**-----ENUM-----**

MANEIRA DE REPRESENTAR UM CONJUNTO DE CONSTANTES/PALAVRAS, SENDO CADA CONSTANTE/PALAVRA UM VALOR NUMÉRICO.

EX: UMA CLASSE PEDIDO, CUJO ATRIBUTO Status É UM ENUM DA SEGUINTE FORMA

class Order

{

public int Id { get; set; }

public DateTime Moment { get; set; }

public OrderStatus Status { get; set; }

}

ATRIBUTO Status É DO TIPO enum OrderStatus, SENDO ESSE enum

enum OrderStatus : int

{

PendingPayment = 0,

Processing = 1,

Shipped = 2,

Delivered = 3

}

ASSIM, OS ESTADOS DO PEDIDO PODEM SER RELACIONADOS COM ESSES VALORES NO MOMENTO DE INSTANCIAR UM OBJETO Order

NO PROGRAMA PRINCIPAL

Order order1 = new Order

{

Id = 01,

Moment = DateTime.Now,

Status = OrderStatus.PendingPayment //A PARTIR DO . O COMPILADOR SUGERE

};

PARA ORGANIZAR ESSA ESTRUTURA NUM PROJETO, O IDEAL É CRIAR UMA PASTA ENTIDADES, ONDE FICARÃO TODAS AS CLASSES QUE REPRESENTAM ENTIDADES DA APLICAÇÃO, E DENTRO DESSA PASTA ENTIDADES, CRIAR UMA SUBPASTA ENUMS, ONDE FICARÃO OS ENUMS RELACIONADOS.

ADICIONAR->CLASSE->NOMEAR->TROCAR class POR enum

LEMBRAR DE: PARA QUE UMA ENUM/CLASSE DE UM NAMESPACE DIFERENTE SEJA VISÍVEL, É NECESSÁRIO IMPORTÁ-LA NO CÓDIGO

using ProjetoOO\_3.Entities;

using ProjetoOO\_3.Entities.Enums;

**---CONVERTENDO ENUM PARA STRING---**

string txt = OrderStatus.PendingPayment.ToString();

**---CONVERTENDO STRING PARA ENUM---**

OrderStatus order\_status = Enum.Parse<OrderStatus>("Delivered");

//CONSIDERANDO QUE NA ESTRUTURA DO ENUM OrderStatus JÁ TENHA O VALOR Delivered

//ASSIM A STRING PASSADA NO PARÂMETRO SE TORNA O VALOR DO OBJETO order\_status

//NÃO FUNCIONARIA SE A STRING NÃO EXISTISSE DENTRO DA ESTRUTURA DO ENUM (Delivere POR EXEMPLO)

**REPRESENTAÇÃO UML ENUM**

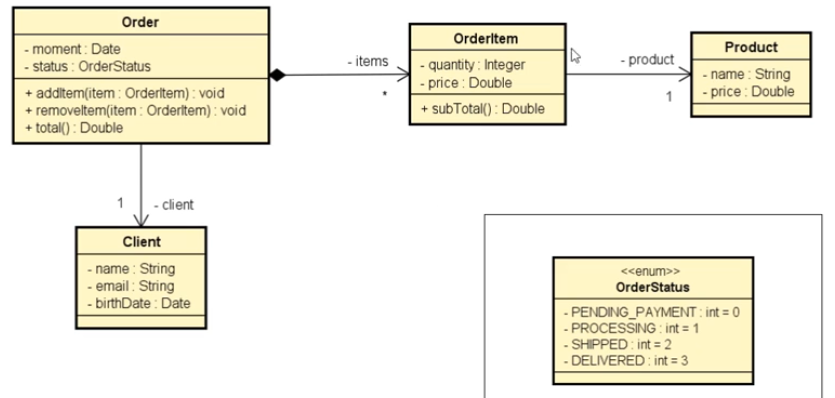


**---COMPOSIÇÃO---**

UM OBJETO CONTÉM OUTRO - TEM UM / TEM VÁRIOS

DIAMANTE PRETO - TODO PARTE

**ENTRE ENTIDADES**

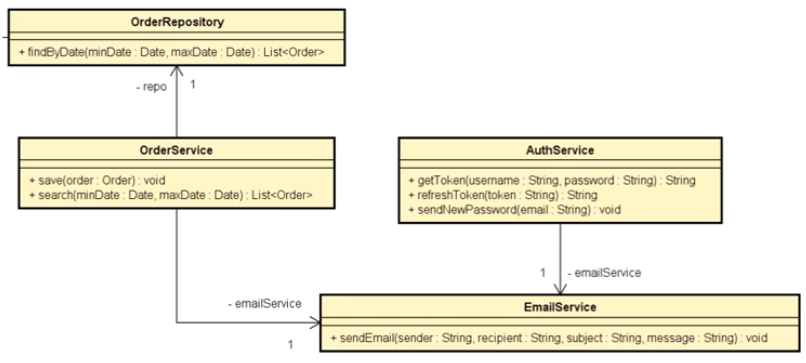


UM PEDIDO CONTÉM VÁRIOS ITENS

UM PEDIDO ESTÁ RELACIONADO A UM CLIENTE (COMPOSIÇÃO TAMBÉM)

UM ITEM TEM 1 PRODUTO

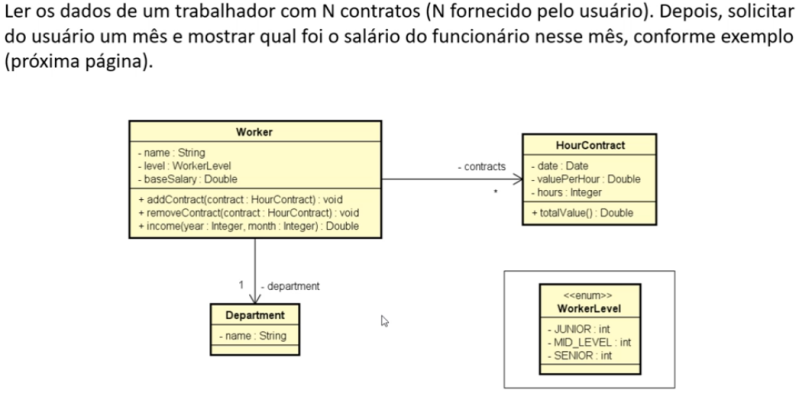
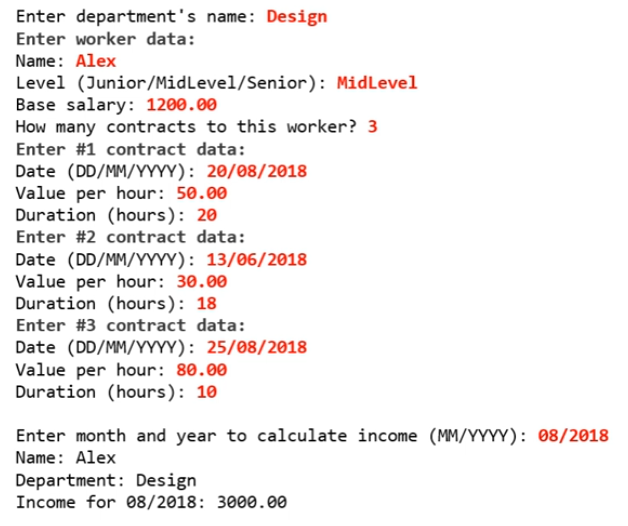
**ENTRE SERVIÇOS**



TANTO UM SERVIÇO DE PEDIDO QUANTO UM SERVIÇO DE AUTENTICAÇÃO ESTÃO RELACIONADOS COM SERVIÇO DE E-MAIL

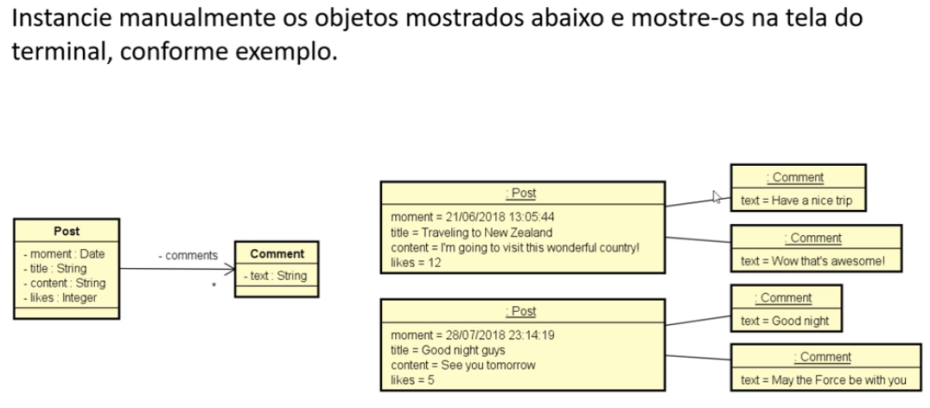
UM SERVIÇO DE PEDIDO ESTÁ RELACIONADO COM UM REPOSITÓRIO DO PEDIDO (QUE ACESSA O BD)

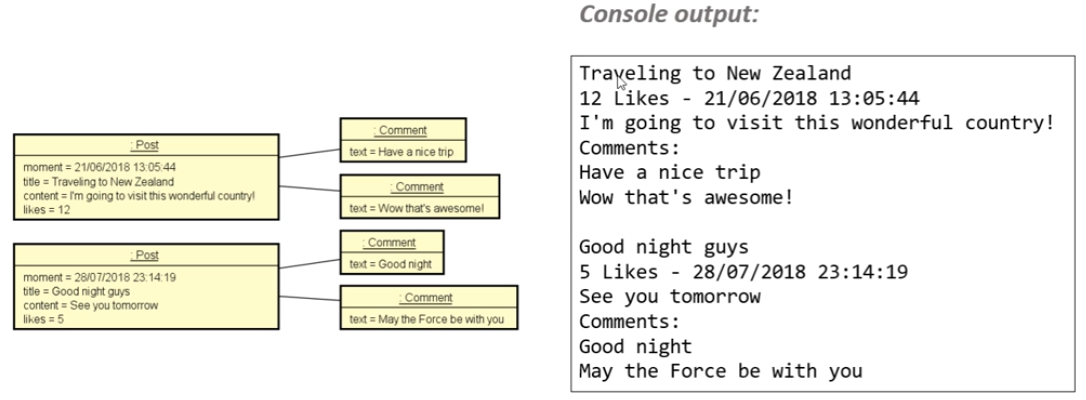
**EXERCÍCIO RESOLVIDO 1**

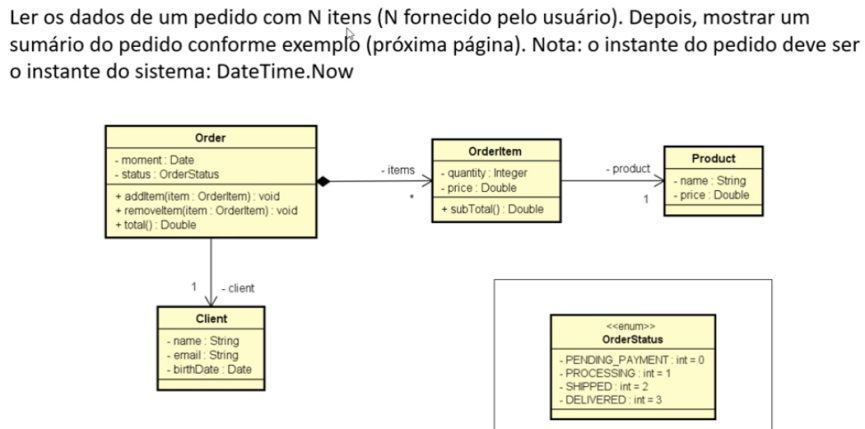
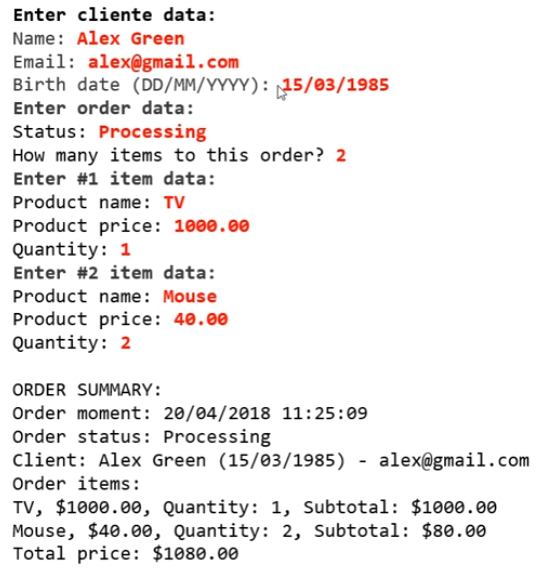
\*NO CONSTRUTOR DE WORKER, NÃO USAR COMO ARGUMENTO A LISTA. SEMPRE NUMA RELAÇÃO DE COMPOSIÇÃO DE UM PARA MUITOS, O CONSTRUTOR DE “UM” NÃO INICIA JÁ COM A LISTA DE “MUITOS”, O MUITOS É ADICIONADO AOS POUCOS.

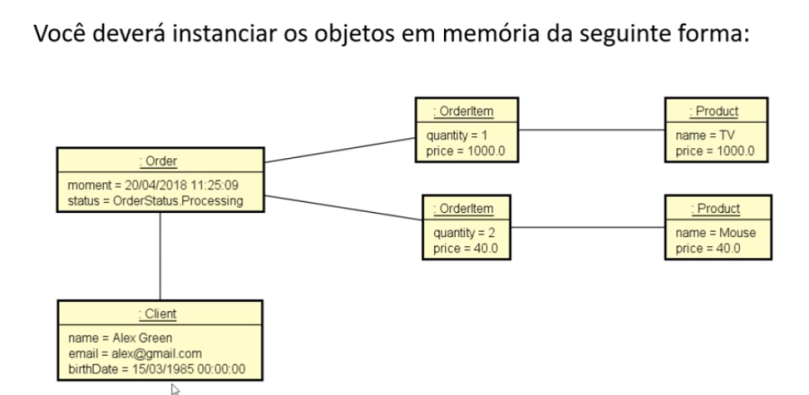
**EXERCÍCIO RESOLVIDO 2**





**EXERCÍCIO RESOLVIDO 3**



**StringBuilder**

UMA CLASSE QUE PODE SER UTILIZADA PARA OTIMIZAR O MÉTODO ToString DE UMA CLASSE. QUANDO O MÉTODO ToString TEM UM CONTEÚDO MUITO GRANDE, É RECOMENDÁVEL UTILIZAR O StringBuilder PARA CRIAR A SAÍDA.

public override string ToString()

{

StringBuilder sb = new StringBuilder();//MONTAR STRING COM MAIS EFICIÊNCIA

sb.AppendLine(Title);

sb.AppendLine($"{Likes} Likes - {Moment.ToString("dd/MM/yyyy HH:mm:ss")}");

sb.AppendLine(Content);

sb.AppendLine("Comments");

foreach(Comment c in Comments)

{

sb.AppendLine(c.Text);

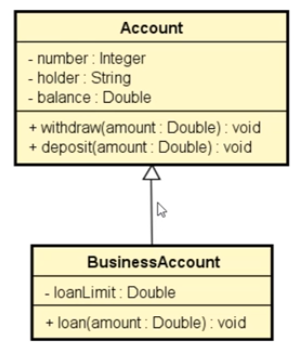
}

return sb.ToString();

}

**HERANÇA**

ASSOCIAÇÃO ENTRE CLASSES QUE PERMITE REUTILIZAR ATRIBUTOS E MÉTODOS DA CLASSE BASE PARA A CLASSE DERIVADA, ADICIONANDO SOMENTE O QUE FOR NECESSÁRIO, AO INVÉS DE REPETIR TUDO.



PARA GARANTIR QUE UM ATRIBUTO/MÉTODO SÓ É ACESSÍVEL NA CLASSE OU EM UMA DE SUAS SUBCLASSES, USAR protected

**---USANDO CONSTRUTOR DA BASE NA DERIVADA---**

CONSTRUTOR DA BASE

public Account(int number, string holder, double balance)

{

Number = number;

Holder = holder;

Balance = balance;

}

CONSTRUTOR DA DERIVADA - USA AS INFORMAÇÕES E ADICIONA UM PARÂMETRO

public BusinnesAccount(int number, string holder, double balance, double loanLimit)

: base(number, holder, balance)

{

LoanLimit = loanLimit;

}

**---UPCASTING E DOWNCASTING---**

SÃO AÇÕES DE:

UPCASTING - INSTANCIAR UM OBJETO DA CLASSE BASE RECEBENDO UM CONSTRUTOR DA CLASSE DERIVADA

DOWNCASTING-INSTANCIAR UM OBJETO DA CLASSE DERIVADA RECEBENDO UM CONSTRUTOR DA CLASSE BASE.

AMBOS SÃO ACEITOS

CONSIDERANDO AS CLASSES Account(BASE) , BusinessAccount(DERIVADA) E SavingAccount(DERIVADA)

Account a = new Account(1, "Luciano", 10500);

BusinnesAccount b = new BusinnesAccount(2, "Comercial Luciano", 35000, 12000);

Account c = new SavingsAccount(3, "Lara", 12000, 0.05);//UPCASTING

//CASTING USANDO as

//Account d = (BusinnesAccount)b;

//Account d = b as BusinnesAccount;

SavingsAccount e = (SavingsAccount)c; //DOWN CASTING

//c É SavingAccount - COMPATÍVEL COM e

//BusinnesAccount e = (BusinnesAccount)a;//DOWNCASTING DEFEITUOSO - NÃO COMPATÍVEL

//BusinnesAccount f = (BusinnesAccount)c;//DOWNCASTING DEFEITUOSO - NÃO COMPATÍVEL

//PARA TESTAR ANTES DE FAZER O CASTING

if (c is BusinnesAccount)

{

BusinnesAccount f = (BusinnesAccount)c;//DOWNCASTING DEFEITUOSO - NÃO COMPATÍVEL

}

else

{

Console.WriteLine("NÃO COMPATÍVEL");

}

DOWNCASTING - OPERAÇÃO INSERGURA - DEVE SER USADA SOMENTE QUANDO NECESSÁRIO

**SOBREPOSIÇÃO COM VIRTUAL E OVERRIDE**

SOBREPOR UM MÉTODO SIGNIFICA QUE, NA CLASSE DERIVADA, O MÉTODO HERDADO DA CLASSE BASE SERÁ REESCRITO, MANTENDO SEU NOME MAS MUDANDO SUA LÓGICA. PRA QUE ISSO SEJA POSSÍVEL, NA CLASSE BASE ELE PRECISA USAR A PALAVRA virtual E NA CLASSE DERIVADA override.

virtual - INDICA QUE O MÉTODO PODE SER SOBREPOSTO NAS SUAS SUBCLASSES

override - INDICA QUE O MÉTODO É UMA SOBREPOSIÇÃO DO MESMO MÉTODO NA SUA CLASSE BASE

NA CLASSE BASE

//SAQUE - CADA SAQUE TEM TAXA DE 5$

public virtual void WithDraw(double ammount)

{

Balance -= ammount+5;

}

NA CLASSE DERIVADA

//MÉTODO SOBRESCRITO - CONTA POUPANÇA NÃO TEM DESCONTO PARA SAQUE

public override void WithDraw(double ammount)

{

Balance -= ammount;

}

CASO SEJA NECESSÁRIO, POR ALGUM MOTIVO, USAR NA CLASSE DERIVADA ALGUM MÉTODO DA CLASSE BASE, SÓ É PRECISO USAR A PALAVRA base

NA CLASSE DERIVADA

base.WithDraw(ammount);

**---CONTROLADOR sealed---**

NAS CLASSES

USADO NA DEFINIÇÃO DE UMA CLASSE PARA IMPEDIR QUE ELA SEJA HERDADA

sealed class SavingsAccount

NOS MÉTODOS

USADO EM MÉTODOS QUE JÁ FORAM SOBREPOSTOS PARA EVITAR QUE SEJAM SOBREPOSTOS NOVAMENTE EM SUAS SUBCLASSES

public sealed override void WithDraw(double ammount)

\*BOA PRÁTICA: SEMPRE SELAR MÉTODOS SOBREPOSTOS - MAIS SEGURO E MAIS EFICIENTE

**POLIMORFISMO**

RECURSO PRA POO QUE PERMITE QUE VARIÁVEIS DE UM MESMO TIPO GENÉRICO POSSAM APONTAR PARA OBJETOS DE TIPOS ESPECÍFICOS DIFERENTES (EM TEMPO DE EXECUÇÃO).

TIPO GENÉRICO Account APONTANDO PARA DOIS TIPOS ESPECÍFICOS (BusinessAccount E SavingsAccount)

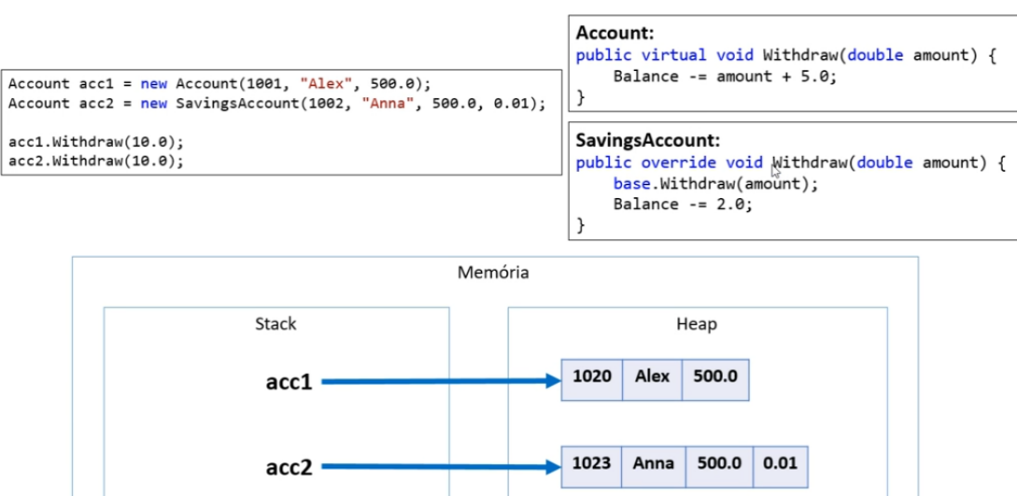
Account a = new BusinnesAccount(1, "Luciano", 400,1200);

Account b = new SavingsAccount(2, "Ana", 400, 0.05);

a.WithDraw(10);

b.WithDraw(10);

Console.WriteLine($"{a.Balance}\n{b.Balance}");



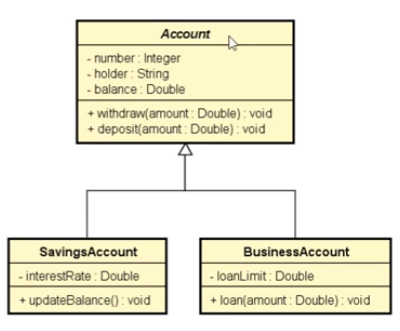
**CLASSES ABSTRATAS**

NÃO PODEM SER INSTANCIADAS, APENAS SUAS SUBCLASSES

NOTAÇÃO UML: ITÁLICO

MESMO SENDO ABASTRATA, AINDA É POSSÍVEL DECLARAR UMA LISTA DO TIPO DESSA CLASSE

abstract class Account



\*MÉTODOS ABSTRATOS SÃO AQUELES QUE NÃO POSSUEM IMPLEMENTAÇÃO, CONSIDERANDO QUE SÃO TÃO GENÉRICOS, QUE SUA LÓGICA SÓ PODE SER IMPLEMENTADA NA SUBCLASSE ESPECÍFICA.

SE UMA CLASSE POSSUI AO MENOS UM MÉTODO ABSTRATO, A CLASSE TAMBÉM SE TORNA ABSTRATA.

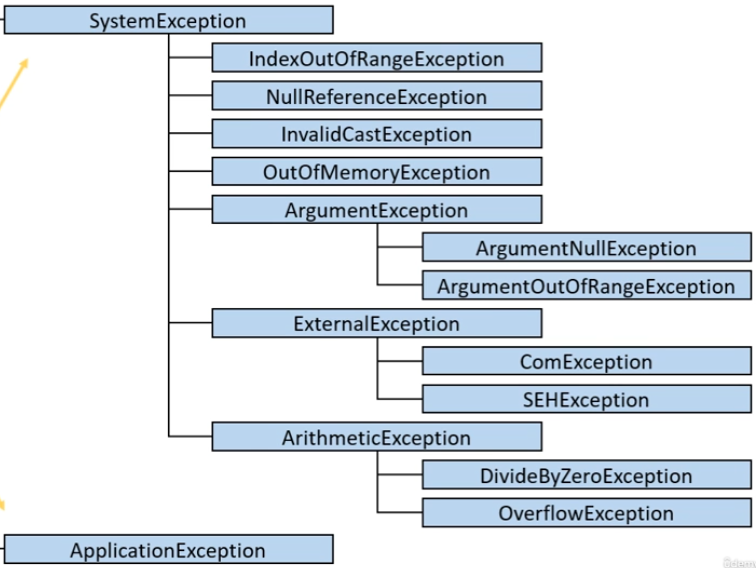
public abstract double Area();

//SEM IMPLEMENTAÇÃO - SEM CHAVES

**---EXCEÇÕES---**

SystemException - Exceções do próprio sistema.

ApplicationException - Para criação de exceções pelo programador na aplicação.



**--TRY-CATCH--**

try

{

CÓDIGO COM POSSIBILIDADE DE ERRO

}

catch (TIPO EXCEÇÃO x)

{

Console.WriteLine($"Erro! {x.Message}");

}

**--FINALLY--**

Finally é um bloco opcional que poder ser incluído ao try-catch. Ele será executado independente de um catch ter sido ativado ou não. Muito usado para fechar conexão com arquivo de texto e banco de dados

try

{

int a = int.Parse(Console.ReadLine());

int b = int.Parse(Console.ReadLine());

int c = a / b;

Console.WriteLine(c);

}

catch (FormatException e)//VARIÁVEL DO TIPO ERRADO

{

Console.WriteLine($"Erro! {e.Message}");

}

catch (DivideByZeroException e1)//DIVISÃO POR ZERO

{

Console.WriteLine($"Erro! {e1.Message}");

}

catch(Exception e)//GENERIC ERROR

{

Console.WriteLine($"Erro! {e.Message}");

}

finally

{

Console.WriteLine("Aparece de qualquer forma");

}

**---EXCEÇÕES PERSONALIZADAS---**

PARA TRATAR EXCEÇÕES QUE SÃO REGRAS DE NEGÓCIOS, USA-SE CONDICIONAIS ALIADAS COM COMANDO THROW E UMA ESTRURA DE UMA CLASSE DomainException NA PASTA Execptions QUE HERDA DE ApplicationException

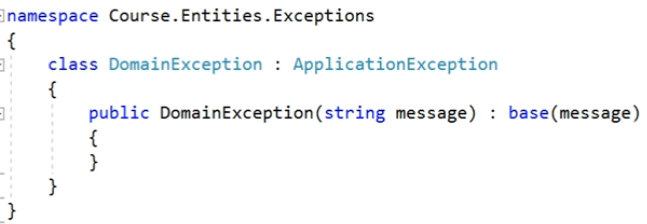
DomainException - NA ANÁLISE DE SISTEMA, Domain É O ESCOPO DO NEGÓCIO

Herdar de ApplicationException - SIGNIFICA QUE OS ERROS SÃO QUEBRAS DAS REGRAS DE NEGÓCIO

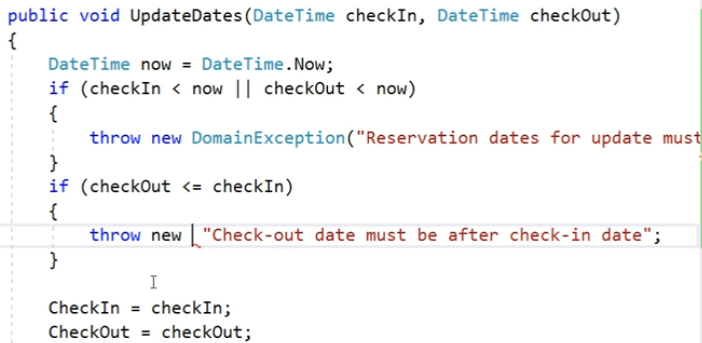
CONSIDERANDO O EXEMPLO DE UM SISTEMA DE RESERVERSAS ONDE

- A DATA DE RESERVA NÃO PODE SER NO PASSADO

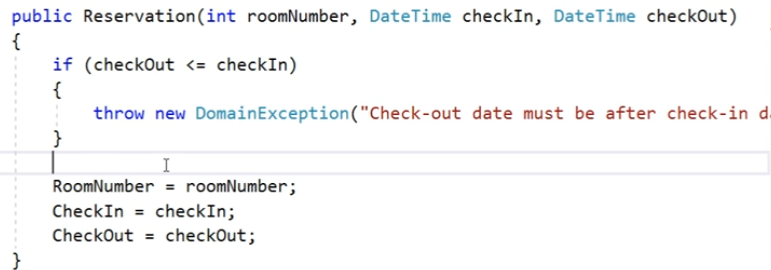
-A DATA DE CHECKIN NÃO PODE SER DEPOIS DE CHECKOUT



CLASSE DomainException com um construtor que usa o construtor da classe base (ApplicationException)

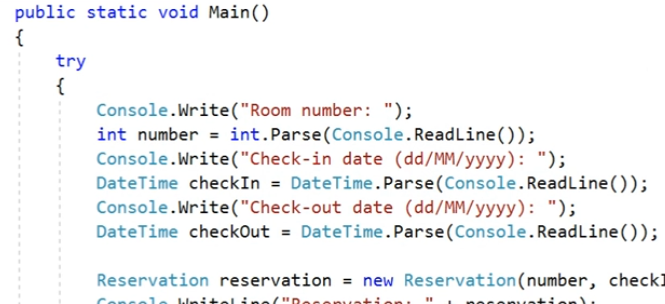


NA CLASSE RESERVA, NA FUNÇÃO DE ATUALIZAR A RESERVA, QUANDO FEITA A COMPARAÇÃO DE DATAS, SE QUEBRAR A REGRA DE NEGÓCIO, É LANÇADA A EXCEÇÃO, QUE É UMA INSTÂNCIA DE DomainException RECEBENDO COMO ARGUMENTO A MENSAGEM DE ERRO.



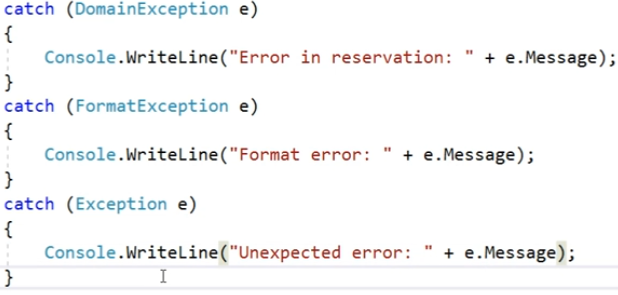
TAMBÉM É FEITA A VERIFICAÇÃO DE ERRO NO CONSTRUTOR DA CLASSE. CASO AS DATAS QUEBREM AS REGRAS DE NEGÓCIO, É LANÇADA A EXCEÇÃO.

E PARA QUE ESSA EXECÇÕES SEJAM LANÇADAS, NO PROGRAMA PRINCIPAL É NECESSÁRIO COLOCAR TODAS AS LINHAS QUE POSSAM GERAR ESSES ERROS DENTRO DE UM BLOCO TRY, COM O CATCH ABAIXO COM TIPO DE ERRO DomainException



.

.



**-----OPERAÇÕES COM ARQUIVOS-----**

System.IO.File

System.IO.FileInfo

File - ESTÁTICO - NÃO PRECISA DE OBJETO INSTANCIADO - MAIS LENTO

FileInfo - PRECISA DE OBJETO INSTANCIADO - MAIS RÁPIDO



//CAMINHO ARQUIVO ORIGEM

string sourcePath = @"C:\Users\Luciano\Desktop\arquivos\file1.txt";

//CAMINHO ARQUIVO ALVO

string targetPath = @"C:\Users\Luciano\Desktop\arquivos\file2.txt";

try

{

FileInfo fileInfo = new FileInfo(sourcePath);

fileInfo.CopyTo(targetPath);//GERA UM ARQUIVO NOVO A PARTIR DO ARQUIVO ORIGEM

//LÊ CADA LINHA E SALVA NUMA POSIÇÃO DO VETOR

string[] lines = File.ReadAllLines(sourcePath);

foreach(string line in lines)

{

Console.WriteLine(line);

}

}

catch (IOException e)

{

Console.WriteLine($"Erro. {e.Message}");

**---FileStream E StreamReader---**

OPERAÇÕES COM ARQUIVOS PARA LEITURA

FileStream - MODO DE ABERTURA DO ARQUIVO

StreamReader - OPERAÇÕES DE LEITURA NO ARQUIVO

StremReader -> FileStream -> ARQUIVO

//CAMINHO ARQUIVO ORIGEM

string path = @"C:\Users\Luciano\Desktop\arquivos\file1.txt";

FileStream fs = null;

StreamReader sr = null;

try

{

//ABRIR ARQUIVO JÁ EXISTENTE

fs = new FileStream(path, FileMode.Open);

sr = new StreamReader(fs);//ASSOCIAÇÃO StreamReader com FileStream

string line = sr.ReadLine();//LER UMA LINHA DO ARQUIVO

Console.WriteLine(line);

}

catch (IOException e)

{

Console.WriteLine($"Erro. {e.Message}");

}

finally

{

//FECHAR MANUALMENTE CONEXÃO COM ARQUIVO

if (sr != null) sr.Close();

if (fs != null) fs.Close();

}

ABORDAGEM ALTERNATIVA - AO INVÉS DE TER QUE INSTANCIAR UM OBJETO FILESTREAM E TAMBÉM UM STREAMREADER, FAZER SOMENTE UM STREAMREADER QUE USA A CLASSE FILE. SENDO ASSIM, INTERNAMENTE ELE CRIA UM FILESTREAM, MAS SEM PRECISAR CITÁ-LO EXPLICITAMENTE NO CÓDIGO

//CAMINHO ARQUIVO ORIGEM

string path = @"C:\Users\Luciano\Desktop\arquivos\file1.txt";

StreamReader sr = null;

try

{

//ABRIR ARQUIVO JÁ EXISTENTE

sr = File.OpenText(path);//StreamReader USANDO File

string line = sr.ReadLine();//LER UMA LINHA DO ARQUIVO

Console.WriteLine(line);

}

catch (IOException e)

{

Console.WriteLine($"Erro. {e.Message}");

}

finally

{

//FECHAR MANUALMENTE CONEXÃO COM ARQUIVO

if (sr != null) sr.Close();

}

LER TODAS AS LINHAS DO ARQUIVO

sr = File.OpenText(path);//StreamReader USANDO File

while (!sr.EndOfStream)

{

string line = sr.ReadLine();//LER UMA LINHA DO ARQUIVO E PULAR PARA PRÓXIMA

Console.WriteLine(line);

}

FAZER ABERTURA DO ARQUIVO COM FECHAMENTO AUTOMÁTICO

UTILIZAR PALAVRA using

//CAMINHO ARQUIVO ORIGEM

string path = @"C:\Users\Luciano\Desktop\arquivos\file1.txt";

try

{

using (FileStream fs = new FileStream(path, FileMode.Open))

{

using (StreamReader sr = new StreamReader(fs))

{

while (!sr.EndOfStream)

{

string line = sr.ReadLine();

Console.WriteLine(line);

}

}

}

}

catch(IOException e)

{

Console.WriteLine($"Erro. {e.Message}");

}

USANDO APENAS STREAM READER E FILE

try

{

using (StreamReader sr = File.OpenText(path))

{

**---FileStream e StreamWriter---**

ESCRITA NO ARQUIVO

//CAMINHO ARQUIVO ORIGEM

string sourcePath = @"C:\Users\Luciano\Desktop\arquivos\file1.txt";

string targetPath = @"C:\Users\Luciano\Desktop\arquivos\file2.txt";

try

{

//VETOR EM QUE CADA ITEM É UMA LINHA DO ARQUIVO ORIGINAL

string[] lines = File.ReadAllLines(sourcePath);

//MODO DE ESCRITA - NO FINAL DO ARQUIVO

using(StreamWriter sw = File.AppendText(targetPath))

{

//CADA ITEM DO VETOR É ESCRITO NO FINAL DO ARQUIVO - EM MAIÚSCULO

//ESCREVE UMA LINHA - ENTER

foreach(string line in lines)

{

sw.WriteLine(line.ToUpper());

}

}

}

**-----Directory E DirectoryInfo-----**

OPERAÇÕES COM PASTAS - MESMA LÓGICA DE File E FileInfo

LISTAR TODAS SUBPASTAS A PARTIR DE UMA PASTA INFORMADA

try

{

IEnumerable<string> folders = Directory.EnumerateDirectories(path, "\*.\*", SearchOption.AllDirectories);

//LISTA IEnumerable É UMA LISTA GENÉRICA

//\*.\* É UMA MÁSCARA DE PESQUISA - SIGNIFICA QUALQUER NOME DE QUALQUER TIPO DE ARQUIVO

Console.WriteLine("Folders");

foreach(string folder in folders)

{

Console.WriteLine(folder);

}

}

LISTAR TODOS ARQUIVOS DA PASTA E SUAS SUBPASTAS

IEnumerable<string> files= Directory.EnumerateFiles(path, "\*.\*", SearchOption.AllDirectories);

CRIAR UMA NOVA PASTA

Directory.CreateDirectory(@"C:\Users\Luciano\Desktop\myfolder\newfolder");

**--PATH--**

OPERAÇÕES COM STRINGS QUE REPRESENTAM CAMINHOS DE ARQUIVOS OU PASTAS

//CAMINHO DA PASTA

string path = @"C:\Users\CREAS\Desktop\myfolder\file1.txt";

try

{

Console.WriteLine($"Nome do arquivo (com extensão): {Path.GetFileName(path)}");

Console.WriteLine($"Nome do arquivo (sem extensão):{Path.GetFileNameWithoutExtension(path)}");

Console.WriteLine($"Extensão do arquivo: {Path.GetExtension(path)}");

Console.WriteLine($"Nome da pasta do arquivo: {Path.GetDirectoryName(path)}");

Console.WriteLine($"Caminho absoluto do arquivo: {Path.GetFullPath(path)}");

Console.WriteLine($"Caractere de separação: {Path.DirectorySeparatorChar}");

Console.WriteLine($"Pasta temporária da máquina: {Path.GetTempPath()}");

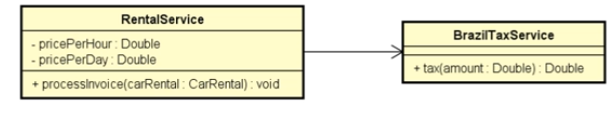
}

**-----INTERFACE-----**

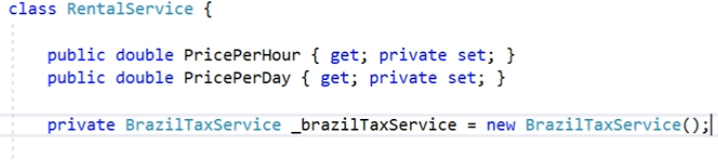
TIPO QUE DEFINE UM CONJUNTO DE OPERAÇÕES QUE UMA CLASSE OU STRUCT DEVE IMPLEMENTAR

SEMPRE COM I NA FRENTE DO NOME

SERVIÇO DE ALUGUEL QUE USA O SERVIÇO DE TAXAÇÃO BRASILEIRA (SEM INTERFACE)

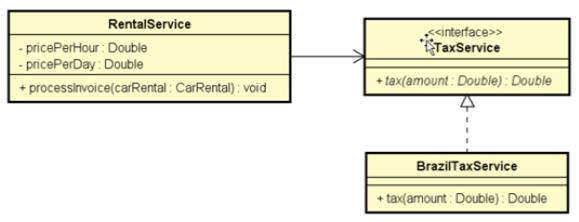


DEPENDÊNCIA MUITO FORTE ENTRE AS DUAS CLASSES DE SERVIÇO. SE FOR NECESSÁRIO MUDAR O SERVIÇO DE TAXAÇÃO, TAMBÉM SERÁ PRECISO MUDAR O CÓDIGO EM RentalService

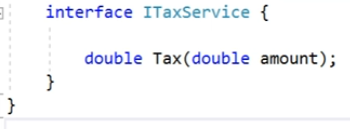


PARA QUE ISSO SEJA REMEDIADO, USA-SE INTERFACES.

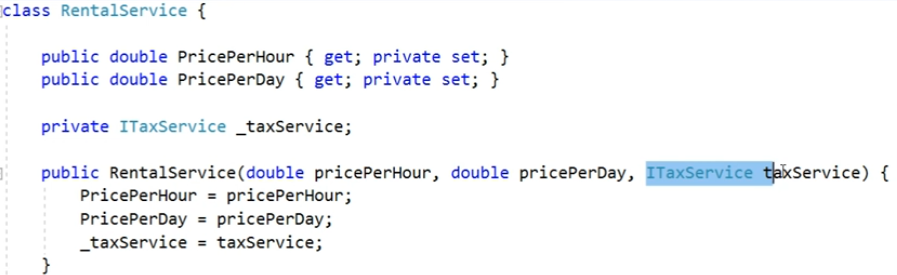
CRIA-SE UMA INTERFACE ITaxService QUE PODE SER IMPLEMENTADA POR QUALQUER SERVIÇO DE TAXAÇÃO, E ESSA INTERFACE É LIGADA AO SERVIÇO DE RentalService



INTERFACE ITaxService

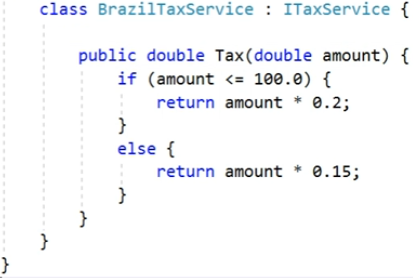


IMPLEMENTAÇÃO EM RentalService



ATRIBUTO PRIVADO \_taxService , DO TIPO ITaxService É USADO NO CONSTRUTOR DA CLASSE

IMPLEMENTAÇÃO DO CLASSE DE SERVIÇO BrasilTaxService, QUE USA A INTERFACE ITaxService



\*BrasilTaxService : ITaxService - NÃO É HERANÇA, É IMPLEMENTAÇÃO DE INTERFACE.

E NO PROGRAMA PRINCIPAL, AO INSTANCIAR UM OBJETO RentalService, USAR NO CONSTRUTOR UM OBJETO BrasilTaxService, QUE VAI SER COMPATÍVEL COM O OBJETO ITaxService DO CONSTRUTOR

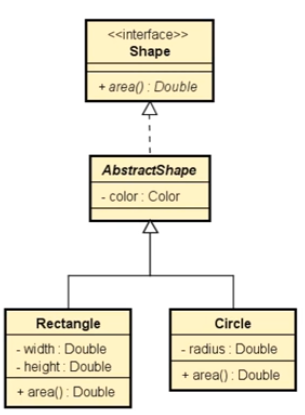
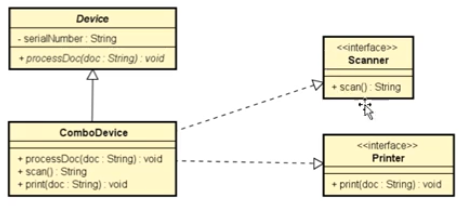


**---CLASSES ABSTRATAS + INTERFACES---**

INTERFACES - CONTRATO A SER CUMPRIDO- ESTRUTURAR MÉTODOS

CLASSES ABSTRATAS - HERANÇA - ESTRUTURAR MÉTODOS E ATRIBUTOS.

ESSES CONCEITOS PODEM SER APLICADOS JUNTOS

Classe ComboDevice herda da classe abstrata Device e implementa as duas interfaces Scanner e Printer

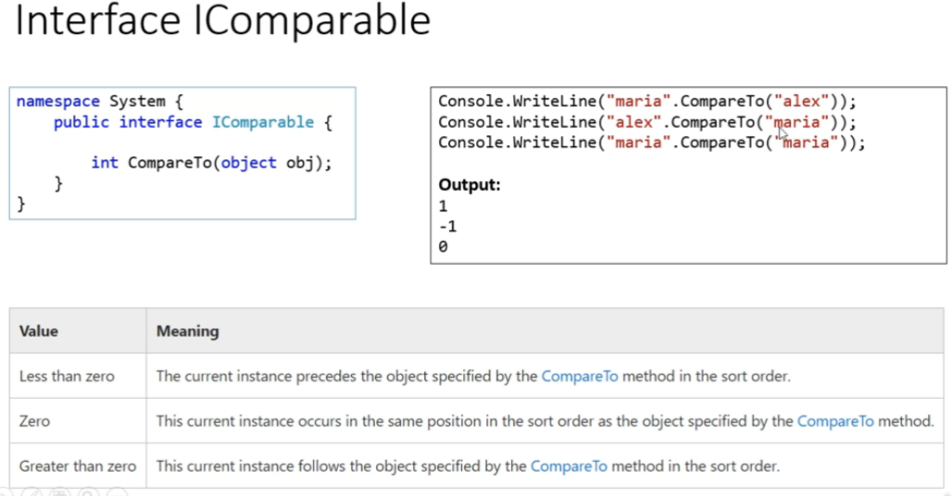
**-----IComparable-----**

É UMA INTERFACE QUE PERMITE QUE UMA CLASSE SEJA CAPAZ DE TER SEUS OBJETOS COMPARADOS COM OUTROS.

GERALMENTE, OBJETOS QUE NÃO HERDAM DE IComparable NÃO PODEM SER COMPARADOS, SENDO ASSIM, NA CLASSE CUJOS OBJETOS PRECISEM SER COMPARADOS (POR EXEMPLO, SEREM ORDENADOS), É PRECISO IMPLEMENTAR A INTERFACE IComparable (MÉTODO CompareTo) NESSA CLASSE. SUPONDO UMA CLASSE Employee COM ATRIBUTOS Name E Salary, TENDO O NOME COMO PRIORIDADE NA COMPARAÇÃO



FUNCIONAMENTO DO MÉTODO CompareTo



CompareTo

ENTRE STRINGS - ORDENA ALFABETICAMENTE

ENTRE NUMÉRICOS - MAIOR VALOR

**-----COMPARISON-----**

COMPRARA OBJETOS SEM PRECISAR IMPLEMENTAR A INTERFACE IComparable (E O MÉTODO CompareTo) DENTRO DA CLASSE DO OBJETO.

A CLASSE É CONSTRUÍDA NORMALMENTE, E NO PROGRAMA PRINCIPAL, USA-SE O MÉTODO Sort APLICADA À LISTA/COLEÇÃO COM OS OBJETOS, USANDO COMO PARÂMETRO DO SORT UM MÉTODO ESTÁTICO QUE USA O CompareTo. Exemplo, SUPONDO UMA CLASSE Product COM NOME E PREÇO.

NO PROGRAMA PRINCIPAL

class Product

{

public string Name { get; set; }

public double Price { get; set; }

//CONSTRUTORES

public Product()

{

}

public Product(string name, double price)

{

Name = name;

Price = price;

}

public override string ToString()

{

return $"{Name} - ${Price}";

}

}

NO PROGRAMA PRINCIPAL

static void Main(string[] args)

{

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

lista.Add(new Product("TV", 900));

lista.Add(new Product("Notebook", 1200));

lista.Add(new Product("Tablet", 450));

//lista.Sort(CompareProducts);

/\*

Sort NÃO FUNCIONARIA SEM A PASSAGEM DO MÉTODO COMO PARÂMETRO

PQ OS OBJETOS Product NÃO SABEM COMO PODEM SE COMPARAR, JÁ QUE

A CLASSE Product NÃO IMPLEMENTA A INTERFACE IComparable

A UTILIZAÇÃO DO MÉTODO COMO PARÂMETRO É UMA REFERÊNCIA PARA A FUNÇÃO

COM TYPESAFETY - A FUNÇÃO É DO TIPO DELEGATE

O MÉTODO ESTÁTICO É USADO COMO PARÂMETRO, ASSIM O SORT USA A LÓGICA

DELE PARA ORDERNAR

CompareProducts FICA NO MESMO ARQUIVO, FORA DA MAIN

\*/

//ASSIM TAMBÉM FUNCIONARIA

//Comparison<Product> comp = CompareProducts;

//lista.Sort(comp);

//ASSIM TAMBÉM FUNCIONARIA - FUNÇÃO LAMBDA - NÃO PRECISA CONSTRUIR A FUNÇÃO ESTÁTICA FORA DO MAIN

//Comparison<Product> comp = (p1, p2) => p1.Name.ToUpper().CompareTo(p2.Name.ToUpper());

//lista.Sort(comp);

//ASSIM TAMBÉM FUNCIONARIA - PASSANDO A FUNÇÃO LAMBDA DIRETO NO SORT

//lista.Sort((p1, p2) => p1.Name.ToUpper().CompareTo(p2.Name.ToUpper()));

//CADA VEZ MENOS LINHAS E MENOS COMPREENSÍVEL

foreach (Product produto in lista)

{

Console.WriteLine(produto);

}

}

//MÉTODO ESTÁTICO QUE É USADO NO DELEGATE PARA A FUNÇÃO Sort

static int CompareProducts(Product p1, Product p2)

{

return p1.Name.ToUpper().CompareTo(p2.Name.ToUpper());

}

**-----GENERICS-----**

RECURSO QUE PERMITE QUE CLASSES, INTERFACES E MÉTODOS SEJAM PARAMETRIZADOS POR TIPO.

NA CONSTRUÇÃO DA CLASSE, USAR UM TIPO GENÉRICO DE NOME QUALQUER TODA VEZ QUE FIZER REFERÊNCIA AO TIPO DA VARIÁVEL

class PrintService<T>

{

private T[] \_values = new T[10];

.

.

public void AddValue(T value)

{

.

.

public T First()

{

E NO PROGRAMA PRINCIPAL,USAR A CLASSE DEFININDO QUAL O TIPO QUE SUBSTITUIRÁ O GENÉRICO

PrintService<string> printService = new PrintService<string>();

LEMBRANDO QUE, PARA USAR GENERIC, O TIPO QUE IRÁ SUBSTITUÍ-LO PRECISA SER SUBTIPO DE IComparable.TODOS TIPOS NATIVOS SÃO(STRING, INT, DOUBLE, FLOAT,....). APENAS OBJETOS DE CLASSES CRIADAS QUE PRECISAM TER NA SUA CLASSE O OVERRIDE DE CompareTo PARA FUNCIONAR

PORQUÊ NÃO USAR OBJECT? PORQUE NÃO TEM TYPESAFETY - ACEITA QUALQUER VALOR E PODE INFLUENCIAR NA LÓGICA DO PROGRAMA (AO INVÉS DE SOMAR, CONCATENA,.....) - ISSO O COMPILADOR NÃO PREVÊ, E O PROBLEMA SÓ ACONTECE EM TEMPO DE EXECUÇÃO.

**--MÉTODO GENÉRICO NUMA CLASSE COMUM USANDO TIPOS NATIVOS--**

SUPONDO UMA CLASSE CalculationService QUE COMPARA OS ITENS DE UMA LISTA E RETORNA O MAIOR

//T Max<T> - MÉTODO Max COM RETORNO TIPO T

//<T> SIGNIFICA QUE O MÉTODO É GENÉRICO - DO TIPO T

//where T: IComparable - SIGNIFICA "TENDO ELEMENTOS COMPARÁVEIS PELO IComparable

public T Max<T>(List<T> list) where T: IComparable

{

//VERIFICAÇÃO SE A LISTA NÃO ESTÁ VAZIA

if (list.Count == 0)

{

throw new ArgumentException("List empty");

}

//PRIMEIRO ELEMENTO É DEFINIDO COMO MAIOR

T max = list[0];

for(int i = 1; i < list.Count; i++)

{

if (list[i].CompareTo(max) > 0)//SE O ELEMENTO ATUAL FOR MAIOR QUE max

{

max = list[i];//ELEMENTO ATUAL SE TORNA O MAIOR

}

}

return max;

}

**--MÉTODO GENÉRICO-CLASSE COMUM USANDO SEUS OBJETOS PARA COMPARAÇÃO--**

NESSE CASO, AO INVÉS DE SÓ USAR O MÉTODO CompareTo NA LÓGICA, É NECESSÁRIO SOBRESCREVER O CompareTo NA CLASSE DO PRODUTO QUE VAI SER COMPARADO

SUPONDO UMA CLASSE DE PRODUTOS

class Product : IComparable

{

.

.

public int CompareTo(object? obj)

{

//VERIFICAÇÃO SE O QUE ESTÁ SENDO COMPARADO É PRODUTO

if (!(obj is Product))//SE NÃO FOR UM PRODUTO

{

throw new ArgumentException("Can't compare. Object is not a Product!");

}

//COMPARAÇÃO ENTRE PRODUTOS BASEADA NO PREÇO

//JÁ USANDO O MÉTODO CompareTo DO PRÓPRIO PREÇO - QUE É DOUBLE

Product other = obj as Product;

return Price.CompareTo(other.Price);

}

**---GetHashCode E Equals---**

SÃO MÉTODOS PARA COMPARAR OBJETOS. PARA TIPOS NATIVOS FUNCIONAM NORMALMENTE. PARA OBJETOS DE CLASSE É NECSSÁRIO SOBRESCREVER.

GetHashCode - É MAIS RÁPIDO MAS HÁ UMA PEQUENA CHANCE DO RESULTADO POSITIVO NÃO SER CORRETO

Equals - É MAIS LENTO MAS SEU RESULTADO É 100% CORRETO

USAR GetHashCode PARA VARRER GRANDES QUANTIDADE DE DADO E O Equals PARA CONFIRMAR O RESULTADO.

string a = "Luciano";

string b = "Luciano";

string c = "Dudu";

Console.WriteLine(a.Equals(b)); //true

Console.WriteLine(a.Equals(c)); //false

Console.WriteLine(a.GetHashCode());//NÚMERO HASH

Console.WriteLine(b.GetHashCode());//MESMO NÚMERO HASH

Console.WriteLine(c.GetHashCode());//NÚMERO HASH DIFERENTE

**--GetHashCode E Equals COM OBJETOS---**

É NECESSÁRIO IMPLEMENTAR A LÓGICA NA CLASSE

SUPONDO UMA CLASSE Client COM ATRIBUTOS NOME E EMAIL

public override bool Equals(object? obj)

{

//VERIFICAÇÃO SE O ARGUMENTO É UM CLIENTE

if(!(obj is Client))

{

return false;//SE NÃO FOR, JÁ É FALSE

}

Client other = obj as Client;//obj PASSADO PARA other - COMO CLIENT(DOWNCASTING)

//NESSA REGRA DE NEGÓCIO, OS CLIENTES SÃO IGUAIS SE TIVEREM O E-MAIL IGUAL

return Email.Equals(other.Email);

}

//LEMBRANDO QUE O MÉTODO Equals PODE CONSIDERAR MAIS DE UM CAMPO

//return Email.Equals(other.Email) && Name.Equals(other.Name);

public override int GetHashCode()

{

return Email.GetHashCode();//MESMA LÓGICA DE NEGÓCIO

}

LEMBRANDO QUE, O MÉTODO PODE CONSIDERAR MAIS DE UM CAMPO PARA O HASHCODE

//return Email.GetHashCode() + Name.GetHashCode();

E NO PROGRAMA PRINCIPAL

Client client1 = new Client("Luciano", "luciano@gmail");

Client client2 = new Client("Maria", "maria@gmail");

Client client3 = new Client("Carlos", "luciano@gmail");

Console.WriteLine(client1.Equals(client2));//false

Console.WriteLine(client1.Equals(client3));//true

Console.WriteLine(client1.GetHashCode());//HASH

Console.WriteLine(client2.GetHashCode());//HASH DIFERENTE

Console.WriteLine(client3.GetHashCode());//HASH

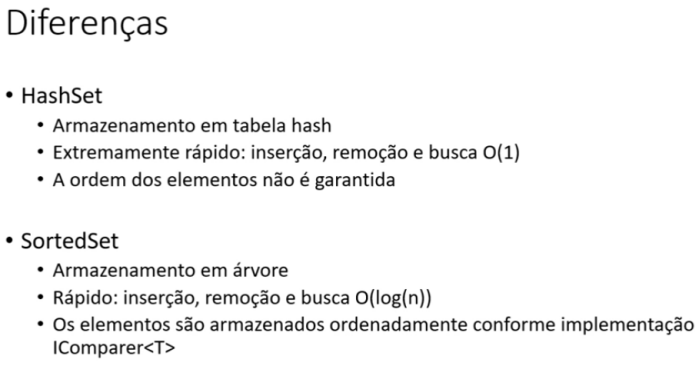
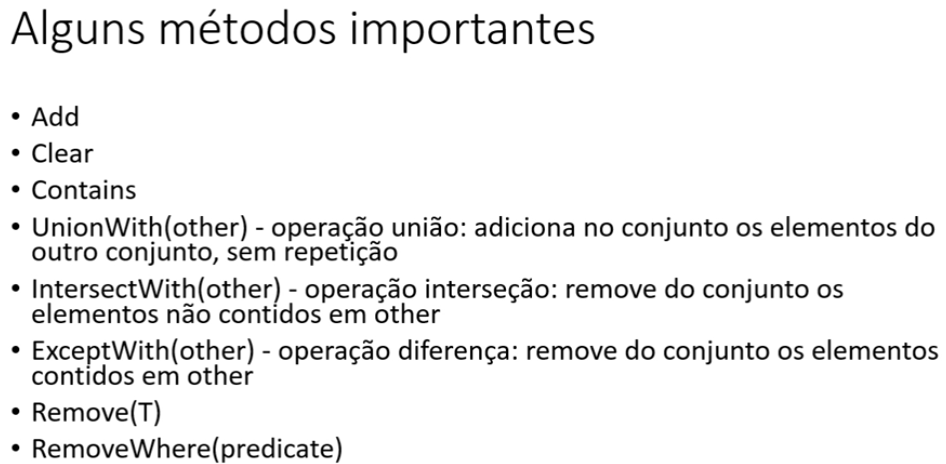
LEMBRANDO QUE USAR == PARA COMPARAR OBJETOS GERA false, DADO QUE O == NÃO COMPARA SEU CONTEÚDO, MAS SUA REFERÊNCIA DE MEMÓRIA.

**-----CONJUNTOS - HashSet<T> E SortedSet<T>------**

NÃO ADMITEM REPETIÇÕES

NÃO POSSUEM POSIÇÕES - PERCORRER COM FOREACH (NÃO FOR)

OPERAÇÕES RÁPIDAS - INSERÇÃO, REMOÇÃO, ACESSO, INTERSEÇÃO, UNIÃO, DIFERENÇA

**HashSet**

HashSet<string> colecao\_set = new HashSet<string>();

colecao\_set.Add("Luciano");

colecao\_set.Add("Ana");

colecao\_set.Add("Carlos");

foreach(string nome in colecao\_set)

{

Console.WriteLine(nome);

}

**SortedSet**

SortedSet<int> a = new SortedSet<int>() {0, 4, 2, 5, 8, 6, 10 };

SortedSet<int> b = new SortedSet<int>() {5, 6, 7, 8, 9, 10};

//ALGUMAS OPERAÇÕES

//COLEÇÃO INICIADA COM TODOS ELEMENTOS DA OUTRA

SortedSet<int> c = new SortedSet<int>(a);

//UNINDO DOIS CONJUNTOS - FINAL SÓ COM ELEMENTOS DIFERENTES - NÃO ACEITA REPETIÇÃO

c.UnionWith(b);

//COLEÇÃO INICIADA COM TODOS ELEMENTOS DA OUTRA

SortedSet<int> d = new SortedSet<int>(a);

//INTERSEÇÃO - SOMENTE ELEMENTOS EM COMUM

d.IntersectWith(b);

//COLEÇÃO INICIADA COM TODOS ELEMENTOS DA OUTRA

SortedSet<int> e = new SortedSet<int>(a);

//DIFERENÇA - TIRA DE A O QUE TIVER EM COMUM COM B

e.ExceptWith(b);

//MÉTODO ESTÁTICO PARA PERCORRER A COLEÇÃO

//IEnumerable - INTERFACE PARA GetEnumerate

//GetEnumerate - PERCORRER COLEÇÕES

//TODOS TIPOS NATIVOS SUPORTAM IEnumerate

static void PrintCollection<T>(IEnumerable<T> collection)

{

foreach(T item in collection)

{

Console.Write($"{item} ");

}

Console.WriteLine();

}

NO FUNCIONAMENTO INTERNO, HASH COLLECTIONS(HashSet, SortedSat, Map) USAM OS MÉTODOS GetHashCode (primeiro) E Equals(PARA CONFIRMAR) PARA FAZER COMPARAÇÕES NO SEU FUNCIONAMENTO. POR EXEMPLO, PARA USAR O MÉTODO CONTAINS, PROCURANDO UMA STRING NA COLEÇÃO, ESSA É A MANEIRA DE COMPARAR.

MAS PARA ESTRUTURAS QUE NÃO TENHAM ESSES MÉTODOS IMPLEMENTADOS, O QUE ACONTECE?

PARA CLASSES - COMPARAM OS ENDEREÇOS DE MEMÓRIA

PARA STRUCTS - COMPARAM O CONTEÚDO

OU SEJA

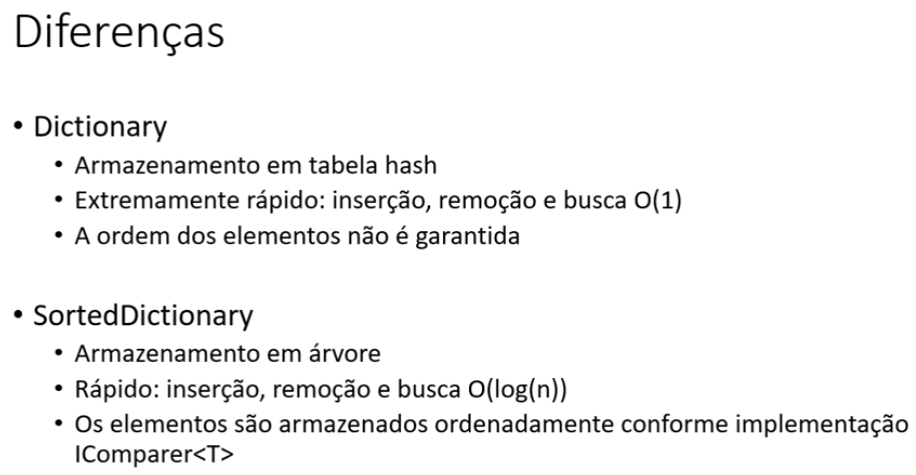
COMPARAR UM OBJETO COM AS MESMAS INFORMAÇÕES DE UM OBJETO DENTRO DA COLEÇÃO - DIFERENTES

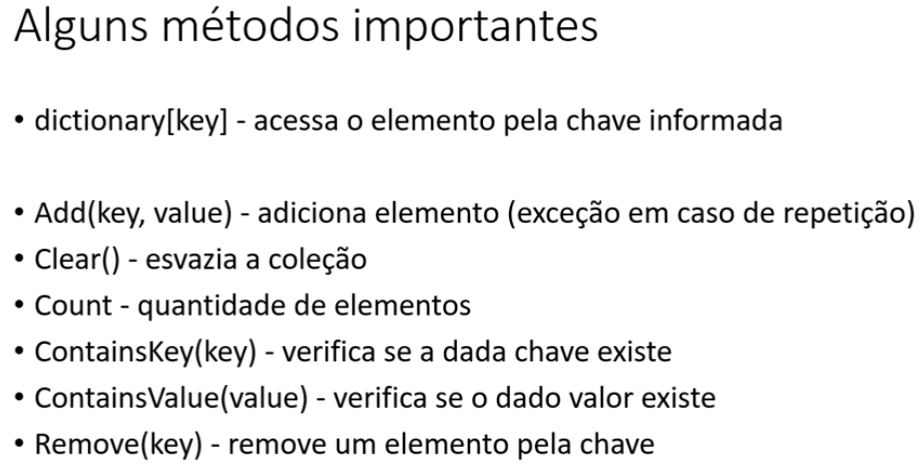
COMPARAR UMA VARIÁVEL STRUCT COM AS MESMAS INFORMAÇÕES DE OUTRA IGUAL - IGUAIS

**-----DICTIONARY E SORTEDDICTIONARY-----**

ESTRUTURAS DE DADOS COM CHAVE E VALOR - SIMILAR A HASHSET E SORTEDSET







//DECLARAÇÃO

//<string, string> - TIPO CHAVE, TIPO VALOR

Dictionary<string, string> cookie = new Dictionary<string, string>();

//ATRIBUIÇÕES ÀS CHAVES VALORES

cookie["user"] = "maria";

cookie["email"] = "maria@gmail.com";

cookie["phone"] = "33991457895";

cookie["phone"] = "31995784985";//SOBRESCREVE O ÚLTIMO VALOR

**--PERCORRER DICTIONARY--**

foreach(KeyValuePair<string, string> item in cookie)

{

Console.WriteLine($"{item.Key} : {item.Value}");

}

OU, USANDO var

foreach(var item in cookie)

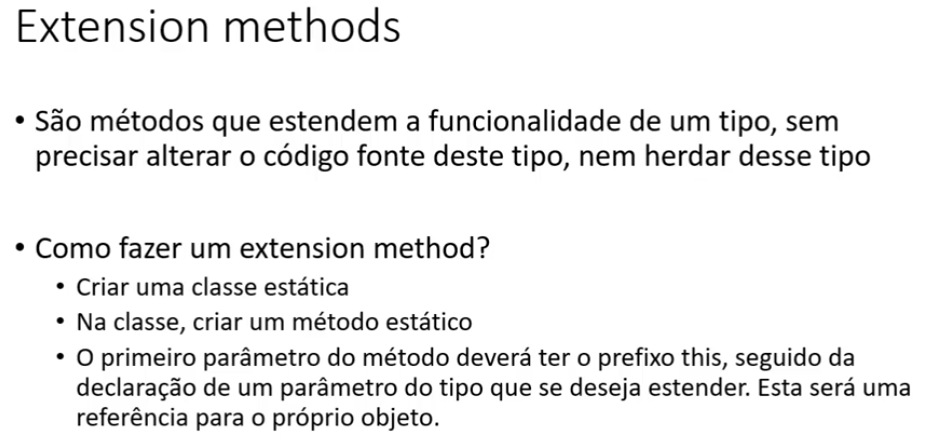
{

Console.WriteLine($"{item.Key} : {item.Value}");

}

**-----EXTENSION METHODS-----**

SÃO MÉTODOS ESTÁTICOS DE CLASSES ESTÁTICAS, PORÉM, UTILIZADOS POR MEIO DE VARIÁVEIS/OBJETOS.



EXEMPLO: CRIAR UM MÉTODO PARA VARIÁVEIS DATETIME QUE MOSTRA QUANTO TEMPO SE PASSOU DESDE A VARIÁVEL ATÉ O MOMENTO ATUAL, EM DIAS OU HORAS (CASO SEJA MENOS QUE 24H)

NO CÓDIGO PRINCIPAL

Console.Write("Digite uma data passada [DD/MM/YYYY]:");

DateTime data = DateTime.Parse(Console.ReadLine());

while (data >= DateTime.Now)

{

Console.WriteLine("Not a past data. Try again");

Console.Write("Digite uma data passada [DD/MM/YYYY]:");

data = DateTime.Parse(Console.ReadLine());

}

Console.WriteLine(data.ElapsedTime());

NA CLASSE

namespace ProjetoOO\_12.Extensions

{

static class DateTimeExtension

{

//this DateTime thisObj - SIGNIFICA QUE NO CÓDIGO PRINCIPAL, QUANDO FOR CHAMADA,

//A FUNÇÃO VAI USAR A VARIÁVEL DateTime QUE CHAMOU O MÉTODO

public static string ElapsedTime(this DateTime thisObj)

{

TimeSpan duration = DateTime.Now.Subtract(thisObj);

Console.WriteLine("Time elapsed: ");

if(duration.TotalHours < 24)

{

return duration.TotalHours.ToString("f1", CultureInfo.InvariantCulture)+" hours";

}

else

{

return duration.TotalDays.ToString("f1", CultureInfo.InvariantCulture)+" days";

}

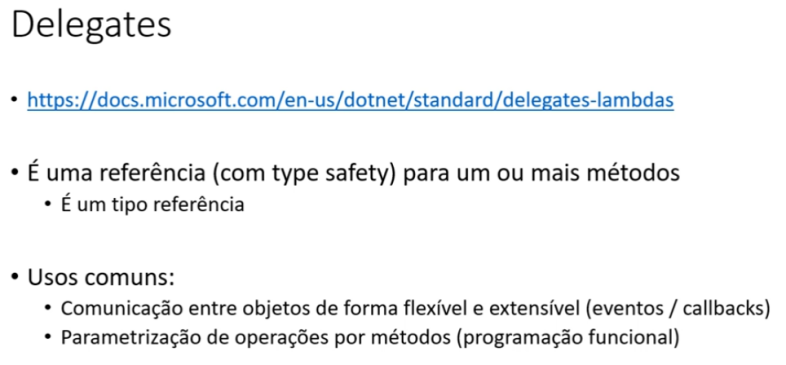
}

}

}

\*DICA: NA CONSTRUÇÃO DE UM MÉTODO PARA UMA VARIÁVEL DE TIPO ESPECÍFICO, COLOCAR O NAMESPACE SENDO O MESMO DO TIPO DA VARIÁVEL. POR EXEMPLO, NESSE CASO, O MÉTODO É PARA UMA VARIÁVEL DateTime, QUE É DO NAMESPACE System, SE NA ESTRUTURAÇÃO DA CLASSE DateTimeExtension, for usado namespace System TAMBÉM, NO PROGRAMA PRINCIPAL, AO USAR UMA VARIÁVEL DateTime E IMPORTAR SEU namespace (System), JÁ IMPORTA A CLASSE DateTimeExtension TAMBÉM.

-----DELEGATES-----



DELEGATE É UMA MANEIRA DE REFERENCIAR UM MÉTODO ATRIBUINDO ELE A UMA VARIÁVEL.

SÓ FUNCIONA SE O DELEGATE FOR CONSTRUÍDO COM A MESMA ASSINATURA DA FUNÇÃO A QUE REFERENCIA.

POR EXEMPLO, UM DELEGATE QUE ACEITA DOIS DOUBLE E DEVOLVE UM DOUBLE SÓ PODE REFERENCIAR MÉTODOS QUE FUNCIONEM DA MESMA FORMA.

EXEMPLO: FAZER COM QUE UMA VARIÁVEL operation RECEBA UM MÉTODO DA CLASSE CalculationService QUE SOMA DOIS DOUBLE E RETORNA O RESULTADO

NA CLASSE

class CalculationService

{

public static double Max(double x, double y)

{

return (x > y) ? x : y;

}

public static double Sum(double x, double y)

{

return x + y;

}

}

NO PROGRAMA PRINCIPAL

namespace ProjetoOO\_13

{

//DECLARAÇÃO DO DELEGATE NOME issoEhUmDelegate

delegate double issoEhUmDelegate(double n1, double n2);

class Program

{

static void Main(string[] args)

{

double a = 10;

double b = 5;

//VARIÁVEL operation AGORA É UMA REFERÊNCIA PARA O MÉTODO Sum

issoEhUmDelegate operation = CalculationService.Sum;

//TAMBÉM FUNCIONA ASSIM

//issoEhUmDelegate operation = new issoEhUmDelegate(CalculationService.Sum);

double resultado = operation(a, b);//Sum(a,b)

//TAMBÉM FUNCIONA ASSIM

//double resultado = operation.Invoke(a, b);//Sum(a,b)

Console.WriteLine(resultado);

**---MULTICAST DELEGATE---**

SÃO DELEGATES QUE AGREGAM A REFERÊNCIA A MAIS DE UM MÉTODO. QUANDO CHAMADOS, EXECUTAM TODOS NA ORDEM QUE FORAM REFERENCIADOS

NA CLASSE

class CalculationService

{

public static void ShowMax(double x, double y)

{

double max = (x > y) ? x : y;

Console.WriteLine($"Max: {max}");

}

//FUNÇÃO QUE VAI USAR O DELEGATE

public static void ShowSum(double x, double y)

{

Console.WriteLine($"Sum: {x + y}");

}

}

NO PROGRAMA PRINCIPAL

namespace ProjetoOO\_13

{

//DECLARAÇÃO DO DELEGATE NOME issoEhUmDelegate

delegate void issoEhUmDelegate(double n1, double n2);

class Program

{

static void Main(string[] args)

{

double a = 10;

double b = 5;

//VARIÁVEL operation AGORA É UMA REFERÊNCIA PARA OS MÉTODOS ShowSum e ShowMax

issoEhUmDelegate operation = CalculationService.ShowSum;

operation += CalculationService.ShowMax;

operation.Invoke(a, b);

**---DELEGATE PREDICATE---**

DELEGATE QUE SÓ ACEITA OBJETO TIPO GENÉRICO E RETORNA BOOL

EXEMPLO, RETIRAR DE UMA LISTA SOMENTE OS PRODUTOS COM PREÇO >= 100

CONSIDERANDO UMA CLASSE PRODUTO COM NOME E PREÇO

NO PROGRAMA PRINCIPAL

List<Product> produtos = new List<Product>();

produtos.Add(new Product("TV", 900));

produtos.Add(new Product("MOUSE",50));

produtos.Add(new Product("TABLET",350));

produtos.Add(new Product("HD",80.9));

//USANDO MÉTODO ESTÁTICO ProductTest - FORA DO MAIN

//MÉTODO QUE ACEITA TIPO GENÉRICO Product E RETORNA bool

//produtos.RemoveAll(ProductTest);

//TAMBÉM FUNCIONA ASSIM

//FUNÇÃO RemoveAll USA PREDICATE INTERNAMENTE

//REMOVE O PRODUTO p SE O PREÇO FOR >= 100 - FUNÇÃO LAMBDA

produtos.RemoveAll(p => p.Price >= 100);

foreach (Product produto in produtos)

{

Console.WriteLine(produto);

}

**---DELEGATE ACTION---**

ACEITA UM OU MAIS TIPOS GENÉRICOS, REALIZA UMA AÇÃO E NÃO TEM RETORNO.

EXEMPLO: DADA UMA LISTA DE PRODUTOS, AUMENTAR EM 10% SEUS VALORES

PODERIA SER FEITO COM UM LAÇO FOREACH, MAS USANDO ACTION É MELHOR (PQ?)

NO PROGRAMA PRINCIPAL

static void Main(string[] args)

{

List<Product> produtos = new List<Product>();

produtos.Add(new Product("TV", 900));

produtos.Add(new Product("MOUSE",50));

produtos.Add(new Product("TABLET",350));

produtos.Add(new Product("HD",80.9));

//USO DO DELEGATE ACTION PELA FUNÇÃO ForEach - NATIVO DE LISTAS

//QUE RECEBE O MÉTODO UpdatePrice COMO PARÂMETRO

//UpdatePrice É UM MÉTODO ESTÁTICO VOID QUE ACEITA UM TIPO GENÉRICO

//OU SEJA, PODE SER USADO COM DELEGATE ACTION

//UpdatePrice AUMENTA EM 10% TODOS VALORES

produtos.ForEach(UpdatePrice);

//TAMBÉM FUNCIONA ASSIM - OBJETO TIPO ACTION

//Action<Product> action = UpdatePrice;

//produtos.ForEach(action);

//TAMBÉM FUNCIONA ASSIM - FUNÇÃO LAMBDA

//Action<Product> action = p => { p.Price += p.Price \* 0.1; };

foreach (Product produto in produtos)

{

Console.WriteLine(produto);

}

}

//MÉTODO USADO NO ACTION - ACEITA TIPO GENÉRICO E NÃO TEM RETORNO

static void UpdatePrice(Product produto)

{

produto.Price += produto.Price \* 0.1;//AUMENTO 10%

}

**---DELEGATE FUNC---**

ACEITA UM OU MAIS TIPOS GENÉRICOS, REALIZA UMA AÇÃO E NÃO TEM UM RETORNO.

EXEMPLO: A PARTIR DE UMA LISTA DE PRODUTOS, DEVOLVER UMA LISTA COM OS NOMES DOS PRODUTOS EM CAIXA ALTA

static void Main(string[] args)

{

List<Product> produtos = new List<Product>();

produtos.Add(new Product("tv", 900));

produtos.Add(new Product("mouse",50));

produtos.Add(new Product("tablet",350));

produtos.Add(new Product("hd",80.9));

//A NOVA LISTA RESULTADO RECEBE PARA CADA ITEM DE produtos, O RESULTADO DA FUNÇÃO

//NameUpper APLICADA A ELES. - UMA LISTA DE STRINGS

List<string> resultado = produtos.Select(NameUpper).ToList();

//TAMBÉM FUNCIONA ASSIM

//Func<Product, string> func = NameUpper;

//List<string> resultado = produtos.Select(func).ToList();

//TAMBÉM FUNCIONA ASSIM - FUNÇÃO LAMBDA - NÃO É PRECISO O MÉTODO ESTÁTICO

//Func<Product, string> func = p => p.Name.ToUpper();

//List<string> resultado = produtos.Select(func).ToList();

//TAMBÉM FUNCIONA ASSIM - FUNÇÃO LAMBDA - NÃO É PRECISO O MÉTODO ESTÁTICO

//List<string> resultado = produtos.Select(p => p.Name.ToUpper()).ToList();

foreach (Product produto in produtos)

{

Console.WriteLine(produto);

}

Console.WriteLine();

foreach (string nome in resultado)

{

Console.WriteLine(nome);

}

}

//MÉTODO USADO NO FUNC - ACEITA TIPO GENÉRICO E TEM RETORNO STRING

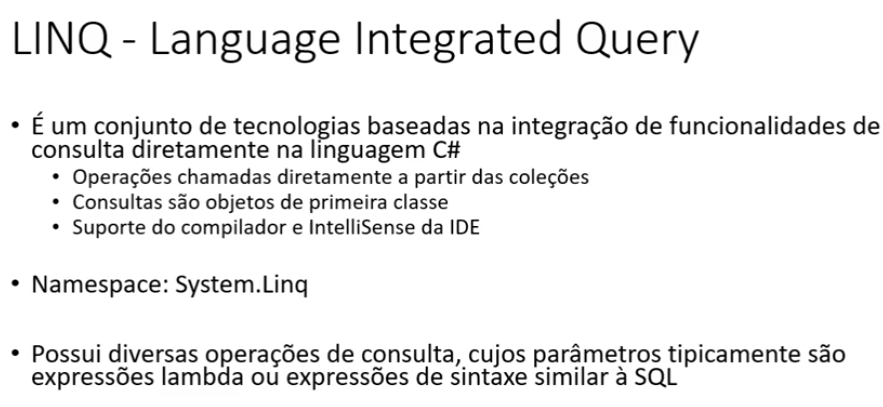
static string NameUpper(Product p)

{

return p.Name.ToUpper();

}

-----LINQ-----

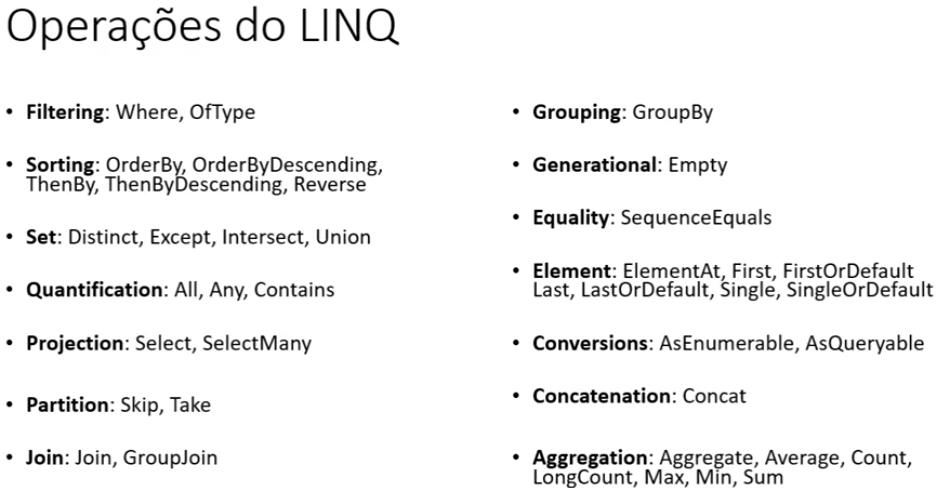


É UM CONJUNTO DE MÉTODOS DE EXTENSÃO QUE FUNCIONAM COM COLEÇÕES.

MELHOR QUE FAZER CONSULTAS USANDO STRINGS, AS QUAIS O COMPILADOR NÃO APONTA ERROS DE SINTAXE.

MÉTODOS LINQ GERAM LISTA IEnumerable - GENÉRICAS.

SE FOR PRECISO CONVERTER SEU RESULTADO PARA LISTA, USAR ToList() AO FINAL.



PASSO A PASSO: CRIAR A FONTE DE DADOS -> DEFINIR A QUERY -> EXECUTAR A QUERY

PROGRAMA QUE, DADA UMA LISTA DE NÚMEROS INTEIROS, SELECIONA OS PARES E OS MULTIPLICA POR 10, SALVANDO O RESULTADO NUMA LISTA GENÉRICA

static void Main(string[] args)

{

//FONTE DE DADOS

int[] numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

//QUERY

//SELECIONAR VALORES PARES

//E MULTIPLICÁ-LOS POR 10

IEnumerable<int>resultado = numbers.Where(x => x % 2 == 0)

.Select(x => x \* 10);

//IEnumerable - UMA LISTA GENÉRICA

//EXECUTAR CONSULTA

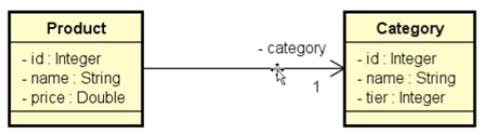
foreach(int x in resultado)

{

Console.WriteLine(x);

}

PROGRAMA QUE, DADA UMA LISTA DE PRODUTOS, FAZ AS SEGUINTES OPERAÇÕES



Category c1 = new Category() { Id = 1, Name = "Tools", Tier = 2 };

Category c2 = new Category() { Id = 2, Name = "Computers", Tier = 1 };

Category c3 = new Category() { Id = 3, Name = "Electronics", Tier = 1 };

List<Product1> products = new List<Product1>() {

new Product1() { Id = 1, Name = "Computer", Price = 1100.0, Category = c2 },

new Product1() { Id = 2, Name = "Hammer", Price = 90.0, Category = c1 },

new Product1() { Id = 3, Name = "TV", Price = 1700.0, Category = c3 },

new Product1() { Id = 4, Name = "Notebook", Price = 1300.0, Category = c2 },

new Product1() { Id = 5, Name = "Saw", Price = 80.0, Category = c1 },

new Product1() { Id = 6, Name = "Tablet", Price = 700.0, Category = c2 },

new Product1() { Id = 7, Name = "Camera", Price = 700.0, Category = c3 },

new Product1() { Id = 8, Name = "Printer", Price = 350.0, Category = c3 },

new Product1() { Id = 9, Name = "MacBook", Price = 1800.0, Category = c2 },

new Product1() { Id = 10, Name = "Sound Bar", Price = 700.0, Category = c3 },

new Product1() { Id = 11, Name = "Level", Price = 70.0, Category = c1 }

};

var r1 = products.Where(p => p.Price < 900 && p.Category.Tier == 1);

Print("PRODUTOS TIER 1 E PREÇO < 900",r1);

//LÊ-SE

//Where(p => p.Price) -OBJETOS P ONDE O PREÇO P....

var r2 = products.Where(p => p.Category.Name == "Tools")

.Select(p => p.Name);

Print("PRODUTOS CATEGOARIA TOOLS", r2);

//LÊ-SE

//.Select(p=>p.Name) - SELECIONANDO P E TRANSFORMANDO EM p.NAME

var r3 = products.Where(p => p.Name[0] == 'C')

.Select(p => new { p.Name,p.Price, CategoryName = p.Category.Name });

Print("PRODUTOS QUE COMEÇAM COM A LETRA C, MOSTRAR NOME, PREÇO E CATEGORIA",r3);

//new { p.Name,p.Price, CategoryName = p.Category.Name }

//OBJETO ANÔNIMO QUE OBTEM AS INFORMAÇÕES DO PRODUTO (NOME, PREÇO E NOME DA CATEGORIA)

//CategoryName = p.Category.Name - É ASSIM PQ OS CAMPOS TEM O MESMO NOME, ENTÃO CRIA-SE UM APELIDO

//PARA UM DELE

var r4 = products.Where(p => p.Category.Tier == 1)

.OrderBy(p => p.Price)

.ThenBy(p => p.Name);

Print("PRODUTOS DE CATEGORIA TIER 1, ORDENADOS POR: PREÇO -> NOME", r4);

//LÊ-SE

//OrderBy(p => p.Price) - ORDENADOS OBJETOS P CONSIDERANDO O p.Price

//ThenBy(p => p.Name) - E DEPOIS, CASO HAJA "EMPATE", ORDENAR OBJETOS P CONSIDERANDO p.Name

var r5 = r4.Skip(2).Take(4);

Print("PRODUTOS DE CATEGORIA TIER 1, ORDENADOS POR: PREÇO -> NOME\n" +

"PULA OS DOIS PRIMEIROS - MOSTRA OS 4 PRÓXIMOS", r5);

var r6 = products.First();

//var r6 = products.FirstOrDefault();

Console.WriteLine($"PRIMEIRO ELEMENTO DA LISTA: {r6}\n");

//CASO SEJA UMA LISTA VAZIA, GERA ERRO.

//MELHOR USAR .FirstOrDefault() - QUE NÃO MOSTRA NADA CASO NÃO HAJA ELEMENTOS

var r8 = products.Where(p => p.Id == 3)

.SingleOrDefault();

Console.WriteLine($"PRODUTO COM ID 3 - USANDO SingleOrDefault PARA GARANTIR UM ÚNICO ELEMENTO\n" +

$"{r8}\n");

//GERA EXCEÇÃO CASO O RESULTADO SEJA MAIS QUE UM ITEM

//SÓ ACEITA UM OU NENHUM ELEMENTO

var r10 = products.Max(p => p.Price);

Console.WriteLine($"MAIOR PREÇO: {r10}\n");

//MAX PARA NUMÉRICO - MAIOR VALOR

//MAX PARA STRING - MAIS DISTANTE NA ORDEM ALFABÉTICA

var r11 = products.Min(p => p.Price);

Console.WriteLine($"MENOR PREÇO: {r11}\n");

//MAX PARA NUMÉRICO - MAIOR VALOR

//MAX PARA STRING - MAIS DISTANTE NA ORDEM ALFABÉTICA

var r12 = products.Where(p => p.Category.Id == 1)

.Sum(p => p.Price);

Console.WriteLine($"SOMA DE TODOS VALORES DE PRODUTOS DA CATEGORIA 1: {r12}\n");

//var r13 = products.Where(p => p.Category.Id == 1).Average(p => p.Price);

//Console.WriteLine($"MÉDIA DE TODOS VALORES DE PRODUTOS DA CATEGORIA 1: {r13}\n");

//MÉDIA GERA EXEÇÃO CASO SEJA MÉDIA DE UM CONJUNTO VAZIO. PARA REMEDIAR ISSO, USA-SE O CAMPO QUE

//SERIA CALCULADA A MÉDIA PARA FORMAR UMA LISTA DE DOUBLEs (USANDO SELECT) JUNTO COM A FUNÇÃO

//DefaultIfEmpty(0.0) E SOBRE TODA ESSA EXPRESSÃO, APLICA-SE O Average

var r13 = products.Where(p => p.Category.Id == 1)

.Select(p => p.Price)

.DefaultIfEmpty(0)//ATÉ AQUI, É GERADA UMA LISTA COM OS VALORES - SE FOR VAZIA, 0

.Average();//MÉDIA DE 0 É 0

Console.WriteLine($"MÉDIA DE TODOS VALORES DE PRODUTOS DA CATEGORIA 1: {r13}\n");

//AGGREGATE - FUNÇÃO PERSONALIZÁVEL PARA PROCESSAR DADOS

var r14 = products.Where(p => p.Category.Id == 1)//PRODUTOS DE CATEGORIA 1

.Select(p => p.Price) //SELECIONO O PREÇO DELES

.Aggregate(0.0, (x, y) => x + y); //AGREGO OS PREÇOS FAZENDO A SOMA

//0.0 - RETORNA 0 CASO SEJA AGGREGATE DE UM CONJUNTO VAZIO

//(x, y) => x + y - FUNÇÃO ANÔNIMA - RETORNA A SOMA DOS PREÇOS

Console.WriteLine($"SOMA DE TODOS OS PREÇOS DOS PRODUTOS DE CATEGORIA 1 (COM AGGREGATE): {r14}\n");

//GROUPBY - AGRUPA OS DADOS EM GRUPOS CONSIDERANDO UM CRITÉRIO/CAMPO

//PRODUTOS AGRUPADOS POR CATEGORIA

var r15 = products.GroupBy(p => p.Category);

//FOR EACH ESPECIAL PARA RESULTADOS DE GROUP BY

foreach(IGrouping<Category, Product1> grupo in r15)

{

Console.WriteLine($"Categoria: {grupo.Key.Name}");

foreach(Product1 p in grupo)

{

Console.WriteLine(p);

}

Console.WriteLine();

}

MÉTODO ESTÁTICO PRINT

//MÉTODO ESTÁTICO GENÉRICO PARA IMPRIMIR RESULTADOS

static void Print<T>(string message, IEnumerable<T> collection)

{

Console.WriteLine(message);

foreach (T obj in collection)

{

Console.WriteLine(obj);

}

Console.WriteLine();

Console.WriteLine();

}