
```

CepService.cs

```
1  using System.Net;
2
3  public class CepService : ICepService
4  {
5      public CepService(string service)
6          => this.client = new HttpClient()
7          {
8              BaseAddress = new Uri(service)
9          };
10
11     private HttpClient client;
12
13     public async Task<CepData> Get(string cep)
14     {
15         var response = await client
16             .GetAsync($"/{cep}/json");
17
18         if (response.StatusCode != HttpStatusCode.OK)
19             return null;
20
21         var obj = await response.Content
22             .ReadFromJsonAsync<CepData>();
23
24         return obj;
25     }
26 }
```

Program.cs

```
var env = builder.Environment;
const string url = "https://viacep.com.br/ws/80320330/json/";

builder.Services.AddScoped<RepoExemploContext>();
if (env.IsDevelopment())
    builder.Services.AddTransient<IRepository<Mensagem>,
FakeMessageRepository>();
else if (env.IsProduction())
    builder.Services.AddTransient<IRepository<Mensagem>,
MessageRepository>();

builder.Services.AddSingleton<ICepService>(p => new CepService(url));

builder.Services.AddTransient<CpfService>();
```

CepController.cs

```
using Microsoft.AspNetCore.Mvc;

namespace ProjetoWeb.Controllers;

[ApiController]
[Route("cep")]
public class CepController : ControllerBase
{
    [HttpGet("{cep}")]
    public async Task<ActionResult<CepData>> Get(
        [FromServices]CepService service, string cep
    )
    {
        var result = await service.Get(cep);

        if (result is null)
            return NotFound();

        return result; // implicit convertion to ActionResult
    }
}
```

5.10 Aula 36 - Conectando Back e Front

5.10.1 Configurando o CORS

Program.cs

```
builder.Services.AddCors(options =>
{
    options.AddPolicy(name: "MainPolicy",
        policy =>
        {
            policy
                .AllowAnyHeader()
                .AllowAnyOrigin()
```

```
                .AllowAnyMethod();
        });
});
```

Program.cs

```
builder.Services.AddScoped<RepoExemploContext>();
if (env.IsDevelopment())
    builder.Services.AddSingleton<IRepository<Mensagem>,
FakeMessageRepository>(); //Mudado para Singleton
else if (env.IsProduction())
    builder.Services.AddTransient<IRepository<Mensagem>,
MessageRepository>();

builder.Services.AddSingleton<ICepService>(p => new CepService(url));

builder.Services.AddTransient<CpfService>();

var app = builder.Build();

app.UseCors(); // Usando Cors
```

MessageController.cs

```
[ApiController]
[Route("message")]
public class MesssageController : ControllerBase
{
    [HttpGet]
    [EnableCors("MainPolicy")] // Configura CORS para este controlador
    public async Task<ActionResult<IEnumerable<Mensagem>>> GetAll(
        [FromServices]IRepository<Mensagem> repo
    )
```

5.10.2 Consumindo API de mensagem

Message.ts

```
export interface Message
{
    id: string;
    texto: string;
    horario: Date;
}
```

app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { MessageService } from './message-service.service'
import { Message } from './Message';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
```

```
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit
{
  data: Message[] = [];
  constructor(private service: MessageService) { }

  ngOnInit(): void
  {
    this.service.getAll()
      .subscribe(x =>
      {
        let list: Message[] = []
        x.forEach(m =>
        {
          list.push(m)
        })
        this.data = list;
        console.log(this.data)
      })
  }
}
```

app.component.html

```
<table>
  <thead>
    <th>Message</th>
    <th>Horario</th>
  </thead>
  <tr *ngFor="let message of data">
    <td>{{message.texto}}</td>
    <td>{{message.horario}}</td>
  </tr>
</table>
```

5.10.3 Enviando um objeto no corpo da requisição

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Message } from './Message';

@Injectable({
  providedIn: 'root'
})
export class MessageService {

  constructor(private http: HttpClient) { }

  getAll()
  {
    return this.http.get<Message[]>("http://localhost:5123/message/");
  }

  add(message: Message)
  {
    return this.http.post("http://localhost:5123/message/", message);
  }
```

```
21  }
```

app.comando.ts

```
1   import { Component, OnInit } from '@angular/core';
2   import { MessageService } from './message-service.service'
3   import { Message } from './Message';
4
5   @Component({
6     selector: 'app-root',
7     templateUrl: './app.component.html',
8     styleUrls: ['./app.component.css']
9   })
10  export class AppComponent implements OnInit
11  {
12    message: string = ""
13    data: Message[] = [];
14    constructor(private service: MessageService) { }
15
16    ngOnInit(): void
17    {
18      this.update();
19    }
20
21    update()
22    {
23      this.service.getAll()
24        .subscribe(x =>
25        {
26          let list: Message[] = []
27          x.forEach(m =>
28          {
29            list.push(m)
30          })
31          this.data = list;
32          console.log(this.data)
33        })
34    }
35
36    add()
37    {
38      this.service.add({
39        id: "0",
40        horario: new Date(),
41        texto: this.message
42      }).subscribe(x =>
43      {
44        this.update();
45      })
46    }
47  }
```

MessageController.cs

```
1   using backend.Model;
2   using Microsoft.AspNetCore.Cors;
```

```
using Microsoft.AspNetCore.Mvc;

namespace ProjetoWeb.Controllers;

[ApiController]
[Route("message")]
public class MesssageController : ControllerBase
{
    [HttpGet]
    [EnableCors("MainPolicy")]
    public async Task<ActionResult<IEnumerable<Mensagem>>> GetAll(
        [FromServices]IRepository<Mensagem> repo
    )
    {
        var result = await repo.Filter(x => true);
        return result;
    }

    [HttpPost]
    [EnableCors("MainPolicy")]
    public async Task<ActionResult> Add(
        [FromBody]Mensagem message,
        [FromServices]IRepository<Mensagem> repo
    )
    {
        repo.Add(message);
        return Ok();
    }
}
```

app.component

```
<table>
  <thead>
    <th>Message</th>
    <th>Horario</th>
  </thead>
  <tr *ngFor="let message of data">
    <td>{{message.texto}}</td>
    <td>{{message.horario}}</td>
  </tr>
</table>

<input [(ngModel)]="message">
<button (click)="add()">Postar</button>
```

5.11 Aula 37 - Um exemplo completo com salvamento de imagens

Abaixo um código SQL para gerar o banco e um script em powershell para gerar um exemplo completo. Analise as estruturas e técnicas empregadas:

```
use master
go

if exists(select * from sys.databases where name = 'FullExample')
    drop database FullExample
go
```

```
create database FullExample
go

use FullExample
go

create table ImageData(
    ID int identity primary key,
    Photo varbinary(MAX) not null
);
go

create table Location(
    ID int identity primary key,
    Nome varchar(60) not null,
    Photo int references ImageData(ID) null
);
go
```

```
clear
"Rode o script do banco de dados antes de dar ENTER e seguir. Também tenha certeza
que não terá problemas com o Proxy do Nuget."
Read-Host

mkdir FullExample
cd FullExample

mkdir Model
cd Model
dotnet new classlib
rm Class1.cs
"DataSource:"
$dataSource = Read-Host
$initialCatalog = "FullExample"
$strconn = "Data Source=" + $dataSource + ";Initial Catalog=" + $initialCatalog +
";Integrated Security=True;TrustServerCertificate=true"
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet tool install --global dotnet-ef
dotnet ef dbcontext scaffold $strconn Microsoft.EntityFrameworkCore.SqlServer --force
cd ..

mkdir DTO
cd DTO
dotnet new classlib
rm Class1.cs
ni LocationDTO.cs
sc LocationDTO.cs "namespace DTO;

public class LocationDTO
{
    public string Name { get; set; }
    public string ImgPath { get; set; }
}"
cd ..

mkdir Back
cd Back
```

```
dotnet new webapi
dotnet add reference ../Model/Model.csproj
dotnet add reference ../DTO/DTO.csproj
rm WeatherForecast.cs
cd Controllers
rm WeatherForecastController.cs
ni ImageController.cs
sc ImageController.cs "using Microsoft.AspNetCore.Cors;
using Microsoft.AspNetCore.Mvc;

using Model;

[ApiController]
[Route(""img"")]
[EnableCors(""MainPolicy"")]
public class ImageController : ControllerBase
{
    [HttpGet(""{code}"")]
    public async Task<ActionResult> Get(
        string code,
        [FromServices]IRepository<ImageDatum> repo
    )
    {
        if (int.TryParse(code, out int id))
        {
            var query = await repo.Filter(im => im.Id == id);
            var img = query.FirstOrDefault();

            if (img is null)
                return NotFound();

            return File(img.Photo, ""image/jpeg"");
        }

        return BadRequest(""code needs to be a integer."");
    }

    [HttpPost]
    [DisableRequestSizeLimit]
    public async Task<ActionResult<string>> Post(
        [FromServices]IRepository<ImageDatum> repo
    )
    {
        var files = Request.Form.Files;
        if (files is null || files.Count == 0)
            return BadRequest();
        var file = Request.Form.Files[0];

        if (file.Length < 1)
            return BadRequest();

        using MemoryStream ms = new MemoryStream();
        await file.CopyToAsync(ms);
        var data = ms.GetBuffer();

        var img = new ImageDatum();
        img.Photo = data;
        await repo.Add(img);

        var code = img.Id.ToString();
```

```
        return Ok(code);
    }
}"
ni LocationController.cs
sc LocationController.cs "using Microsoft.AspNetCore.Cors;
using Microsoft.AspNetCore.Mvc;

using DTO;
using Model;

[ApiController]
[Route(""location"")]
[EnableCors(""MainPolicy"")]
public class LocationController : ControllerBase
{
    [HttpGet]
    public async Task<ActionResult<List<LocationDTO>>> Get(
        [FromServices] IRepository<Location> repo,
        string search = """"
    )
    {
        var query = await repo.Filter(x => x.Nome.Contains(search));
        var locations = query
            .Select(l => new LocationDTO()
            {
                ImgPath = l.Photo?.ToString() ?? """",
                Name = l.Nome
            })
            .ToList();
        return Ok(locations);
    }

    [HttpPost]
    public async Task<ActionResult> Post(
        [FromBody] LocationDTO obj,
        [FromServices] IRepository<Location> repo
    )
    {
        Location newData = new Location();
        newData.Nome = obj.Name;
        newData.Photo = string.IsNullOrEmpty(obj.ImgPath) ?
            null : int.Parse(obj.ImgPath);

        await repo.Add(newData);
        return Ok();
    }
}"
cd ..
ni IRepository.cs
sc IRepository.cs "using System.Linq.Expressions;

public interface IRepository<T>
{
    Task Add(T obj);
    Task<List<T>> Filter(Expression<Func<T, bool>> condition);
}"
ni ImageRepository.cs
sc ImageRepository.cs "using System.Linq.Expressions;
using Microsoft.EntityFrameworkCore;
```

```
using Model;

public class ImageRepository : IRepository<ImageDatum>
{
    private FullExampleContext ctx;
    public ImageRepository(FullExampleContext ctx)
        => this.ctx = ctx;

    public async Task Add(ImageDatum obj)
    {
        await ctx.ImageData.AddAsync(obj);
        await ctx.SaveChangesAsync();
    }

    public async Task<List<ImageDatum>> Filter(Expression<Func<ImageDatum, bool>>
condition)
    {
        var query = ctx.ImageData.Where(condition);
        return await query.ToListAsync();
    }
}"
ni LocationRepository.cs
sc LocationRepository.cs "using System.Linq.Expressions;
using Microsoft.EntityFrameworkCore;

using Model;

public class LocationRepository : IRepository<Location>
{
    private FullExampleContext ctx;
    public LocationRepository(FullExampleContext ctx)
        => this.ctx = ctx;

    public async Task Add(Location obj)
    {
        await ctx.Locations.AddAsync(obj);
        await ctx.SaveChangesAsync();
    }

    public async Task<List<Location>> Filter(Expression<Func<Location, bool>>
condition)
    {
        var query = ctx.Locations.Where(condition);
        return await query.ToListAsync();
    }
}"
sc Program.cs "using Model;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(opt =>
{
   opt.AddPolicy(""MainPolicy"", cors =>
   {
        cors
            .AllowAnyHeader()
            .AllowAnyMethod()
            .AllowAnyOrigin();
   });
});
```

```
builder.Services.AddScoped<FullExampleContext>();
builder.Services.AddTransient<IRepository<Location>, LocationRepository>();
builder.Services.AddTransient<IRepository<ImageDatum>, ImageRepository>();

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.UseCors();

app.Run();
"
start powershell "dotnet watch"
cd ..
clear

"Informe a url do backend (ex http(S?)://localhost:(Porta) sem a barra final"
$url = Read-Host

ng new Front --routing true --style css --skip-git true
cd Front
ng add @angular/material
ng g component uploader --skip-tests
ng g component new-location-page --skip-tests
ng g component locations-page --skip-tests
ng g service location --skip-tests
ng g component location-card --skip-tests
cd src
cd app
sc app.module.ts "
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http'

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { UploaderComponent } from './uploader/uploader.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { MatButtonModule } from '@angular/material/button';
import { NewLocationPageComponent } from './new-location-page/new-location-
page.component';
import { LocationsPageComponent } from './locations-page/locations-page.component';
import { LocationCardComponent } from './location-card/location-card.component';
import { MatInputModule } from '@angular/material/input';
import { MatFormFieldModule } from '@angular/material/form-field';
```

```
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent,
    UploaderComponent,
    NewLocationPageComponent,
    LocationsPageComponent,
    LocationCardComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule,
    BrowserAnimationsModule,
    MatButtonModule,
    MatInputModule,
    MatFormFieldModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

"
ni Location.ts
sc Location.ts "export interface Location
{
    name: string;
    imgPath: string;
}"
sc location.service.ts ("import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Location } from './Location';

@Injectable({
  providedIn: 'root'
})
export class LocationService {

  constructor(private http: HttpClient) { }

  add(location: Location)
  {
    return this.http.post(""" + $url + "/location"", location)
  }

  all()
  {
    return this.http.get<Location[]>(""" + $url + "/location"")
  }

  seach(query: string)
  {
    return this.http.get<Location[]>(""" + $url + "/location?search="" + query)
  }
}
")
sc app-routing.module.ts "
```

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { NewLocationPageComponent } from './new-location-page/new-location-
page.component';
import { LocationsPageComponent } from './locations-page/locations-page.component';

const routes: Routes = [
  { path: '', component: LocationsPageComponent },
  { path: 'add', component: NewLocationPageComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
"
sc app.component.html "<router-outlet></router-outlet>"
cd uploader
sc uploader.component.html "
<input type=""file"" #file placeholder=""Choose
file"" (change)=""uploadFile(file.files)"" style=""display:none;"">
<button mat-raised-button color=""primary"" (click)=""file.click()"">Carregar
imagem</button>
"
sc uploader.component.ts ("import { HttpClient } from '@angular/common/http';
import { Component, EventEmitter, OnInit, Output } from '@angular/core';

@Component({
  selector: 'app-uploader',
  templateUrl: './uploader.component.html',
  styleUrls: ['./uploader.component.css']
})
export class UploaderComponent implements OnInit {
  progress: number = 0;
  message: string = """";
  @Output() public onUploadFinished = new EventEmitter<any>();

  constructor(private http: HttpClient) { }
  ngOnInit() {
  }
  uploadFile = (files: any) => {
    if (files.length === 0) {
      return;
    }
    let fileToUpload = <File>files[0];
    const formData = new FormData();
    formData.append('file', fileToUpload, fileToUpload.name);

    this.http.post('" + $url + "/img', formData)
      .subscribe(result =>
      {
        this.onUploadFinished.emit(result)
      });
  }
}
")
cd ..
cd new-location-page
sc new-location-page.component.ts "import { Component } from '@angular/core';
```

```
import { FormControl } from '@angular/forms';
import { LocationService } from '../location.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-new-location-page',
  templateUrl: './new-location-page.component.html',
  styleUrls: ['./new-location-page.component.css']
})
export class NewLocationPageComponent {

  constructor(private service: LocationService, private route: Router) {}

  emailFormControl = new FormControl('', []);
  imageCode = """
  title = """

  onImageUpdate(code: any)
  {
    this.imageCode = code;
  }

  add()
  {
    this.service.add({
      name: this.title,
      imgPath: String(this.imageCode)
    })
    .subscribe(result =>
      {
        this.route.navigate([""/""])
      }
    )
  }
}
"
sc new-location-page.component.html "<div style=""display: flex; align-items: center;
flex-direction: column; justify-content: center; gap: 20px; height: 100%;"">

    <div>
        <mat-label>Adicionar nova localização</mat-label>
    </div>

    <div>
        <mat-form-field>
            <mat-label>Titulo</mat-label>
            <input type=""text"" matInput [formControl]=""emailFormControl""
 placeholder=""Lugar bonito..."" [(ngModel)]=""title"">
        </mat-form-field>
    </div>

    <div>
        <app-uploader (onUploadFinished)=""onImageUpdate(`$event)""></app-uploader>
    </div>

    <div>
        <button mat-raised-button color=""primary"" (click)=""add()"">Salvar</button>
    </div>

</div>
```

```
"
cd ..
cd locations-page
sc locations-page.component.html "
<a href=""/add"">Adicionar Novo Local...</a>

<li *ngFor=""let location of locations"">
    <app-location-card [location]=""location""></app-location-card>
</li>
"
sc locations-page.component.ts ("import { Component, OnInit } from '@angular/core';
import { Location } from '../Location';
import { LocationService } from '../location.service';

@Component({
  selector: 'app-locations-page',
  templateUrl: './locations-page.component.html',
  styleUrls: ['./locations-page.component.css']
})
export class LocationsPageComponent implements OnInit {
  locations: Location[] = [];

  constructor (private service: LocationService) {}

  ngOnInit(): void {
    this.service.all()
      .subscribe(list =>
      {
        console.log(list)
        var newList: Location[] = []
        list.forEach(element => {
          newList.push({
            name: element.name,
            imgPath: """ + $url + "/img/"" + element.imgPath
          })
        });
        this.locations = newList
      })
  }
}")
cd ..
cd location-card
sc location-card.component.ts "
import { Component, Input } from '@angular/core';
import { Location } from '../Location';

@Component({
  selector: 'app-location-card',
  templateUrl: './location-card.component.html',
  styleUrls: ['./location-card.component.css']
})
export class LocationCardComponent {
  @Input() location: Location = { name: """", imgPath: """"};
}
"
sc location-card.component.html "
<div style=""display: flex; align-items: center; flex-direction: column;"">
    <div>
        {{location.name}}
    </div>
```

```
    <div>
        <img src={{location.imgPath}} width=""300px""/>
    </div>
</div>
"
cd ..
cd ..
cd ..
start powershell "npm start"
start chrome http://localhost:4200/
```

5.12 Aula 38 - Segurança de Sistemas

5.12.1 Criptografia Simétrica

5.12.2 Hashing

5.12.3 Hash vs Criptografia

5.12.4 Armazenamento de Senhas

5.12.5 Salting

5.12.6 SlowHash

5.12.7 BCrypt

5.12.8 Eavesdroping & Man in The Middle

5.12.9 Criptografia Assimétrica e RSA

5.12.10 Assinatura Digital

5.12.11 Sistema de Tokens

5.12.12 Json Web Tokens

5.12.13 Comunicação Criptografada

5.12.14 HTTP/HTTPS/SSL/TSL

5.13 Aula 39 - Sistemas de Autentificação

5.13.1 Implementando um JWT e publicando no Nuget

5.13.2 Canal Criptografado

5.13.3 Fazendo um sistema de Login

5.13.4 Armazenando Tokens

5.13.5 Validações via Email

5.14 Aula 40 - Desafio 12

6 5 - C# para Desenvolvedores Java

6.1 Objetivos

1. Realizar as operações básicas de programação estruturada e orientada a objetos do Java em C# (2 horas).
2. Compreender as diferenças de comportamento entre C# e Java (2 horas).
3. Utilizar métodos iteradores (4 horas).
4. Utilizar métodos de Extensão (2 horas).
5. Utilizar genéricos em C# (2 horas).
6. Utilizar programação funcional em C# (12 horas).
7. Compreender o funcionamento de LINQ em termos de métodos iteradores, de extensão, genéricos e programação funcional (4 horas).
8. Utilizar LINQ na resolução de problemas (8 horas).
9. Utilizar Pattern Matching em C# (2 horas).
10. Utilizar Programação Orientada a Eventos em C# (2 horas).
11. Utilizar Programação Assíncrona em C# (4 horas).
12. Ser capaz de projetar software usando os principais padrões de projeto (24 horas).
13. Utilizar Programação Reflexiva (12 horas).
14. Utilizar o Entity Framework (4 horas).
15. Ser capaz de criar uma API utilizando .NET (16 horas).

Aulas relevantes:

Aula 11 - Design Avançado de Objetos
Aula 12 - Coleções e Introdução a Language Integrated Query (LINQ)
Aula 13 - Programação Funcional e LINQ
Aula 14 - LINQ Avançado
Aula 16 - Pattern Matching e Sobrescrita de Operadores
Aula 17 - Orientação a Eventos
Aula 19 - Programação Paralela e Assíncrona
Aula 22 - Padrões de Projeto Criacionais
Aula 23 - Padrões de Projeto Comportamentais
Aula 24 - Padrões de Projeto Estruturais
Aula 25 - Padrões de Projeto Adicionais
Aula 27 - Programação Genérica e Reflexiva, Expressões e Atributos
Aula 28 - Conexão com Banco de Dados, Object-Relational Mapping e Entity Framework
Aula 35 - Introdução a Desenvolvimento Backend