BOSCH
Invented for life

C# Básico

Docupedia Export

Author: Silio Leonardo (CtP/ETS)
Date: 28-Feb-2023 12:30

Table of Contents

1 Aulas

2 Overview

Neste curso de C# Básico você aprenderá sobre operações básicas, variáveis, estruturas de controle de fluxo, orientação a objetos, programação funcional, gerenciamento básico de memória, LINQ e a tecnologia .NET como um todo. Este curso não foi feito para iniciantes em lógica de programação que ainda não sabem programar, mas sim para pessoas que sabem o básico em qualquer linguagem de programação que desejam aprender a linguagem de programação C# em seus conceitos básicos com profundidade.

3 Competências

As Competências a serem desenvolvidas ao longo do curso são:

1. Compreender o .NET como tecnologia
2. Capacidade de criar e executar aplicativos C#
3. Usar operações básicas da linguagem
4. Manipular texto
5. Compreender sistema de tipos e variáveis
6. Usar estruturas de controle de fluxo
7. Compreender e aplicar o pilar OO da Abstração
8. Compreender e aplicar o pilar OO do Encapsulamento
9. Compreender a organização da memória
10. Gerenciar dados e ponteiros
11. Compreender os tipos genéricos
12. Usar Listas
13. Compreender e aplicar o pilar OO da Herança
14. Compreender e aplicar o pilar OO do Polimorfismo
15. Resolução de Problemas usando o C# e a OO
16. Usar Namespaces e construir aplicações multi-arquivo
17. Entender o básico sobre tratamento de erros
18. Compreender o funcionamento de um csproj
19. Compreender e usar classes estáticas
20. Compreender e usar Enums
21. Compreender e usar tipos anuláveis e propagação nula
22. Compreender e usar System.IO no acesso de arquivos
23. Compreender Iteradores
24. Compreender Coleções
25. Compreender Métodos Iteradores
26. Compreender Métodos de Extensão
27. Compreender Inferência
28. Compreender conceitos de programação funcional e delegados
29. Compreender a tecnologia LINQ
30. Usar agrupamentos, ordenações, seleções e filtros usando LINQ
31. Compreender e usar objetos anônimos

4 Aula 1 - Introdução ao .NET

4.1 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#.

4.2 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

4.3 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**.

4.4 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.

4.5 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.

4.6 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
JavaScirpt	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.

4.7 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.

4.8 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.

4.9 dotnet CLI

Como já comentado, o dotnet CLI é um programa que te ajuda a desenvolver em C#. Apartir 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).

4.10 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.

4.11 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
14  ForegroundColor = ConsoleColor.Red;
15  WriteLine(texto);
```

4.12 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:

```
1   byte b1 = 0; // Inteiro com 2^8 valores de 0 a 255 (naturalmente sem
    sinal).
2   sbyte b2 = 0; // Inteiro com sinal (signed) com 2^8 valores de -128 a 127.
3
4   short s1 = 0; // Inteiro com 2^16 valores de -32'768 a 32'767.
5   ushort s2 = 0; // Inteiro sem sinal (unsigned) com 2^16 valores de 0 a
    65'535.
6
7   int i1 = 0; // Inteiro com 2^32 valores de -2'147'483'648 a 2'147'483'647.
8   uint i2 = 0; // Inteiro sem sinal (unsigned) com 2^32 valores de 0 a
    4'294'967'295.
9
10  long l1 = 0; // Inteiro com 2^64 valores de -9'223'372'036'854'775'808 a
    9'223'372'036'854'775'807.
11  ulong l2 = 0; // Inteiro sem sinal (unsigned) com 2^64 valores de 0 a
    18'446'744'073'709'551'615.
12
13  nint n1 = 0; // int se o sistema for de 32 bits, longo se o sistema for de
    64 bits, ou seja, um inteiro nativo.
14  nuint n2 = 0; // O mesmo que o nint, porém, podendo ser uint ou ulong.
15
16  char c = 'a'; // Caracter.
17  string s = "texto"; // Cadeia de caracteres/texto.
18
19  bool b = true; // Booleano, ou seja, verdadeiro ou falso (true/false).
20
21  float f = 0f; // Tipo de ponto flutuante, isso é, armazena números com
    virgula até uma determinada potência com uma quantidade limitada de casas
    decimais.
22  double d = 0.0; // Semelhante ao float, porém com o dobro de
    armazenamento, podendo ter mais casas decimais e representar maiores
    números.
23  decimal m = 0m; // Tipo desenvolvido para guardar dinheiro. Dobro do
    armazenamento do double. Menor range de valores mas muito mais casas
    decimais.
24
25  var x = 0; // Descobre automaticamente o tipo da variável de forma
    Implícita, sem precisar escrever.
26  // x = "oi"; // Mas não permite a mudança do tipo da variável.
27
28  dynamic y = 0; // Desabilita verificações e guarda qualquer tipo de dado.
29  y = "oi"; // Pode mudar de tipo ao longo do programa, porém, pouco
    utilizado.
30
31  int i3 = int.MaxValue; // Obtém maior valor possível para int.
32  int i4 = 0b_0000_0101; // Valor binários.
```

```
33
34  //int i4 = 5; // Erro, não é possível declarar duas variáveis com mesmo
    nome.
35  i4 = 6; // Ok.
36
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.

4.13 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.
```

4.14 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.
```

5 Aula 2 - Escovando bits com C#

5.1 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	27	26	25	24	23	22	21	20	
resultado	128	64	32	16	8	4	2	1	
contribuição	128	0	32	0	0	4	0	1	175

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_____	196	não	próximo bit
3	111____	228	sim	reverte
4	110____	196	não	próximo bit
5	1101___	204	sim	reverte
6	1100___	196	não	próximo bit
7	11001__	200	não	próximo bit
8	110011_	202	não	próximo bit
9	1100111	203	sim	reverte
10	1100110	202	não	fim

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.

5.2 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

```
using static System.Console;

int idade = int.Parse(ReadLine());
switch (idade)
{
    case 18:
        WriteLine("É maior de idade.");
    break;

    case 19:
        WriteLine("Está ficando velho!");
    goto case 18;

    case 16:
    case 17:
        WriteLine("Não é maior de idade. Mas em alguns países já pode dirigir.");
    break;

    default:
        WriteLine("Menor de idade.");
    break;
}
```

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

```
using static System.Console;

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

int fatorial = 1;

while (n > 1)
{
    fatorial *= n;
    n--;
}
using static System.Console;

int numeroSecreto = 540;
int numero;

do
{
    WriteLine("Tente adivinhar o número secreto...");
```

```
21      numero = int.Parse(ReadLine());
22
23      if (numero > numeroSecreto)
24          WriteLine("O numero secreto é menor");
25      else if (numero < numeroSecreto)
26          WriteLine("O numero secreto é maior");
27
28  } 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
5   WriteLine("Digite 50 valores");
6   for (int i = 0; i < vetor.Lenth; i++)
7   {
8       vetor[i] = int.Parse(ReadLine());
9   }
```

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.

5.3 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:

```
1   int value = modulo(-3);
2
3   int modulo(int i)
4   {
5       if (i < 0)
6           return -i;
7       return i;
8   }
9   using static System.Console;
```

```
10
11  WriteLine("Fatorial de qual valor você irá querer?");
12  int n = int.Parse(ReadLine());
13
14  WriteLine("O resultado é: " + fatorial(n));
15
16  int fatorial(int n)
17  {
18      if (n < 2)
19          return 1;
20
21      return n * fatorial(n - 1);
22  }
23  using static System.Console;
24
25  mostreUmTexto("Hello World!");
26  mostreUmTexto(); // mostrará 'Olá mundo'
27
28  void mostreUmTexto(string texto = "Olá Mundo!") // não retorna nada
29  {
30      WriteLine("Texto:");
31      WriteLine(texto);
32      //return; // pode ter um return aqui, mas o mesmo não é obrigatório
33  }
```

5.4 Exercícios Propostos 1

- a. Converta os seguintes valores:
  - a) 101 para binário
  - b) 102 para binário
  - c) 103 para binário
  - d) 254 para binário
  - e) 11111100 para decimal
  - f) 10000000 para decimal
  - e) 11000001 para decimal
- a. Escreva uma função C# que converta um número decimal para binário (retornando uma string).
- a. Faça um programa que dado dois números dados pelo usuário a e b, desenhe na tela, em binário, o resultado de a << b;

6 Aula 3 - Desafio 1

6.1 Desafio 1.1

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
}
```

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

6.2 Desafio 1.2

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.

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

7 Aula 4 - Strings + Desafio 2

7.1 Trabalhando com Strings

Algumas operações com strings são interessantes e podemos aprende-las para tornar algumas tarefas mais fáceis. 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.

7.2 Desafio 2

Como segundo desafio do curso vamos implementar a função eval que recebe uma string com uma equação e retorna o resultado da mesma:

- eval("1 + 2 * 3"), deve retornar 7
- eval("10 / 4"), deve retornar 2.5
- eval("batata frita"), deve retornar float.NaN (not a number/não é um número)
- eval("100-3"), deve retornar float.NaN
- eval("10 10 + 4"), deve retornar float.NaN

Você pode considerar que todos os números, símbolos vem com espaço, caso não vir, como no último exemplo, você pode retornar NaN.

```
1  float eval(string eq)
2  {
3      // Implemente aqui
4  }
```

Lembre-se de respeitar a precedência: Soma e Subtração vem depois de Multiplicação e Divisão.

> *Desafio Opcional: Trate parêntesis também, assim, o que está no parêntesis deve ser calculado antes do que está fora.*

8 Aula 5 - Orientação a Objetos básica

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
2  {
3      public string Nome;
4      public string Endereco;
5      public long Cpf;
6
7      public void AtualizarEndereco(string novoEndereco)
8      {
9          Endereco = novoEndereco;
10     }
11 }
```

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.

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.

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();
```

```
4   Horario h2 = new Horario();
5
6   h1.Adicionar(20, 40, 30); //h1 = 20:40:30
7   h2.Adicionar(20, 30); //h2 = 00:20:30
8
9   h1.Adicionar(h2); //h1 = 21:01:00, h2 inalterado
10  h2.Adicionar(h1); //h2 = 00:20:30 + 21:01:00 = 21:21:30, h1 inalterado
11
12  WriteLine(h2.Formatar()); //21:21:30
13
14  public class Horario
15  {
16      // Valor inicial é 00:00:00
17      public int Hora = 0;
18      public int Minuto = 0;
19      public int Segundo = 0;
20
21      public void Adicionar(int segundos, int minutos, int horas)
22      {
23          Segundo += segundos;
24          if (Segundo > 59)
25          {
26              minutos += Segundo / 60;
27              Segundo = Segundo % 60;
28          }
29
30          Minuto += minutos;
31          if (Minuto > 59)
32          {
33              horas += Minuto / 60;
34              Minuto = Minuto % 60;
35          }
36
37          Hora += horas;
38          if (Hora > 23)
39          {
40              Hora = Hora % 24;
41          }
42      }
43
44      public void Adicionar(int segundos, int minutos)
45      {
46          Adicionar(segundos, minutos, 0);
47      }
48
49      public void Adicionar(int segundos)
50      {
51          Adicionar(segundos, 0, 0);
52      }
53
54      public void Adicionar(Horario horario)
55      {
56          Adicionar(horario.Hora, horario.Minuto, horario.Segundo);
57      }
58
59      public string Formatar()
60      {
61          return $"{Hora}:{Minuto}:{Segundo}";
62      }
```

```
63  }
```

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 = 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 segundos, int minutos, int horas)
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(segundos, minutos, horas);
32      }
33
34      public void Adicionar(int segundos, int minutos, int horas)
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 segundos, int minutos)
58      {
59          Adicionar(segundos, minutos, 0);
60      }
61
62      public void Adicionar(int segundos)
63      {
64          Adicionar(segundos, 0, 0);
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  }
```

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
```

```
{
    public string Login;
    public string Senha;

    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;
```

```
10      private string senha;
11
12      public void SetSenha(string value)
13      {
14          string senhaCriptografada = "";
15          // Criptografa o value e guarda na variável senhaCriptografada
16          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 segundos, int minutos, int horas)
10     {
11         Adicionar(segundos, minutos, horas);
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 segundos, int minutos, int horas)
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 segundos, int minutos)
71      {
72          Adicionar(segundos, minutos, 0);
73      }
74
75      public void Adicionar(int segundos)
76      {
77          Adicionar(segundos, 0, 0);
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:

```
// Tradicional
cliente.SetLogin("@" + cliente.GetLogin());

// Forma C#
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:

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

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

```
public class Horario
{
    //É possível ler Hora, Minuto e Segundo, mas alterá-la só com a função Adicionar, Construtor ou internalmente na classe
    public int Hora { get; private set; } = 0;
    public int Minuto { get; private set; } = 0;
    public int Segundo { get; private set; } = 0;

    public Horario() { }

    public Horario(int segundos, int minutos, int horas)
    {
        Adicionar(segundos, minutos, horas);
    }

    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);
    }

    public string Formatar()
    {
        return $"{Hora}:{Minuto}:{Segundo}";
    }
}
```

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

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:

```
public class Horario
{
    public int Hora { get; private set; } = 0;
    public int Minuto { get; private set; } = 0;
    public int Segundo { get; private set; } = 0;

    public Horario() { }

    public Horario(int segundos, int minutos, int horas)
        => Adicionar(segundos, minutos, horas);

    public 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;
```

```
32          }
33      }
34
35      public void Adicionar(int segundos, int minutos)
36          => Adicionar(segundos, minutos, 0);
37
38      public void Adicionar(int segundos)
39          => Adicionar(segundos, 0, 0);
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  }
```

8.7 Exercício Proposto 2

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.

9 Aula 6 - Desafio 3

9.1 Desafio 3

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.

10 Aula 7 - Gerenciando dados e memória

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:

```
1
```

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.

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;
```

```
aluno2.Idade++;
WriteLine(aluno.Idade); // 16
WriteLine(aluno2.Idade); // 17

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

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:

```
int numero = 76;
AlterarNumero(numero);
// Aqui número vale 76, seu valor foi copiado e não foi alterado

Aluno aluno = new Aluno();
aluno.Nome = "Pamella";
AlterarNome(aluno);
// Aqui o nome do Aluno é Gilmar, foi passado a referência do objeto que
foi alterado dentro da função

void AlterarNumero(int valor)
{
    valor = valor + 1;
}

void AlterarNome(Aluno aluno)
{
    aluno.Nome = "Gilmar";
}

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

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:

```
Aluno aluno = new Aluno("Gilmar", 25);

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.

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; }
}
```

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.

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.

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

```
using static System.Console;

LinkedList clientes = new LinkedList();

Cliente cliente1 = new Cliente();
cliente1.Nome = "Gilmar";
clientes.Add(cliente1);

Cliente cliente2 = new Cliente();
cliente2.Nome = "Pamella";
clientes.Add(cliente2);

// Se Get retornar null, Nome deve retornar null ao invés de estourar um erro, se Nome retornar null deve-se substituir pelo valor padrão
// 'Cliente não encontrado'
var result = clientes.Get(1)?.Nome ?? "Cliente não encontrado";
WriteLine(result);

// Cadastrando uma quantidade variável de clientes
string input = ReadLine();
while (input != "")
{
    Cliente newClient = new Cliente();
    newClient.Nome = input;
    newClient.Cpf = long.Parse(ReadLine());
    clientes.Add(newClient);

    input = ReadLine();
}

WriteLine("Procure um  cpf de cliente pelo nome:");
input = ReadLine();
for (int i = 0; i < clientes.Count; i++)
{
    var pesquisa = clientes.Get(i);
    if (pesquisa?.Nome == input)
    {
        WriteLine(pesquisa.Cpf);
        break;
    }
}

// Classe cliente a qual queremos armazenas
public struct Cliente
{
    public string Nome { get; set; }
    public long Cpf { get; set; }
}

// Um nó representa um valor na lista com um ponteiro para o próximo valor
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;
99          for (int i = 0; i < index && crr != null; i++)
100             crr = crr.Next;
101
102         // Se crr for nulo, retorna nulo, caso contrário retorna seu Value
103         return crr?.Value;
104     }
105 }
```

11 Aula 8 - Listas + Desafio 4

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:

```
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      WriteLine(list.Get(i));
11  }
12
13  public class ListInt
14  {
15      private int pos = 0;
16      private int[] vetor = new int[10];
17
18      public void Add(int value)
19      {
20          int len = vetor.Length;
21          // Vetor estouraria, vamos aumentá-lo
22          if (pos == len)
23          {
24              // Criamos um vetor maior
25              int[] newVetor = new int[2 * len];
26
27              // Copiamos
28              for (int i = 0; i < pos; i++)
29                  newVetor[i] = vetor[i];
30
31              // Jogamos a referência antiga fora e agora o ponteiro 'vetor'
    irá apontar para um novo vetor
32              vetor = newVetor;
33          }
34
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  }
```

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;
}
```

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)
```

```
39      {
40          int len = vetor.Length;
41          // 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  }
```

11.4 Desafio 4.1

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:

- Push: Adiciona um valor ao topo da pilha
- Pop: Remove e retorna o valor no topo da pilha
- Peek: Retorna, sem remover, o valor no topo da pilha
- Count: Tamanho da pilha

11.5 Desafio 4.2

Depois disso implementaremos a Queue - uma fila. Na fila, entram coisas de um lado e se retiram do outro:

- Enqueue: Adiciona um valor no início da fila
- Dequeue: Remove e retorna o valor no final da fila
- Peek: Retorna, sem remover, o valor no final da fila
- Count: Tamanho da fila

12 Aula 9 - Herança e Polimorfismo

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
6  WriteLine(a.Value); // 2
7  WriteLine(a.OtherValue); // Erro: Other Value não existe em A
8
9  WriteLine(b.Value); // 2
10 WriteLine(b.OtherValue); // 3
11
12 public class A
13 {
14     public int Value { get; set; } = 2;
15 }
16
17 public class B : A
18 {
19     public int OtherValue { get; set; } = 3;
20 }
```

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:

```
1  using static System.Console;
2
3  Gato gato = new Gato("Edjalma");
4  Cao cao = new Cao("Cesar");
5
```

```
6  public class Pet
7  {
8      public Pet(string nome)
9      {
10         this.Nome = nome;
11     }
12
13     public string Nome { get; set; }
14 }
15
16 public class Cao : Pet
17 {
18     // A classe Pet precisa de um nome para ser construído. Assim todas as
   suas classes bases precisam
19     // de um construtor equivalente. Caso contrário obtemos um erro. Você
   pode ainda usar ': base(nome)'
20     // para chamar a funcionalidade do construtor da classe mãe (Pet).
21     public Cao(string nome) : base(nome) { }
22
23     public void Latir()
24     {
25         WriteLine("Au!");
26     }
27 }
28
29 public class Gato : Pet
30 {
31     public Gato(string nome) : base(nome) { }
32
33     public void Miar()
34     {
35         WriteLine("Miau!");
36     }
37 }
```

```
1  using static System.Console;
2
3  public class Conta
4  {
5      private decimal saldo;
6
7      public bool Saque(decimal value)
8      {
9          if (value > saldo)
10             return false;
11
12         saldo -= value;
13
14         return true;
15     }
16 }
17
18 public class Debito : Conta
19 {
20     // 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;
```

```
27          }
28
29          this.Minuto += minutos;
30          if (this.Minuto > 59)
31          {
32              horas += this.Minuto / 60;
33              this.Minuto = this.Minuto % 60;
34          }
35
36          this.Hora += horas;
37          if (this.Hora > 23)
38          {
39              this.Hora = this.Hora % 24;
40          }
41      }
42
43      // Semelhante a um valor privado, porém pode ser visto também nas
    classes filhas
44      protected void remover(int segundos, int minutos, int horas)
45      {
46          this.Segundo -= segundos;
47          if (this.Segundo < 0)
48          {
49              minutos -= this.Segundo / 60;
50              this.Segundo = -(this.Segundo % 60);
51          }
52
53          this.Minuto -= minutos;
54          if (this.Minuto < 0)
55          {
56              horas -= this.Minuto / 60;
57              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.

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
36  {
37      protected override void printSuperior()
38      {
39          WriteLine("--------------------");
40      }
41
42      protected virtual void printInferior()
43      {
44          WriteLine("--------------------");
45      }
46  }
```

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:

```
1   public abstract class Language
2   {
3       public abstract string TranslateNewGame();
4       public abstract string TranslateQuit();
5       public abstract string TranslateLoadGame();
6       public abstract string TranslateOptions();
7   }
8
9   public class Engish : Language
10  {
11      public override string TranslateNewGame()
12          => "New Game";
13      public override string TranslateQuit()
14          => "Quit";
15      public override string TranslateLoadGame()
16          => "Load Game";
17      public override string TranslateOptions()
18          => "Options";
19  }
20
21  public class Portuguese : Language
22  {
```

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

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.

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; }
}
```

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)
        {
            game.Play(X, true);
            crr = O;
        }
        else
        {
            game.Play(O, false);
            crr = X;
        }
    } while (!result.XWins && !result.OWins && !result.Tie && result.IsValid);

    if (result.XWins && randomIsX)
        randomWins++;
    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);
138     }
139
140     public PlayResult Play(int i, int j, bool isX)
141     {
142         PlayResult result = new PlayResult();
143         if (game[i + 3 * j] != 0)
144             return result;
145
146         game[i + 3 * i] = isX ? 1 ? 2;
147
148         sbyte value = (sbyte)(isX ? 1 : -1);
149         game[i] += value;
150         game[3 + j] += value;
151         if (i == j)
152             game[7] += value;
```

```
153             if (i + j == 2)
154                 game[8] += value;
155 
156             for (int i = 0; i < 8; i++)
157             {
158                 if (winChecker[i] == 3)
159                     result.XWins = true;
160 
161                 if (winChecker[i] == -3)
162                     result.OWins = true;
163             }
164 
165             if (result.XWins || result.OWins)
166                 return result;
167 
168             playCount++;
169             if (playCount == 9)
170                result.Tie = true;
171 
172             return result;
173         }
174     }
```

13 Aula 10 - Desafio 5

13.1 Desafio 5

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).

14 Aula 11 - Desafio 6

14.1 Desafio 6

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.

15 Aula 12 - Design Avançado de Objetos

15.1 Namespaces, Projetos, Assemblies, Arquivos e Organização

Você pode organizar seu código C# de muitas formas e subdividi-lo como quiser, mas alguns padrões podem ser respeitados para melhorar a sua experiência e deixar seu trabalho mais produtivo. Nesta secção aprenderemos um pouco sobre como fazer isso. Como background iremos fazer uma aplicação para um sistema escolar. Ou seja, um sistema onde o usuário poderá cadastrar alunos, professores, disciplinas e turmas(que são vários alunos cursando uma disciplina dada por um professor). Como não queremos perder os dados quando a aplicação fecha, salvaremos os dados em arquivos e assim produziremos uma biblioteca que faz isso.

No C# podemos separar nossas implementações em vários **Projetos**. Um projeto é basicamente uma pasta com um arquivo de extensão '.csproj' em estrutura XML com tags como pode ser visto a seguir:

nomedoprojeto.csproj

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

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

</Project
```

Dentro de 'PropertyGroup' temos algumas configurações importantes:

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

Existem muitas outras opções que veremos no futuro.

Um projeto quando executado gera um **Assembly** que é um objeto de código que pode ser utilizado por outros ou executado. Tudo que você fizer em um projeto estará incluso no mesmo assembly e você pode importar vários assemblies em um mesmo projeto como veremos mais tarde nesta aula.

Como você já sabe, quando executado um projeto buscamos a função Main/Arquivo Top-Level e o executamos. Porém, é completamente possível termos muitos arquivos no mesmo projeto. Para usar um arquivo no outro você não precisa fazer nada, só usar os elementos de um arquivo no outro. Observe:

Aluno.cs

```
1  public class Aluno
2  {
3      public int Matricula { get; set; }
4      public string Nome { get; set; }
5  }
```

Program.cs

```
1  Aluno aluno = new Aluno();
2
3  aluno.Matricula = 4;
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:

```
1  public class Classe1 { }
2
3  namespace Namespace1
4  {
5      public class Classe2 { }
6
7      namespace Namespace2
8      {
9          public class Classe3 { }
10     }
11
12     namespace Namespace3.Namespace4
13     {
14         public class Classe4 { }
15     }
16 }
17
18 namespace Namespace1.Namespace2
19 {
20     public class Classe5 { }
21 }
```

Usamos a palavra reservada 'using' para acessar os conteúdos fora do Namespace onde ela foi criada. Por exemplo, se estamos fora do Namespace1 você precisa importá-lo para usar a Classe2. Abaixo você verá 5 exemplos de uso da classe Program usando o código acima, e como se trabalha corretamente com os

Namespaces. Relembrando, apesar do exemplo caótico, em geral você só usa um Namespace específico dentro de pastas em um projeto. Veremos como isso se comporta no futuro.

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

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

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

```
1  Classe2 test = new Classe2(); // Error
2
3  namespace Namespace1
4  {
5      public class Test
6      {
7          public static void Main()
8          {
9              Classe2 obj2 = new Classe2(); // OK, Classe2 e este código
   estão dentro do Namespace1
10         }
11     }
12 }
```

```
1  using Namespace1.Namespace3.Namespace4;
2
3  namespace Namespace1; // Todo código abaixo está no Namespace1
4
5  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  }
```

15.2 Usings Globais

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

Usings.cs

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

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

nomedoprojeto.csproj

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

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

obj/NomeDoProjeto.GlobalUsings.g.cs

```
1  // <auto-generated/>
2  global using global::System;
3  global using global::System.Collections.Generic;
4  global using global::System.IO;
5  global using global::System.Linq;
6  global using global::System.Net.Http;
7  global using global::System.Threading;
8  global using global::System.Threading.Tasks;
```

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

```
1  namespace System
2  {
3      public class ClasseA
```

```
    {
        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.

15.3 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;
```

```
2
3   // Sua própria classe de Console personalizada
4   public static class MyConsole
5   {
6       public int? ReadLineInt()
7       {
8           var str = Console.ReadLine();
9
10          // Um código novo para os aventureiros:
11          // Se a conversão é bem sucedida, cria um int i e joga o valor
    convertido para fora da função
12          // TryParse usando a palavar-reservada out. A palavra ainda
    retorna verdadeiro. Caso contrário
13          // nenhum erro estoura e retorna falso
14          if (int.TryParse(str, out int i))
15              return i;
16
17          return null;
18      }
19
20      public void Print(object obj)
21      {
22          // Usando a conversão para string que todo objeto tem
23          var str = obj.ToString();
24          Console.WriteLine(str);
25      }
26  }
```

Usando a classe MyConsole sem using estática:

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

Usando a classe MyConsole com using estática:

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

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

```
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;
    }
}
```

15.4 Enums

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

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

Sua utilização é fácil:

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

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

```
1  using System;
2
3  Key key1 = Key.Ctrl;
4  Key key2 = (Key)(2); // C
5  Key key = key1 | key2; // União de Ctrl e C
6
7  // Código complexo abaixo, analisar com cuidado
8  if ((key & Key.C) > 0 && (key & Key.Ctrl) > 0)
9      Console.WriteLine("Copiar");
10 else if ((key & Key.V) > 0 && (key & Key.Ctrl) > 0)
11     Console.WriteLine("Colar");
12
13 public enum Key : byte
14 {
15     Ctrl = 1,
16     C = 2,
17     V = 4
18 }
```

15.5 Tratamento de Erros

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

```
1  try
2  {
```

```
    // 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.

15.6 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.

15.7 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:

```
operate(new Sum(), 1, 2);
operate(new Sub(), 10, 5);

float operate(Operation op, float a, float b)
    => op.GetResult(a, b);

public class Sum : Operation
{
    public float GetResult(float a, float b)
        => a + b;
}

public class Sub : Operation
{
    public float GetResult(float a, float b)
        => a - b;
}

public interface Operation
{
    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.

15.8 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>
```

```
  </PropertyGroup>

</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

```
namespace DataBase;

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

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;
```

```
26              var path = this.basePath + fileName + ".csv";
27              return path;
28          }
29      }
30
31      // Funções privadas apenas para separar melhor as implementações
32      private List<string> openFile()
33      {
34          List<string> lines = new List<string>();
35          StreamReader reader = null;
36          var path = this.DBPath;
37
38          if (!File.Exists(path))
39              File.Create(path).Close();
40
41          try
42          {
43              reader = new StreamReader(path);
44              // Lê linhas de um arquivo até que ele acabe e preenche uma
    lista com as linhas
45              while (!reader.EndOfStream)
46                  lines.Add(reader.ReadLine());
47          }
48          catch
49          {
50              lines = null; // Falha
51          }
52          finally
53          {
54              reader?.Close(); // Fecha o arquivo, liberando seu uso
55          }
56
57          return lines;
58      }
59
60      private bool saveFile(List<string> lines)
61      {
62          StreamWriter writer = null;
63          bool success = true;
64          var path = this.DBPath;
65
66          if (!File.Exists(path))
67              File.Create(path).Close();
68
69          try
70          {
71              writer = new StreamWriter(path);
72              // Escrever linhas em um arquivo até que a lista acabe
73              for (int i = 0; i < lines.Count; i++)
74              {
75                  var line = lines[i];
76                  writer.WriteLine(line);
77              }
78          }
79          catch
80          {
81              success = false; // Falha
82          }
83          finally
84          {
```

```
85              writer.Close(); // Fecha o arquivo, liberando seu uso
86          }
87          return success;
88      }
89
90      // Retorna uma lista com todos os obejtos
91      public List<T> All
92      {
93          get
94          {
95              var lines = openFile();
96              if (lines is null)
97                  throw new DataCannotBeOpenedException(this.DBPath); //
    Estouramos nosso erro personalizado
98
99              var all = new List<T>();
100             try
101             {
102                 for (int i = 0; i < lines.Count; i++)
103                 {
104                     var line = lines[i];
105                     var obj = new T(); // Só podemos fazer isso por causa
    da restrição genérica where T : new()
106                     var data = line.Split(',',
    StringSplitOptions.RemoveEmptyEntries); // Splita removendo qualquer dado
    vazio
107                     obj.LoadFrom(data); // Só podemos fazer isso porquê
    T : DataBaseObject
108                     all.Add(obj);
109                 }
110             }
111             catch
112             {
113                 throw new ConvertObjectError();
114             }
115             return all;
116         }
117     }
118
119     // Salva uma lista com todos os objetos
120     public void Save(List<T> all)
121     {
122         List<string> lines = new List<string>();
123         for (int i = 0; i < all.Count; i++)
124         {
125             var data = all[i].SaveTo();
126             string line = string.Empty; // Linha começa vazia
127             for (int j = 0; j < data.Length; j++)
128                 line += data[j] + ",";
129             lines.Add(line);
130         }
131
132         if (saveFile(lines))
133             return;
134
135         throw new DataCannotBeOpenedException(this.DBPath);
136     }
137
138     // Variável estática para obter uma instância de DB que salva dados na
    pasta temporária
```

```
139     private static DB<T> temp = null;
140     public static DB<T> Temp
141     {
142         get
143         {
144             if (temp == null)
145                 temp = new DB<T>(Path.GetTempPath());
146             return temp;
147         }
148     }
149 
150     // Variável estática para obter uma instância de DB que salva dados na
    pasta do executável
151     private static DB<T> app = null;
152     public static DB<T> App
153     {
154         get
155         {
156             if (app == null)
157                 app = new DB<T>("");
158             return app;
159         }
160     }
161 
162     // Variável estática para obter uma instância de DB que salva dados em
    uma pasta customizável
163     private static DB<T> custom = null;
164     public static DB<T> Custom
165     {
166         get
167         {
168             if (custom == null)
169                 throw new CustomNotDefinedException();
170             return custom;
171         }
172     }
173 
174     public static void SetCustom(string path)
175         => custom = new DB<T>(path);
176 }
```

DataBase/Exceptions/ConvertObjectError.cs

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

DataBase/Exceptions/ CustomNotDefinedException.cs

```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

```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

```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

```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

```cs
using System.Collections.Generic;

namespace Model;

// Representa um repositório de dados de um tipo T qualquer. Você pode implementar várias vezes para conectar com qualquer tipo de coisa:
// Arquivos, Banco de Dados, Nuvem ou dados estáticos. O interessante é perceber que apenas trocando a implementação, de um objeto para
// outro, trocamos a forma como nossa aplicação se relaciona com dados sem quebrar nada, pois todos os repositórias implementarão as mesmas
// funcionalidades, porém com diferentes comportamentos
public interface IRepository<T>
{
    List<T> All { get; }
    void Add(T obj);
}
```

Model/AlunoFakeRepository.cs

```cs
using System.Collections.Generic;

namespace Model;

// Repositório Fake com uma lista interna. Pode ser usado para testes sem se comprometer em afetar os dados reais da aplicação.
public class AlunoFakeRepository : IRepository<Aluno>
```

```
{
    List<Aluno> alunos = new List<Aluno>();
    public AlunoFakeRepository()
    {
        alunos.Add(new Aluno()
        {
            Nome = "Pamella",
            Idade = 22
        });

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

    public List<Aluno> All => alunos;

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

Model/ProfessorFakeRepository.cs

```
using System.Collections.Generic;

namespace Model;

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

    public List<Professor> All => profs;

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

Model/AlunoFileRepository.cs

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

namespace Model;
```

```
5
6   // Esse repositório sim, salva os dados em arquivos e não temporáriamente
    na memória
7   public class AlunoFileRepository : IRepository<Aluno>
8   {
9       public List<Aluno> All
10          => DB<Aluno>.App.All;
11
12      public void Add(Aluno obj)
13      {
14          var newAll = All;
15          newAll.Add(obj);
16          DB<Aluno>.App.Save(newAll);
17      }
18  }
```

Model/ProfessorFileRepository.cs

```
1   using DataBase;
2   using System.Collections.Generic;
3
4   namespace Model;
5
6   public class ProfessorFileRepository : IRepository<Professor>
7   {
8       public List<Professor> All
9           => DB<Professor>.App.All;
10
11      public void Add(Professor obj)
12      {
13          var newAll = All;
14          newAll.Add(obj);
15          DB<Professor>.App.Save(newAll);
16      }
17  }
```

Front/Program.cs

```
1   using static System.Console;
2   using Model;
3
4   IRepository<Aluno> alunoRepo = null;
5   IRepository<Professor> profRepo = null;
6
7   // Se você estiver no modo debug (dotnet run) não acessará arquivos,
    apenas um repositório fake que traz dados de mentira
8   // para facilitar testes sem ter que apagar ou reiniciar os dados da
    aplicação. Sem em release (dotnet run -c release)
9   // você está gerando o produto final que acessa o 'banco de dados' nos
    arquivos. Você ainda pode criar um novo tipo de repositório
10  // na Model, conectando com o banco de dados SQL, por exemplo, e só
    alterar aqui sem ter que alterar mais nada da aplicação.
11  #if DEBUG
12
13  alunoRepo = new AlunoFakeRepository();
```

```
14  profRepo = new ProfessorFakeRepository();
15
    #else
16
17  alunoRepo = new AlunoFileRepository();
18  profRepo = new ProfessorFileRepository();
19
    #endif
20
21  while (true)
22  {
23      try
24      {
25          Clear();
26          WriteLine("1 - Cadastrar Professor");
27          WriteLine("2 - Cadastrar Aluno");
28          WriteLine("3 - Ver Professores");
29          WriteLine("4 - Ver Alunos");
30          WriteLine("5 - Sair");
31          int opt = int.Parse(ReadLine());
32
33          switch (opt)
34          {
35              case 1:
36                  break;
37
38              case 2:
39                  Aluno aluno = new Aluno();
40                  aluno.Nome = ReadLine();
41                  aluno.Idade = int.Parse(ReadLine());
42                  alunoRepo.Add(aluno);
43                  break;
44
45              case 3:
46                  var profs = profRepo.All;
47                  for (int i = 0; i < profs.Count; i++)
48                  {
49                      WriteLine(profs[i].Nome);
50                      WriteLine(profs[i].Formacao);
51                      WriteLine();
52                  }
53                  break;
54
55              case 4:
56                  var alunos = alunoRepo.All;
57                  for (int i = 0; i < alunos.Count; i++)
58                  {
59                      WriteLine(alunos[i].Nome);
60                      WriteLine(alunos[i].Idade);
61                      WriteLine();
62                  }
63                  break;
64
65              case 5:
66                  return;
67          }
68      }
69      catch
70      {
71          // Um Catch voltado ao usuário final e não mais a renomear o erro
```

```
72              WriteLine("Erro na aplicação, por favor consulte a TI");
73          }
74
75          WriteLine("Aperte qualquer coisa para continuar...");
76          ReadKey(true);
77      }
```

15.9 Exercícios Propostos

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

16 Aula 13 - Coleções e Introdução a Language Integrated Query (LINQ)

16.1 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:

```
1  public interface IEnumerator<T>
2  {
3      T Current { get; }
4      bool MoveNext();
5      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++)
69                  newVetor[i] = vetor[i];
70              vetor = newVetor;
71          }
72
73          vetor[pos] = value;
74          pos++;
75      }
76
77      // Simplismente retornamos o iterador que fizemos
78      public IEnumerator<T> GetEnumerator()
79      {
80          ListIterator<T> it = new ListIterator<T>();
81          it.List = this;
82          return it;
83      }
84
85      // Mesma condição que na implementação do iterador
86      IEnumerator IEnumerable.GetEnumerator()
87          => GetEnumerator();
88  }
```

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:

```
1   List<int> lista = new List<int>();
2
3   lista.Add(1);
4   lista.Add(2);
5   lista.Add(3);
6
7   foreach(var n in lista)
8   {
9       Console.WriteLine(n);
10  }
11
12  // ...
```

16.2 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:

```
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  foreach(var n in lista)
12      Console.WriteLine(n);
13
14  public class List<T> : IEnumerable<T>
15  {
16      private int pos = 0;
17      private T[] vetor = new T[10];
18      public int Count => pos;
19
20      public T this[int index]
21      {
22          get => this.vetor[index];
23          set => this.vetor[index] = value;
24      }
25
26      public void Add(T value)
27      {
28          int len = vetor.Length;
29          if (pos == len)
30          {
31              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);

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

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

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

IEnumerator<int> get()
{
    Console.WriteLine("Entrei no get");
    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

16.3 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);
11  stack.Push(2);
12  stack.Push(3);
13
14  int[] array = new int[] { 1, 2, 3 };
15
16  Console.WriteLine(Enumerable.Count(list)); // 3
17  Console.WriteLine(Enumerable.Count(stack)); // 3
18  Console.WriteLine(Enumerable.Count(array)); // 3
19
20  public static class Enumerable
21  {
22      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;
    }

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

        var it = coll.GetEnumerator();
        while (it.MoveNext())
```

```
            count++;

        return count;
    }
}
```

16.4 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:

```
using System;

MyExtensionMethods.Print("Oi");

public static class MyExtensionMethods
{
    public static void Print(string text)
    {
        Console.WriteLine(text);
    }
}
```

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:

```
using System;

// Quando isso compila...
"Oi".Print();
// Vira isso...
// MyExtensionMethods.Print("Oi");

// objeto estranho.PrintObj() também funciona
new AppDomainUnloadedException().PrintObj();

public static class MyExtensionMethods
{
    // Basta adicionar 'this' antes do primeiro parâmetro no método estático e adicionamos o método Print dentor da string
    public static void Print(this string text)
    {
        Console.WriteLine(text);
    }

    // Ou de QUALQUER objeto
    public static void PrintObj(this object obj)
    {
        Console.WriteLine(obj);
    }
}
```

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

```cs
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;
    }
}
```

16.5 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:

```cs
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

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;
    }
}
```

16.6 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.

16.7 Exercícios Propostos

Implemente as seguintes funções LINQ:

```
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

```

```
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

17 Aula 14 - Programação Funcional e LINQ

17.1 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)
```

17.2 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)
{
```

```
35      void funcaoProduto(float x)
36      {
37          return f(x) * g(x);
38      }
39      return funcaoProduto;
40  }
41
42  // a * x + b
43  FuncaoMatematica linear(float a, float b)
44  {
45      var ax = produto(constante(a), fx());
46      return soma(ax, constante(b));
47  }
48
49  public delegate float FuncaoMatematica(float x);
```

17.3 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)
```

```
FuncaoMatematica constante(float c)
    => x => c;

FuncaoMatematica fx()
    => x => x;

FuncaoMatematica soma(FuncaoMatematica f, FuncaoMatematica g)
    => x => f(x) + g(x);

FuncaoMatematica produto(FuncaoMatematica f, FuncaoMatematica g)
    => x => f(x) * g(x);

FuncaoMatematica linear(float a, float b)
    => x => a * x + b;

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

17.4 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)).

17.5 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;

```

```
17  if (h(100, 10))
18  {
19      var value = g(5, 2).ToString();
20      f(value);
21  }
```

17.6 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:

```
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 o Primeiro valor par, caso contrário retorna o valor default
    (0)
10  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.

17.7 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.

17.8 Exercícios Propostos

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

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

IEnumerable<int> get()
{
    for (int i = 0; i < 1000; i++)
        yield return i + 1;
}
```

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.

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

int[] arrA = new int[] { 1, 3, 5, 7 };
int[] arrB = new int[] { 2, 3, 5, 9};

foreach (var k in arrA.Zip(arrB, (i, j) => j - i))
    Console.Write(k);
// Result: 1002

public static class Enumerable
{
    public static IEnumerable<T> TakeWhile<T>(IEnumerable<T> coll, Func<T, bool> func)
    {
        throw new NotImplementedException();
    }

    public static IEnumerable<T> SkipWhile<T>(IEnumerable<T> coll, Func<T, bool> func)
    {
        throw new NotImplementedException();
    }

    public static IEnumerable<R> Zip<T, U, R>(IEnumerable<T> coll, IEnumerable<U> second, Func<T, U, R> func)
    {
        throw new NotImplementedException();
    }
```

```
27  }
```

Give feedback

18 Aula 15 - LINQ Avançado + Desafio 7

18.1 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,
22      MesNascimento = 4
23  });
24  list.Add(new Pessoa()
25  {
26      Nome = "Pamella",
27      AnoNascimento = 2000,
28      MesNascimento = 6
29  });
30  list.Add(new Pessoa()
31  {
32      Nome = "Bernardo",
33      AnoNascimento = 2000,
34      MesNascimento = 1
35  });
36
37  // Agrupa por Ano de Nascimento
38  var query = list.GroupBy(p => p.AnoNascimento);
39
40  // Cada vez que o foreach roda ele pega um grupo diferente
41  foreach (var group in query)
42  {
```

```
    // 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)
    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.

18.2 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.

```
using static System.Console;

var obj = new { nome = "Pamella", AnoNascimento = 2000 };

```

```
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);
```

18.3 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 != ' ')
10     .Select(g => new {
11         letra = g.Key.ToString().ToUpper(),
12         quantidade = g.Count()
13     })
14     .OrderBy(x => x.letra) // Ordena pela letra, ou seja, deixa em ordem alfabética
15     .Select(x => $"A letra {x.letra} foi escrita {x.quantidade} vezes");
16
17 foreach (var s in query)
18     Console.WriteLine(s);
```

18.4 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:

```
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 =
8      from c in texto
9      group c by c into g
10     where g.Key != ' '
11     select new {
12         letra = g.Key.ToString().ToUpper(),
13         quantidade = g.Count()
14     } into x
15     orderby x.letra
16     select $"A letra {x.letra} foi escrita {x.quantidade} vezes";
17
18 foreach (var s in query)
19     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:

```
1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4
5  int[] arr = new int[]
6  { 1, 3, 5, 7, 9 };
7
8  var query1a = from n in arr select n * n;
9
10 var query1b = arr.Select(n => n * n);
11
12
13 var query2a =
14     from n in arr
15     where n > 4
16     select n * n into x
17     where x < 50
18     select x * x;
19
20 var query2b = arr
```

```
21          .Where(n => n > 4)
22          .Select(n => n * n)
23          .Where(x => x < 50)
24          .Select(x => x * x);
```

18.5 Desafio 7

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.