C# COMPLETO Programação Orientada a Objetos + Projetos

* Namespace: Agrupamento de classes relacionadas. É uma boa prática colocar as classes criadas dentro de namespaces;

# Primeiro Programa

* Visual Studio Code com a extensão C# instalada;
* O SDK do .net 5,0;
* Criar a pasta em que seu nome será o nome do projeto;
* Abrir o VSCode
* E no terminal (posicionado na pasta criada) executar: ***dotnet new console --framework net5.0***
* O código de modelo define uma classe, Program , com um único método, Main.
* Main é o ponto de entrada do aplicativo, o método que é chamado automaticamente pelo runtime quando ele inicia o aplicativo. Quaisquer argumentos de linha de comando fornecidos quando o aplicativo for iniciado estão disponíveis na matriz args.

Texto

Descrição gerada automaticamente

* Para executar o aplicativo: dotnet run

# Estrutura de um Program C#

* .csproj: Arquivo de configuração
* .cs: Arquivo com o fonte em C#
* Pasta obj e bin: Arquivos compilados

# Lógica de Programação

* Principais tipos de dados:

Tabela

Descrição gerada automaticamente

* Declaração: int x = 10;
* Overflow: Cálculo extrapola o tamanho da variável fazendo com que a variável receba o valor do limite oposto (Exemplo1: byte n1 = 255 // n1++ // Resultado = 0; Exemplo2: sbyte n1 = 127 // n1++ // Resultado = -128)
* Ao utilizar long, acrescentar o “L” no final como boas práticas: long y = 246298356L
* Ao declarar um char, é possível utilizar o código Unicode (Exemplo: char letra = ’\u0041’)
* Char: Utilizar aspas simples; Strings: aspas duplas
* Variável float: utilizar um ‘f’ no final (Exemplo: float a = 1.75f)

Interface gráfica do usuário, Texto, Aplicativo, chat ou mensagem de texto

Descrição gerada automaticamente

* Toda classe em C# é uma subclasse de object, logo, ele pode receber tipos float, string, char e etc.

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

# Algumas Convenções

* Camel Case: lastName (parâmetros de métodos, variáveis dentro de métodos)
* Pascal Case: LastName (namespaces, classe, properties e métodos)
* Padrão \_lastName (atributos "internos" da classe)

# Combinar Strings

* PlaceHolders: Console.WriteLine("{0} tem {1} anos e tem saldo igual a {2:F2} reais", nome, idade, saldo);
* Interpolação: Console.WriteLine($"{nome} tem {idade} anos e tem saldo igual a {saldo:F2} reais");
* Concatenação: Console.WriteLine(nome + " tem " + idade + " anos e tem saldo igual a " + saldo.ToString("F2", CultureInfo.InvariantCulture) + " reais"); Necessário: using System.Globalization;

# Operadores de Atribuição:

Tabela

Descrição gerada automaticamente

# Conversão de tipos

* <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/numeric-conversions>
* Conversão implícita: Quando a conversão de tipos de variáveis pode ocorrer naturalmente. Ex.: Converter de float -> double (de 4 para 8 bits);
* Casting (conversão explícita): b = (float)a (no caso o tipo de b é double); b = (int)a
* Se eu dividir um int por outro int, o compilador entende que eu quero apenas a parte inteira dessa divisão (mesmo que eu esteja atribuindo o valor p um double ou float) então é necessário utilizar o casting para receber o valor real dessa operação: int a; int b; double div = (double) a/b;

# Operadores Aritméticos

* +, -, \*, /, %

# Entrada de Dados

* Console.ReadLine() -> Stgring;
* Split: string[] vet = s.split(‘<Separador>’)
* Conversão de string pra outro tipo: <tipo>.Parse(Console.ReadLine()). Ex.: int.Parse(Console.ReadLine())

# Operadores Comparativos

Tabela

Descrição gerada automaticamente

# Operadores Lógicos

Tabela

Descrição gerada automaticamente

* Precedência: ! > && > ||

# Estrutura Condicional

Interface gráfica do usuário, Aplicativo, Word

Descrição gerada automaticamente

# Funções

* Sintaxe:

static <tipo retorno> Nome\_Funcao(<tipo> a, <tipo> b, ...) {

<comandos>

Return x

}

# Estrutura de repetição

* While:

Word

Descrição gerada automaticamente com confiança baixa

* For:

Linha do tempo

Descrição gerada automaticamente

# Classes

* As classes é um tipo estruturado que podem conter atributos (dados) e métodos (funções/operações);
* Ainda podem possuir construtores, sobrecarga, encapsulamento, herança e polimorfismo;
* Vantagens:
* Reaproveitamento de código: elimina-se código repetido no programa principal.
* Delegação de responsabilidades: quem deve ser responsável por saber como calcular a área de um triângulo é o próprio triângulo. A lógica do cálculo da área não deve estar em outro lugar.
* Classe (exemplo):

namespace NomeNameSpace {

public class nomeClasse {

public <tipo> atributo1; //Public para conseguir acessar os atributos

<...>}}

* Para instanciar a classe a classe seria como se fosse um tipo:

nomeClasse x, y;

x = new nomeClasse();

y = new nomeClasse();

Diagrama

Descrição gerada automaticamente

* Métodos: São funções atribuídas a essa classe, logo são declaradas dentro do escopo da classe;

using *System*;

namespace *\_003\_Sld39\_Classes*{

    public class Triangulo{

        public *double* ladoA; //Public para conseguir acessar os atributos

        public *double* ladoB;

        public *double* ladoC;

        public *double* CalcArea(){

*double* p = (ladoA + ladoB + ladoC) / 2;

*double* area = Math.Sqrt(p \* (p - ladoA) \* (p - ladoB) \* (p - ladoC));

            return area;

        }}}

Resumo:

Diagrama

Descrição gerada automaticamente

Projeto da Classe (UML):

Diagrama

Descrição gerada automaticamente

* Toda classe em C# é subclasse de Object.
* GetType: Tipo do objeto;
* Equal: Compara se o objeto é igual ao outro;
* GetHashCode: Codigo hash do objeto;
* ToString: Converte o objeto para string;
* Sobreposição de método:

        public override *string* ToString()

        {

            return "\nProduto: " + Nome

                + "\nPreço: R$" + Preco.ToString("F2")

                + "\nQuantidade em estoque: " + Quantidade

                + "\nValor em estoque " + ValEstoque().ToString("F2");

        }

* Para chamar o método sobrescrito basta concatenar o objeto: Console.WriteLine("Dados do Produto: " + prod1);

# Membros Estáticos

* Membros estáticos (membros de classe) são aqueles independentes de objetos, ou seja, não precisam de um objeto instaciado para serem chamados. Utiliza-se o próprio nome da classe para chamá-lo (Ex.: Math.Sqrt(double));
* Se a classe possui apenas membros estáticos pode ser chamada de classe estática;
* Criando o método estático:

public static <tipo\_retorno> nomeMetodo(<tipo> <param>){

(...)

return retorno

}

* Chamando o método estático: nomeClasse.nomeMetodo(param)

# Construtores

* Operação que é executada no momento da instanciação do objeto;
* É utilizado para iniciar o objeto com alguns atributos já definidos;
* É possível especificar mais de um construtor (sobrecarga);
* Sintaxe:

    public class Produto

    {

        public *string* Nome;

        public *double* Preco;

        public *int* Quantidade;

        public Produto(*string* *nome*, *double* *preco*) //Construtor

        {

            Nome = nome;

            Preco = preco;

        }

(...)

* Para instanciar:

*Produto* prod1 = new *Produto*(nome, preco);

# Sobrecarga

* Recurso em que uma classe oferece mais de uma operação com mesmo nome, mas com diferentes parâmetros.
* Sintaxe: os dois construtores:

        public Produto(*string* *nome*, *double* *preco*)

        { //Construtor

            Nome = nome;

            Preco = preco;

            Quantidade = 0;

        }

        public Produto(*string* *nome*, *double* *preco*, *int* *quantidade*){

            Nome = nome;

            Preco = preco;

            Quantidade = quantidade;

        }

        public Produto(){ //Permitir o Construtor padrão

        }

# Palavra This

* Referência para diferenciar atributos de variáveis na declaração de construtores (nesse caso não é muito utilizado em C# pois é padronizado que os atributos iniciem-se com letra maíuscula):

Interface gráfica do usuário, Aplicativo, Word

Descrição gerada automaticamente

* Ao fazer a sobrecarga de construtores referenciar um outro construtor já definido:

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

# Encapsulamento

* Forma segura de expor e receber valores em atributos passando sempre por métodos (Get e Set);
* O objeto deve ser sempre consistente e a classe deve garantir isso;
* Para isso todos os atributos são definidos como private, tornando-os inacessíveis de forma direta pela aplicação.
* Get e Sets não são usuais em C#;
* Exemplo:

Texto, Linha do tempo

Descrição gerada automaticamente

# Properties

* Análogo a métodos encapsulados, mas com uma sintaxe similar à atributos;
* As propriedades podem ser usadas como se fossem atributos públicos, mas na verdade elas são métodos especiais.
* Permitem que os dados sejam acessados facilmente e ainda ajuda a promover a segurança e a flexibilidade dos métodos.

public string Nome{

            get { return \_nome; }

            set{

                if (value != null && value.Length >= 2){

                    \_nome = value.ToUpper

                }

                else{

                    Console.WriteLine("O deve ter mais de um caractere");

                }

            }

        }

# Auto Properties

* Utilizadas para simplificar o encapsulamento para em situações em que não se necessita de lógicas particulares;

private string \_nome;

private double \_preco;

public int Quantidade { get; private set; } // Auto properties

* No exemplo, apenas o get é permitido;
* A declaração é feita logo na declaração de atributos;
* Atalho: digitar “prop” e apertar “tab”;

# Ordem sugerida de Implementação de Membros de Classe

* Atributos privados; Propriedades Auto implementadas; Construtores; Propriedades customizadas; Outros métodos;

# Modificadores de Acesso

* Para membros:

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente com confiança média

* Para classes:

Interface gráfica do usuário, Texto, Aplicativo, chat ou mensagem de texto

Descrição gerada automaticamente

# Tipos Referência

* Tipos referência: Variáveis cujo tipo são classes elas apontam (fazem referência) para um endereço de memória específico:

Diagrama

Descrição gerada automaticamente

* Valor null: Variável foi alocada mas não recebeu nenhum valor ou não aponta pra ninguém;

# Tipo Valor (Tipos básicos)

* Também conhecidas como structs (são caixas e não ponteiros): Int, long, double, float (...).

Interface gráfica do usuário, Aplicativo, Word

Descrição gerada automaticamente

* É possível criar tipos structs:

Texto

Descrição gerada automaticamente

# Valores Padrão

* Quando alocamos qualquer tipo estruturado, valores padrão são atribuídos:
* Números: 0;
* Booleanos: False;
* Caractere: código ‘0’;
* Objeto: null;
* Uma variável só será acessada caso ela seja atribuída. Se for apenas declarada, o compilador não permite operações com elas;

# Tipo Classe x Tipo Struct

Texto, Tabela

Descrição gerada automaticamente

# Desalocação de Memória

* Garbage coletor: Processo que automatiza o gerenciamento de memória de um programa em execução.
* O garbage collector monitora os objetos alocados dinamicamente pelo programa (no heap), desalocando aqueles que não estão mais sendo utilizados.

No momento em que um objeto, armazenado no Heap, perde sua referência (variável na stack) o Garbage Coletor desaloca es objeto da memória em um momento futuro;

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente com confiança média

# Desalocação por Escopo

* Uma variável declarada dentro de um escopo específico, quando o programa sai desse escopo, a variável é desalocada automaticamente.

# Nullable

* Recurso para que structs recebam valor nulo (contexto: campos do banco de dados que não são obrigatórios e serão utilizados para algum processamento);
* Ao declarar a variável acrescentar “?” após o tipo:
* double x = null; //(erro)
* Nullable<double> x = null; //OK
* Forma simplificada: double? X = null; //OK
* Métodos:
* GetValueOrDefault: Retorna o valor ou default (<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/default>).
* HasValue: True (possui um valor) ou False (null).
* Value: Retorna o valor da variável (ou uma exceção se for null);
* Operador de coalescência nula (??): Atribui um valor a variável caso ela seja nula;
* Double y = x ?? 0.0 //Se x for nula y receberá 0.0.

# Vetores

* Arranjos de dados unidimensionais (todos elementos do mesmo tipo);
* Primeira posição 0;
* Alocado em um bloco contíguo de memória;
* Vantagens: Acesso imediato;
* Desvantagem: Tamanho fixo; inserção e deleção são dificultosos;
* Declaração:

double[] alturas = new double[num];

# Listas

* É uma estrutura de dados:
  + São homogêneas;
  + Ordenadas;
  + Podem ser iniciadas vazias e são alocadas sob demanda (se diferenciam dos arrays nesse ponto);
  + Cada elemento ocupa um nó.
* Pertencem a classe list;
* Namespace: System.Collections.Generic
* Vantagens:
  + Tamanho variável;
  + Fáceis de fazer inserir e deletar elementos;
* Desvantagem:
  + Acesso sequencial aos elementos (se quiser ir ao elemento x tem que passar item a item até alcançá-lo). É possível ter um acesso mais rápido a um determinado elemento (como?).
* Instanciação:
  + List<tipo> nameList = new list <string> (); //Lista Vazia
  + List<tipo> nameList = new list <string> {“str1”,”str2”,”str3”}; //Lista com elementos em sua instanciação.
* Operações
  + Adicionar elementos: list.Add(<elemento>);
  + Inserir (Especifica a posição que o elemento será adicionado (não exclui o elemento que estava na posição especificada)): list.Insert(pos, <elemento>);
  + Tamanho da lista: list.Count;
  + List.Find(x => x[0] == ‘A’)// -> Expressão lambda

COMECEI A FOCAR NOS TEMAS MAIS IMPORTANTES, POSTERIORMENTE, VOLTAREI REFAZENDO (SE NECESSÁRIO)

# Modificadores de Parâmetros (params)

* Solução para criar um método com n parâmetros de entrada:

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

# Modificadores de Parâmetros Ref e Out (Devem ser evitados)

* Ref: Passar para um método uma variável por referência. Logo, se eu alterar o valor da variável dentro do escopo da função, o valor da variável será alterado automaticamente para o escopo que o método foi chamado.
* O termo *ref* deve ser passado tanto na implementação do método quanto na chamada do mesmo.

Interface gráfica do usuário, Aplicativo, Word

Descrição gerada automaticamente

* Out: Assim como o *ref* faz o parâmetro ser uma referência para a variável original, mas não exige que a variável inicial seja iniciada.

Interface gráfica do usuário, Aplicativo, Word

Descrição gerada automaticamente

# Inferência de Tipos

* Deixar que o compilador defina o tipo de variável (var): var x = 10;

# Expressão Condicional Ternária

* (<condição>) ? <valor se verdadeiro> : <valor se falso>

# Algumas Funções pra String

* Formatar: ToLower(), ToUpper(), Trim();
* Buscar: IndexOf, LastIndexOf ;
* Recortar: Substring(inicio), Substring(inicio, tamanho);
* Substituir: Replace(char, char), Replace(string, string);
* String.IsNullOrEmpty(str), String.IsNullOrWhiteSpace(str);
* str.Split(' ');
* Conversão para numero: int x = int.Parse(str), int x =Convert.ToInt32(str);
* Conversão de número: str = x.ToString(), str = x.ToString("C"), str = x.ToString("C3"), str = x.ToString("F2");

Enumeração e Composição

# Enumeração

* É um tipo especial que serve para especificar de forma literal um conjunto de constantes relacionadas.

Texto

Descrição gerada automaticamente com confiança média

* UML:

Uma imagem contendo Interface gráfica do usuário

Descrição gerada automaticamente

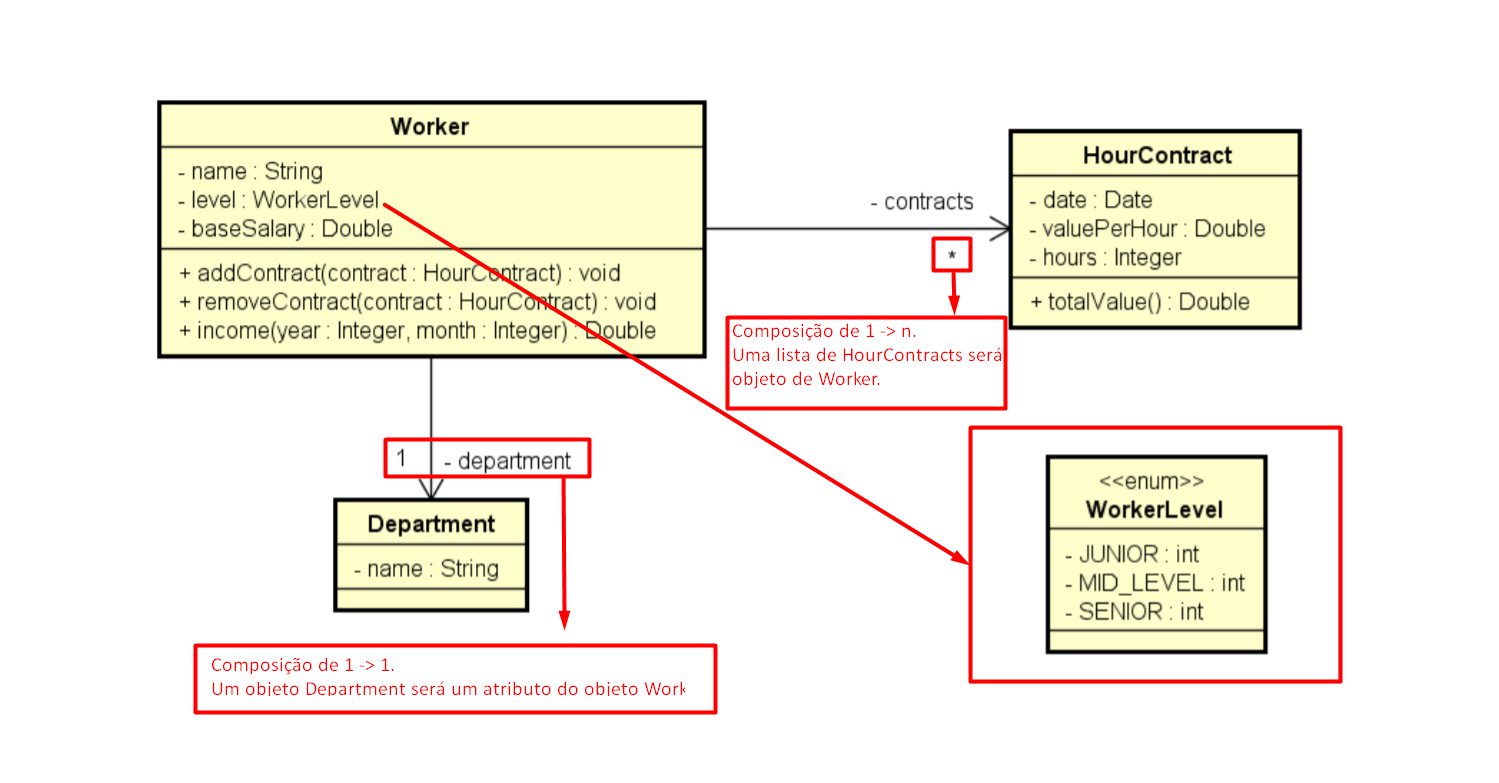
* Conversão string to enum:
  + OrderStatus os = Enum.Parse<OrderStatus>("Delivered");
  + OrderStatus os = (OrderStatus)Enum.Parse(typeof(OrderStatus), "Delivered");

# Falando sobre Design

* Em um sistema OO, tudo é objeto:
* Por questões de design tais como organização, flexibilidade, reuso, delegação etc., há várias categorias de classes:
  + Views: Telas;
  + Controllers: Meio de campo entre a tela e o sistema;
  + Entities: Classes;
  + Services: Serviços;
  + Repositories: Entidades de acesso a banco de dados;

# Composição

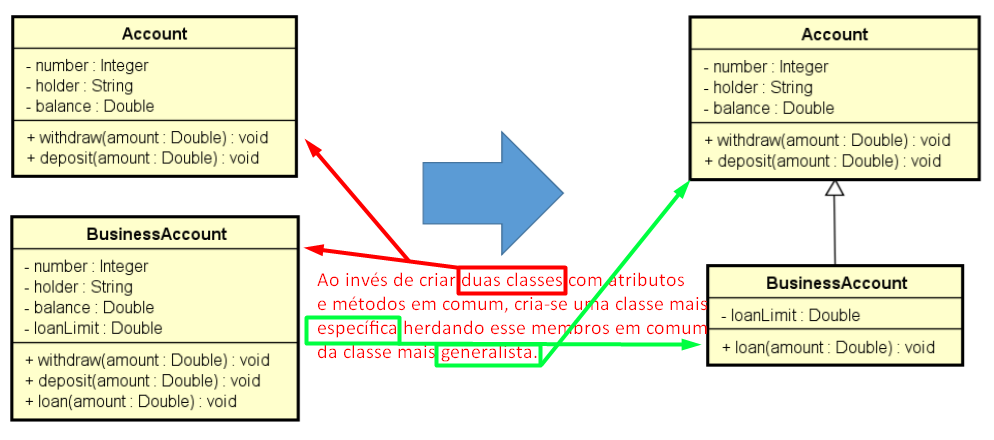
* Associação que permite que um objeto contenha outro;
* Vantagens:
  + Organização: Cada classe tem a sua responsabilidade;
  + Coesão: Cada objeto é responsável por uma única coisa. Responsabilidades bem definidas;
  + Flexibilidade;
  + Reuso: Objeto reaproveitado em mais de um lugar;



Herança e Polimorfismo

# Herança

* Tipo de associação que permite que uma classe herde dados de outra;
* Vantagens: Reuso e Polimorfismo;
* Sintaxe:
  + : (extende);
  + base (referência para a super-classe).



* Relação “é um”. A conta Business é uma “conta”;
* Generalização/Especialização;
* A subclasse estende a superclasse;
* Herança é uma associação entre classes e não entre objetos.

Texto, Linha do tempo

Descrição gerada automaticamente com confiança média

* Relembrando os modificadores de acesso:

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente com confiança média

# Upcasting e Downcasting

* Upcasting: Conversão da subclasse para a superclasse;
  + A conversão é feita implicitamente basta uma superclasse receber um objeto da subclasse (SuperClasse objSuperclasse = objSubClasse)
* Downcasting: Conversão da superclasse para a subclasse;
  + A conversão não pode ser feita implicitamente (SubClasse ).
  + É uma operação insegura;

Texto

Descrição gerada automaticamente

# Sobreposição (ou sobrescrita)

* Implementação de um método da superclasse na subclasse;
* Para um método não abstrato para permitir a sobrescrita, é necessário o uso do prefixo “virtual”;
* Para sobrescrever um método é necessário o uso do prefixo “override”;

Texto

Descrição gerada automaticamente com confiança média

* Palavra “base”: Executa o método da forma que foi implementado na superclasse (reaproveitamento de código) e ainda é permitido acrescentar alguma tratativa a mais.

Texto

Descrição gerada automaticamente

# Classe e Métodos Selados

* Palavra: sealed;
* Evita que a classe seja herdada;
* Na declaração da classe: sealed class NameClass;

Texto

Descrição gerada automaticamente com confiança média

* Evita que um método sobreposto (override) possa ser sobrescrito novamente;

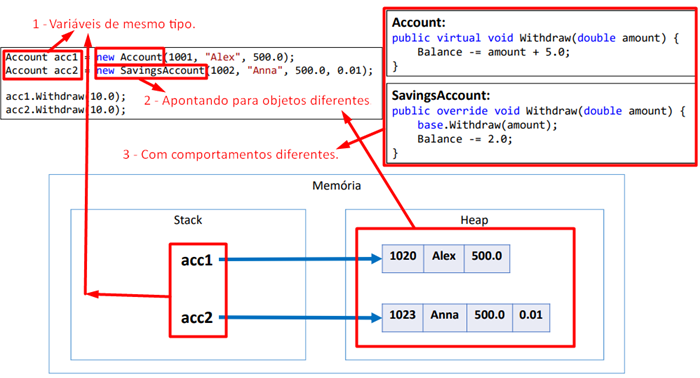
Interface gráfica do usuário, Texto

Descrição gerada automaticamente com confiança média

* Justificativa:
  + Segurança dependendo da regra de negócio;
  + É uma boa prática de programação, selar métodos sobrepostos;
  + Performance: classe selada são mais performáticas;

# Polimorfismo

* Em Programação Orientada a Objetos, polimorfismo é recurso que permite que variáveis de um mesmo tipo mais genérico (1) possam apontar para objetos de tipos específicos diferentes (2; upcasting), tendo assim comportamentos diferentes conforme cada tipo específico (3).



# Classes Abstratas

* Classes que não podem ser instanciadas;
* Uso do termo abstract na declaração da classe;
* Vantagens: Reuso; Polimorfismo; é possível criar uma lista do tipo das classes abstratas mas com objetos das subclasses (exemplo abaixo).

Texto

Descrição gerada automaticamente

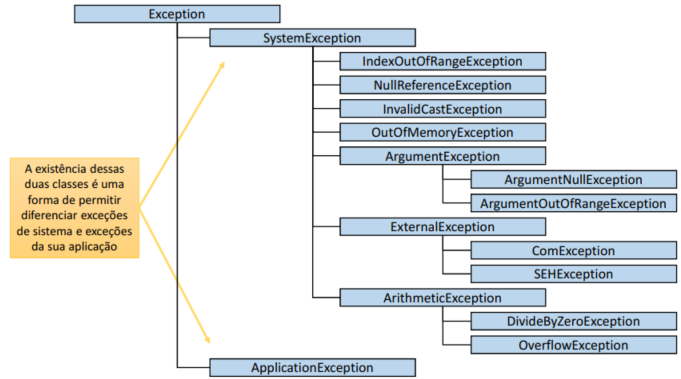
# Métodos Abstratos

* Métodos que não possuem implementação, muita das vezes pois a classe é muito genérica para uma implementação;
* Se a classe possuir pelo menos um método abstrato, então ela é uma classe abstrata;

Tratamento de Exceções

# Introdução

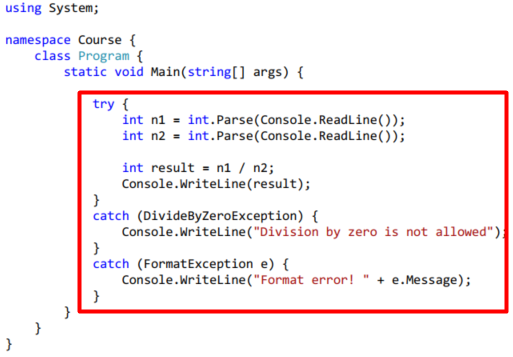
* Condição de erro ou tratamento inesperado em um programa em execução;
* No .NET: Objeto da classe Exception;
* Exceção, quando ocorre ela é lançada na pilha de chamadas e é capturada (tratada) ou até que o programa seja encerrado;



* SystemException ocorre devido a erro das classes do .NET;
* ApplicationException são erros da nossa própria aplicação;
* Serve para o tratamento de erros de forma flexível utilizando boas práticas;
* Vantagens:
  + Delega a lógica do erro para a própria classe;
  + Trata de forma hierárquica exceções de tipos diferentes;
  + É possível passar dados para dentro das exceções;

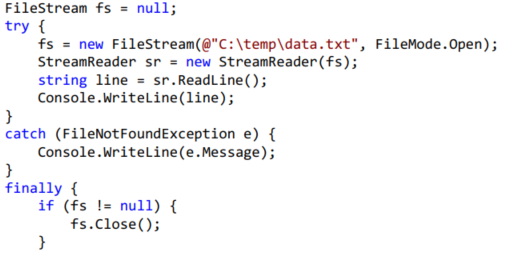
# Try – Catch

* Try: Código com possível erro de exceção;
* Catch: Bloco a ser executado caso a exceção ocorra;
* Sintaxe:

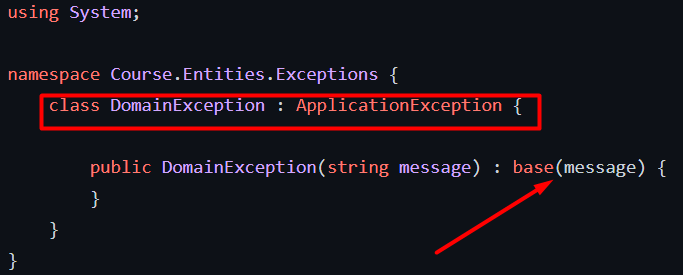
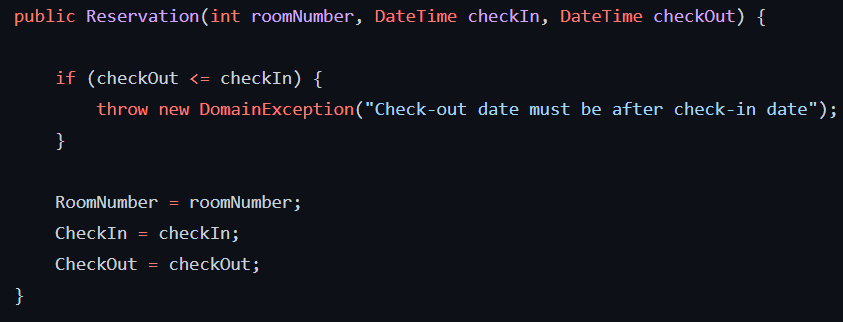


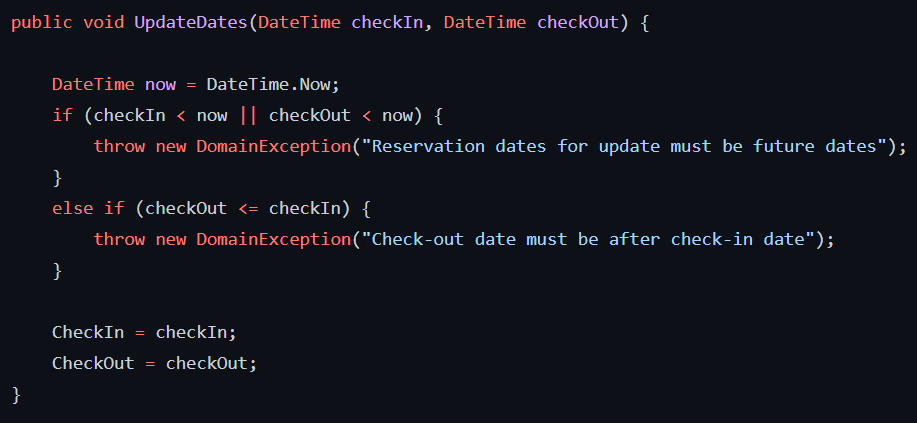
# Bloco Finally

* É um bloco que contém código a ser executado independentemente de ter ocorrido ou não uma exceção.
* Exemplo clássico: fechar um arquivo ou conexão de banco de dados ao final do processamento.



# Exceções Personalizadas (Tratamento dentro da Classe)

* Nova pasta Exceptions;
* Cria-se uma nova classe estendendo ApplicationException e cria sua exceção;
* Utiliza-se como padrão o nome NomeDaExceção + Exceptions;
* Sintase da classe:
* Para lançar a exceção, cria-se a regra utilizando um if e lança a exceção através da palavra reservada “throw”:
  + No construtor:
  + No método:



* + No programa principal, captura a exceção com o try/catch.
* Cláusula throw: lança a exceção / "corta" o método;
* O modelo de tratamento de exceções permite que erros sejam tratados de forma consistente e flexível, usando boas práticas;
* Vantagens:
  + Lógica delegada a classe de domínio;
  + Construtores podem ter exceções;
  + Código mais simples. Não há aninhamento de condicionais: a qualquer momento que uma exceção for disparada, a execução é interrompida e cai no bloco catch correspondente.
  + É possível capturar inclusive outras exceções de sistema

14 - Interfaces

# Interfaces

* Tipo que define o conjunto de operações que uma classe tem que implementar, estabelecendo um tipo de contrato;
* Suas vantagens são garantir o baixo acoplamento e flexibilidade (inicia-se com I em C# por padronização):

Texto

Descrição gerada automaticamente

# Inversão de Controle, Injeção de Dependência

* No exemplo dado, criou-se uma classe para cálculo de um aluguel (RentalService) fortemente acoplada a uma classe que calcula a taxa de imposto a ser pago (BrazilTaxService);
* No entanto, dessa forma, gera-se um forte acoplamento. Uma vez que, se for calcular a taxa de imposto para outro país, seria necessário alterar a classe RentalService (lembrando que classes devem ser fechadas para alteração e abertas para extensão).

Diagrama

Descrição gerada automaticamente

* A forma correta seria acrescentar uma Interface (ITaxService) que seria implementada pelos services de cálculo da taxa;
* Dessa forma, quando a interface é declarada como atributo de RentalService, a classe fica preparada para receber qualquer classe que implementa a interface ITaxService.
* O que gera acoplamento fraco. Consequentemente, se a classe TaxService muda, não é necessário alterar a classe RentalService. Classe fechada para alteração e aberta para extensão.

Diagrama

Descrição gerada automaticamente

* A injeção de dependência foi feita, por tanto, utilizando o construtor da classe RentalService ao ser instanciada pelo programa principal;
* Injeção de dependência é quando se instancia um objeto informando qual é o outro objeto que o ele depende (ver figura abaixo);
* Inversão de controle: O controle de quem informa de quem será a dependência do serviço (classe), não é mais do próprio serviço e sim de outro lugar no sistema (nessa caso, foi feito pelo programa principal utilizando o construtor da classe).

Interface gráfica do usuário, Texto, Aplicativo, Email

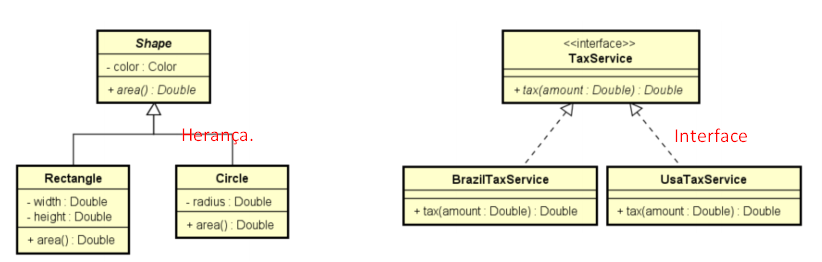
Descrição gerada automaticamente

Texto

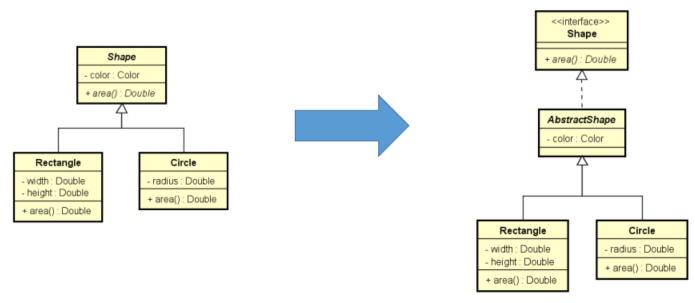
Descrição gerada automaticamente

# Herança vs Cumprir Contrato (Interface)

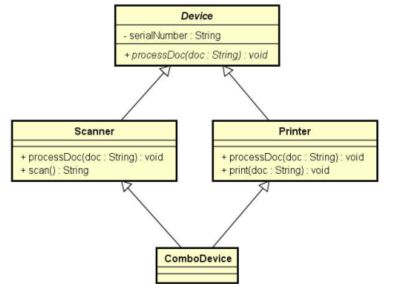
* Ambos estabelecem:
  + Relação é-um;
  + Generalização e especialização;
  + Polimorfismo;
* Diferenças:
  + Herança: Reuso;
  + Interface: Contrato a ser cumprido (obrigatoriedade de implementar o método)



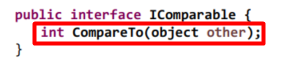
* É possível faxer a seguinte configuração:

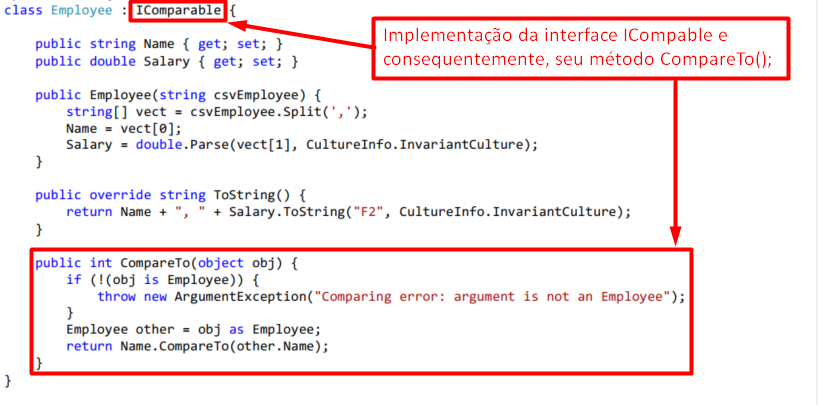


# Problema do Diamante

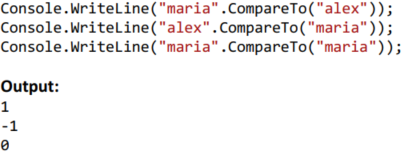
* A herança múltipla pode gerar o problema do diamante: uma ambiguidade causada pela existência do mesmo método em mais de uma superclasse (no caso seria o método processDoc());
* No C# não permite essa situação;

# Interface IComperable

* Interface padrão para se comparar um objeto com outro;
* Logo se quisermos que nossa classe compare seus objetos, devemos implementar a interface IComparable.
* Se colocarmos esse objetos em um list, podemos utilizar list.sort() (mas apenas após implementar a interface e seu método);



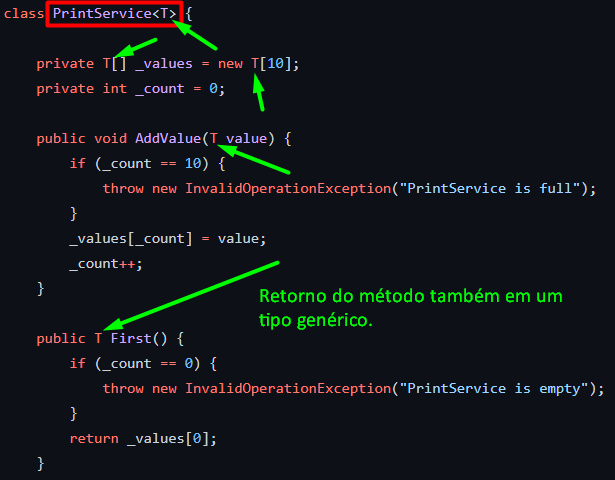
* Os retornos de CompareTo(object obj) podem ser:
  + Menor que zero – Se obj for menor que o objeto atual (this);
  + Igual a zero – Se obj for igual ao objeto atual (this);
  + Maior que zero – Se obj for maior que o objeto atual (this);



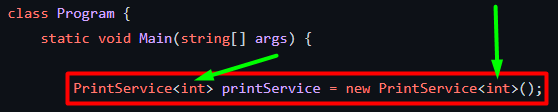
15 – Generics, Set, Dictionary

# Introdução ao Generics

* Permite que classes, interfaces e métodos sejam parametrizados por tipo. Muito comum em coleções. Garante:
  + Reuso;
  + Type Safety (IDE não detecta erros de tipos de variáveis que são detectados apenas exceptions durante a execução);
  + Performance;
* Exemplo de uso: Classe que preenche um vetor de inteiros para depois imprimi-los na tela. Entretanto, se eu quiser reaproveitar essa classe para coletar outros tipos de dados e imprimi-los;
* Sintaxe: Na declaração da classe adiciona-se “<T>” (tipo genérico) ao lado de seu nome. Logo, a classe fica disponível para ser utilizada com qualquer tipo:

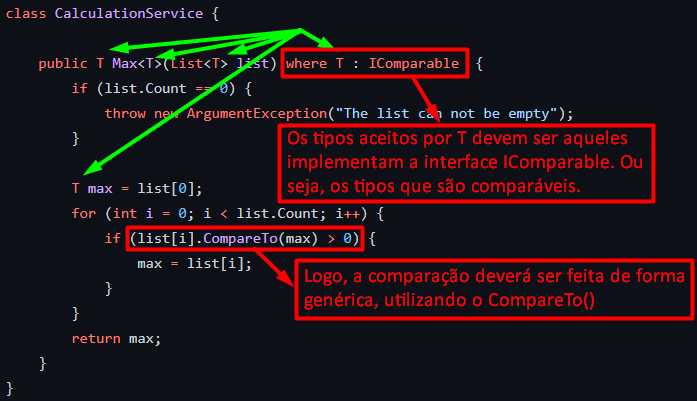


* No momento em que a classe for instanciada, define-se o tipo do objeto:



# Restrições ao Generics

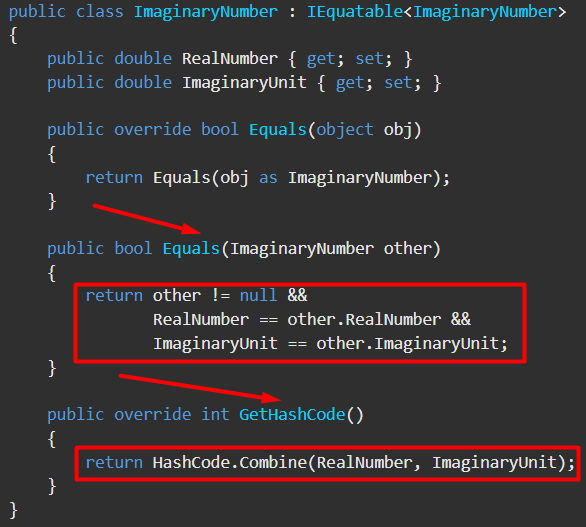
* Restringir quais tipos a classe genérica poderá receber:
* Exemplo: Serviço que retorna o maior item de uma lista. Logo, os tipos aceitos na classe devem ser restringidos a tipos que é possível realizar a comparação do tipo >, <, >=, <=.
* Ao dizer que a entidade é genérica, deve-se definir qual interface que a classe deve implementar para que seja aceito de forma genérica. Ver exemplo abaixo:



* No caso acima, se fosse uma nova classe estendendo IComparable, seria obrigatório (logicamente) implementar o método CompareTo.
* As possíveis restrições são (<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters>):
  + where T: struct
  + where T: class
  + where T: unmanaged
  + where T: new()
  + where T: <base type name>
  + where T: U

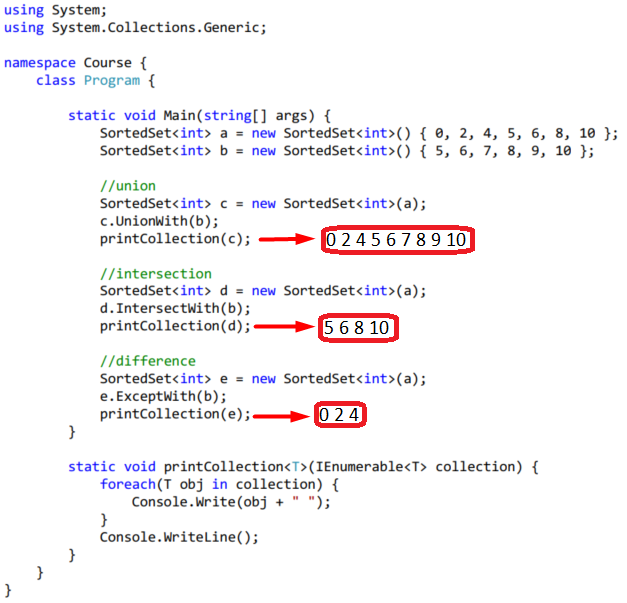
# GetHashCode e Equals

* Utilizados para comparar dois objetos. São métodos virtuais da classe object, logo podemos sobrescrevê-los;
* Tipos primitivos já possuem esses métodos implementados;
* O GetHashCode() é mais eficiente. Entretanto tem uma pequena probabilidade de retornar um falso-verdadeiro (pode haver colisão);
* A ideia é utilizar o GetHashCode() em uma busca dentro de um vetor com inúmeros dados. Retornando verdadeiro, realizar o teste com o Equals() para ter 100% de certeza.
* Ao sobrescrever esses métodos, definimos qual (ou quais) parâmetro(s) determina se dois objetos são iguais:
  + Numa classe pessoa, seria Id? CPF?



# Conjuntos HashSet<T> e SortedSet<T>

* Representa um conjunto de elementos:
  + Não admite repetições;
  + Elementos não possuem posição;
  + Inserção e remoção são rápidos;
  + Oferece operações de conjunto
* HashSet:
  + Armazenamento em tabela hash (cada elemento possui um código relacionado a ele);
  + Extremamente rápido: inserção, remoção e busca em apenas um passo;
  + A ordem dos elementos não é garantida;
* SortedSet:
  + Armazenamento em árvore;
  + Rápido: inserção, remoção e busca numa ordem logarítmica (log(n));
  + Os elementos são armazenados ordenadamente conforme a implementação da interface IComparer<T>;
  + Devolve sempre os conjuntos ordenados;
* Alguns métodos:
  + Add – Adicionar elemento no conjunto;
  + Clear – Limpar conjunto;
  + Contains – Contem no conjunto?
  + UnionWith(other) - operação união: adiciona no conjunto os elementos do outro conjunto, sem repetição;
  + IntersectWith(other) - operação interseção: remove do conjunto os elementos não contidos em other;
  + ExceptWith(other) - operação diferença: remove do conjunto os elementos contidos em other
  + Remove(T)
  + RemoveWhere(predicate)
* Para percorrer os elementos é necessário o “foreach”;

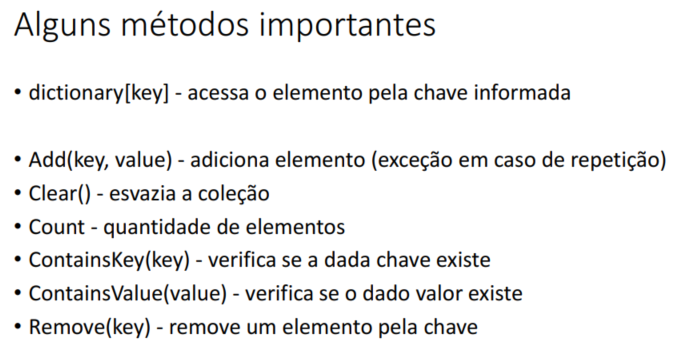


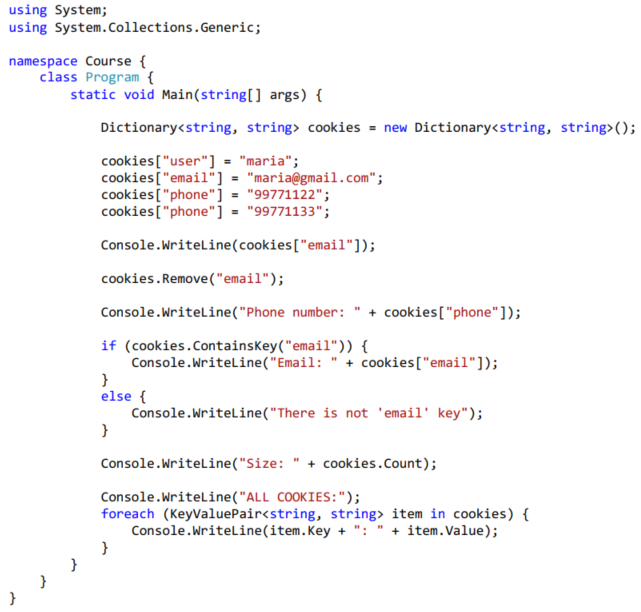
# Como as coleções Hash testam igualdade

* Se GetHashCode e Equals estiverem implementados:
  + Primeiro GetHashCode. Se der igual, usa Equals para confirmar;
* Se GetHashCode e Equals NÃO estiverem implementados:
  + Tipos referência (classe produto, cliente e etc): compara as referências dos objetos
  + Tipos valor: comparar os valores dos atributos

# Dictionary e SortedDictionary (Map)

* Coleção de pares chave / Valor
  + Pra cada elemento temos uma chave e seu valor;
  + Não admite repetições do objeto chave;
  + Os elementos são indexados pelo objeto chave (não possuem posição)
  + Acesso, inserção e remoção de elementos são rápidos
* Uso comum: cookies, local storage, qualquer modelo chave-valor;
* Dictionary
  + Armazenamento em tabela hash
  + Extremamente rápido: inserção, remoção e busca O(1)
  + A ordem dos elementos não é garantida
* SortedDictionary
  + Armazenamento em árvore
  + Rápido: inserção, remoção e busca O(log(n))
  + Os elementos são armazenados ordenadamente conforme implementação IComparer<T>

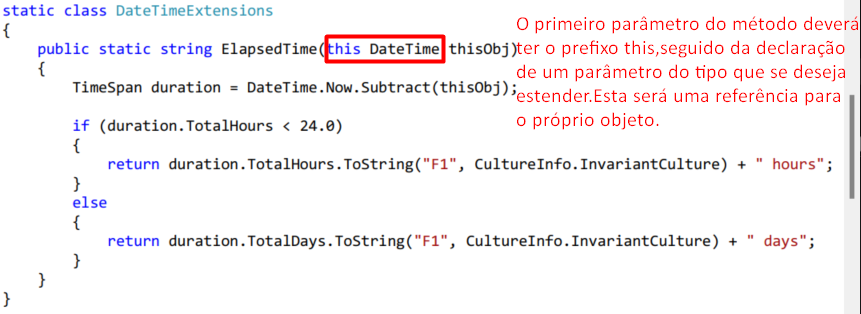


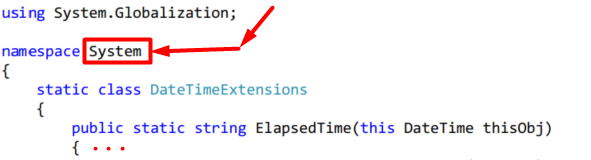


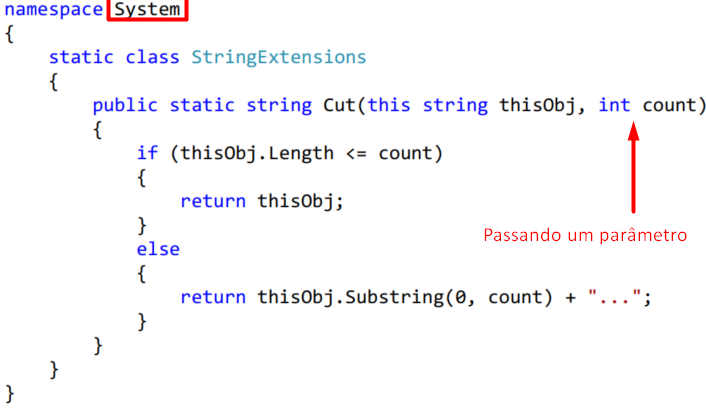
Extension Methods

# Introdução

* Métodos que estendem a funcionalidade desse tipo sem alterar o código e sem herdar desse tipo;
* Pode chamar o método a partir de um objeto já existente;
* Para implementá-lo deve-se criar uma classe estática com um método estático. Exemplo:



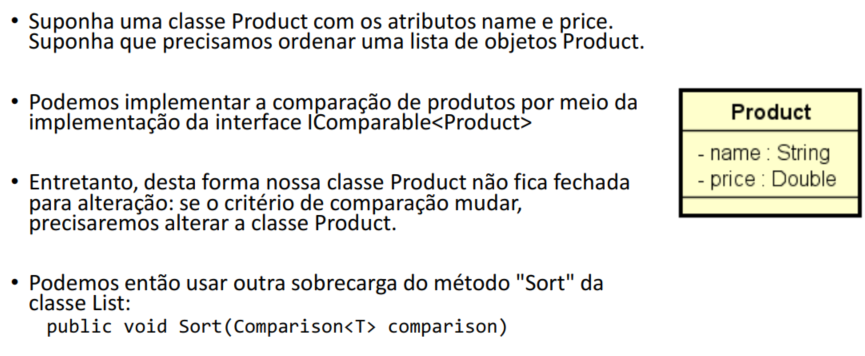
* Existe a opção de colocar a nova classe no mesmo namespace que foi estendido, logo não é necessário um novo using na classe que for utilizar o extension method
* Outro exemplo:



Expressões lambda, delegates, LINQ

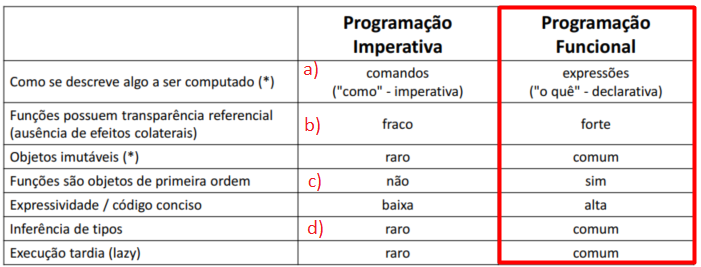
# Comparison <T>

* Problema:

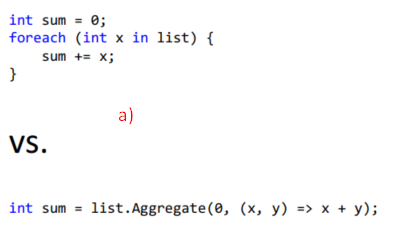


# Programação Funcional e Cálculo Lambda

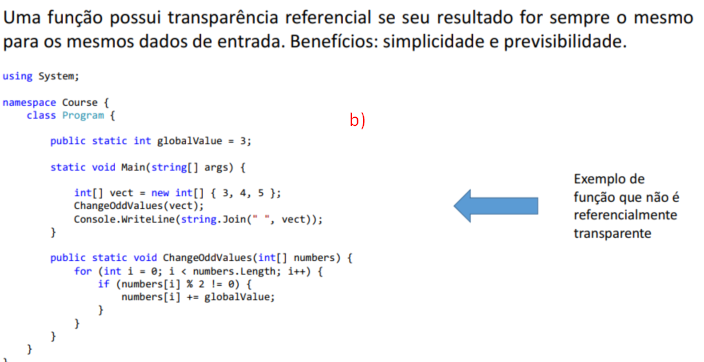
* Programação Funcional: Baseado no formalismo matemático Cálculo Lambda (Church 1930);
* Uma função Lambda corresponde a uma função anônima de primeira classe;
  + list.Sort((p1, p2) => p1.Name.ToUpper().CompareTo(p2.Name.ToUpper()));

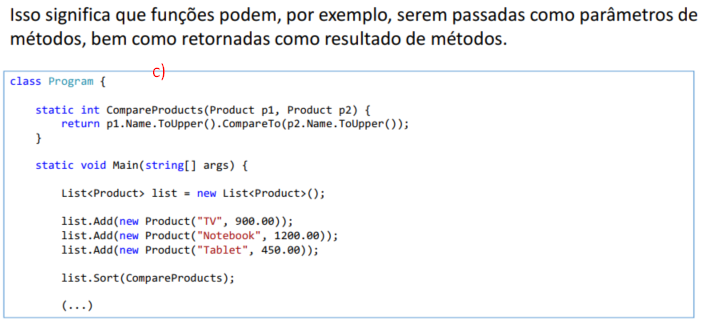


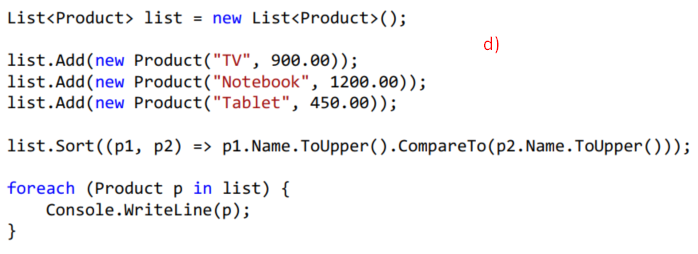
1. Expressividade: “como” vs “o que”



1. Transparência referencial:



1. Funções como objetos de primeira ordem
2. Inferência de tipos:

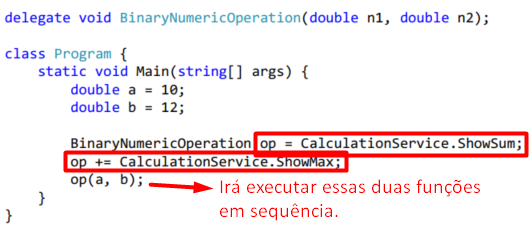


# Introdução ao Delegate

* É um tipo referência para um ou mais métodos;
* Usos comuns:
  + Comunicação entre objetos de forma flexivel;
  + Parametrização de operações por métodos (programação funcional) – Função sendo parâmetro de outra função;
* Delegates
  + Action
  + Func
  + Predicate
* Sintaxe:

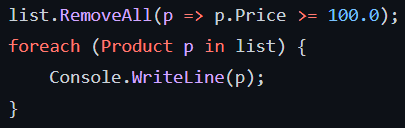
# Multicast Delegates

* Guardam referência de mais de um método;
* Utiliza-se += se quiser atribuir mais de um método ao Delegate
* A chamada (Invoke) executa os métodos na ordem em que foram adicionados;
* Faz sentido para métodos void.

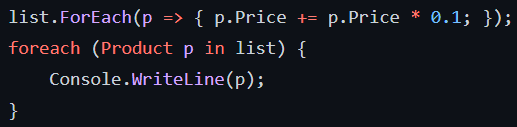


# Delegate Predicate

* Representa um método que recebe um objeto do tipo T e retorna um valor booleano;
* Por padrão deve ser uma função que recebe um objeto e devolve um booleano;
* public delegate bool Predicate<in T>(T obj);

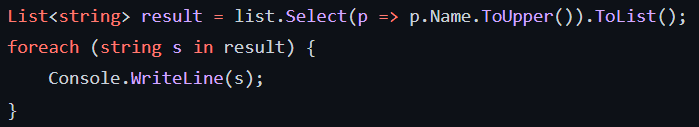


# Action

* Representa um método void que recebe zero ou mais argumentos (total de 16 sobrecargas);
  + public delegate void Action();
  + public delegate void Action<in T>(T obj);
  + public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
  + public delegate void Action<in T1, in T2, in T3>(T1 arg1, T2 arg2, T3 arg3);
  + (…)
* Note que em expressões lambda, quando o retorno é void, acrescentar “arg => {//função}”;

# Func

* Representa um método que recebe zero ou mais argumentos, e retorna um valor(total de 16 sobrecargas);
  + public delegate TResult Func<out TResult>();
  + public delegate TResult Func<in T, out TResult>(T obj);
  + public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
  + public delegate TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3);
  + (…)

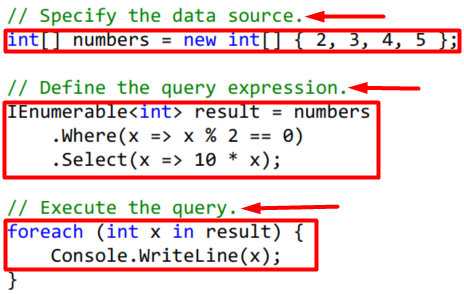


# Introdução ao LINQ (Language Integrated Query)

* É um conjunto de tecnologias baseadas na integração de funcionalidades de consulta diretamente na linguagem C#;
* Possui diversas operações de consulta, cujos parâmetros tipicamente são expressões lambda ou expressões de sintaxe similar à SQL;
* <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/>
* Para trabalhar com o Linq:
  + Criar uma fonte de dados (data source);
  + Definir a Query, não é executada;
  + Execução da consulta (foreach ou consulta terminal);



* Exemplo:



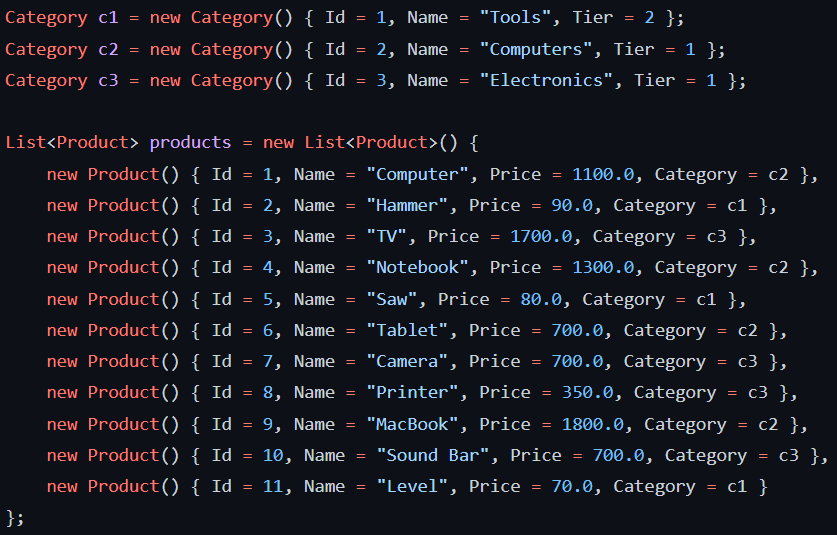
* Principais Operações:



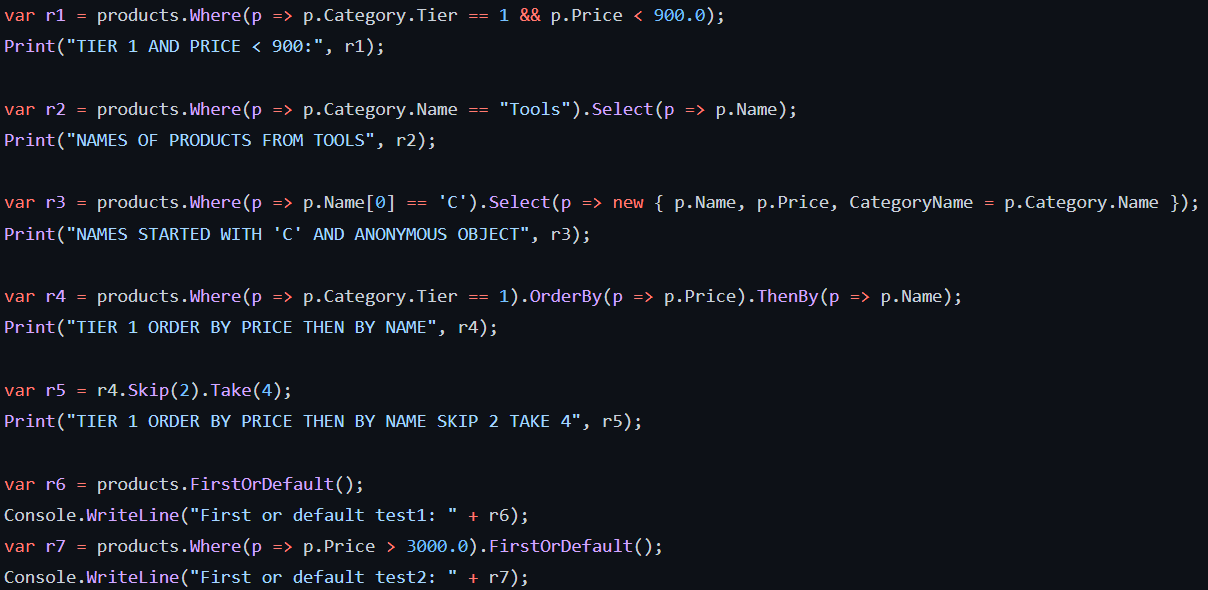
* https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b/view/SamplePack/1?sortBy=Popularity
* https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b/view/SamplePack/2?sortBy=Popularity
* https://odetocode.com/articles/739.aspx
* Exemplo com base na seguinte associação:

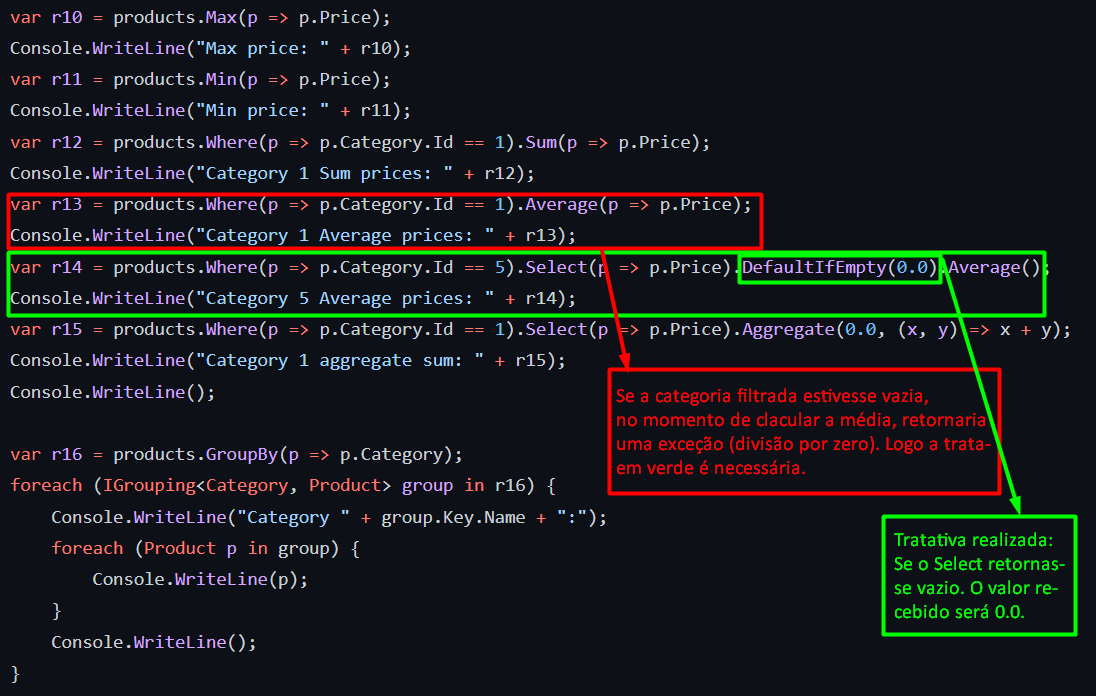


* + Fonte de Dados:



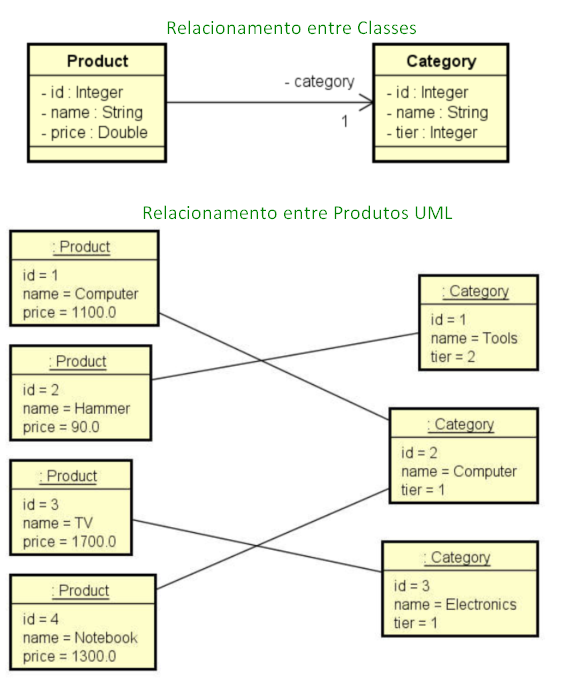
* + Algumas operações utilizando o LINQ:

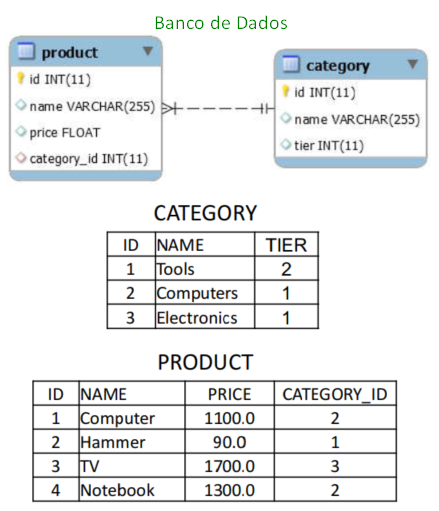




# Álgebra Relacional e SQL

* Comparação entre estruturas de dados em classes e banco de dados;

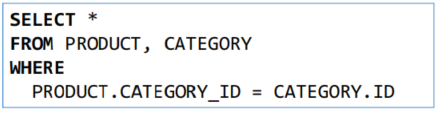


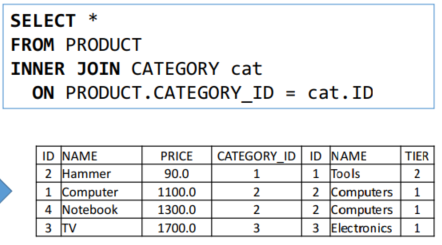


* Operação “Produto Cartesiano”

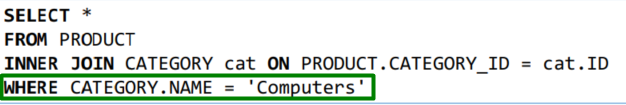


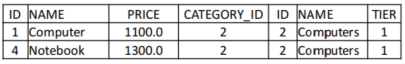
* Operação de Junção: Relacionamento definido pelas chaves estrangeiras (duas opções):

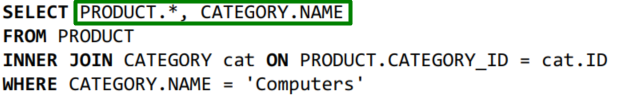


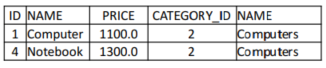


* Restrição: Filtro com determinada regra;

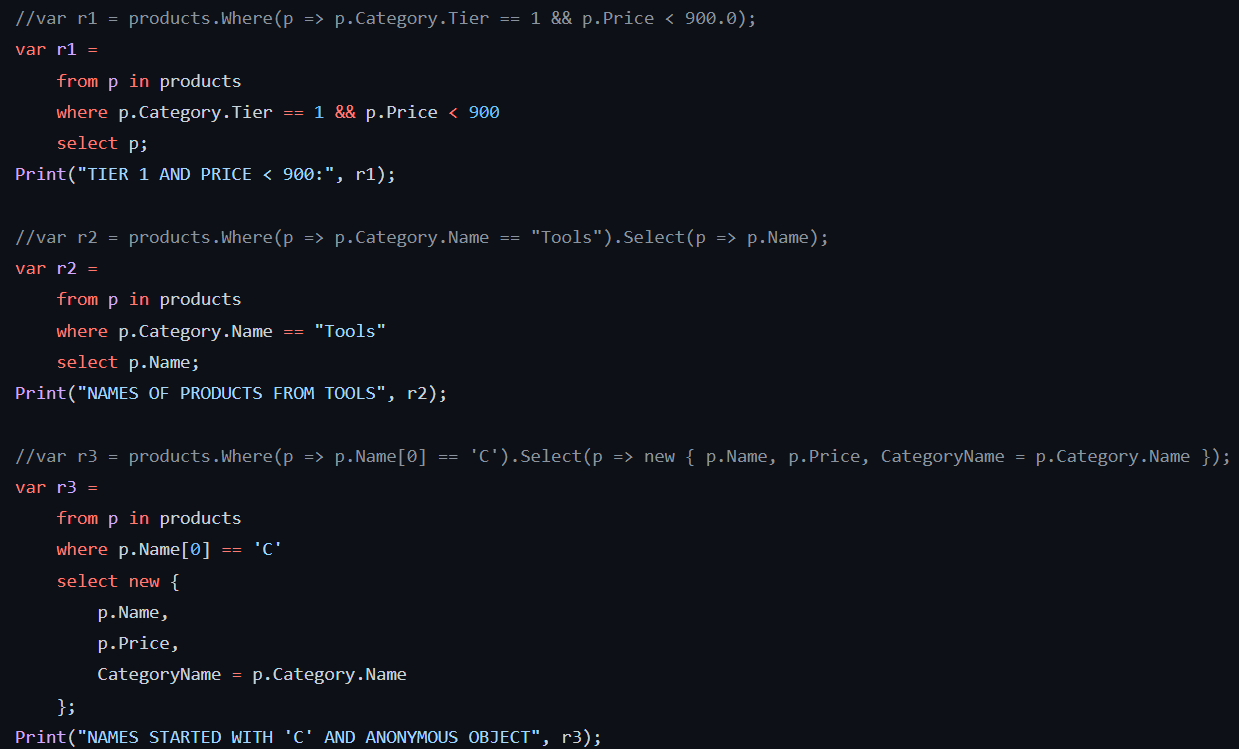


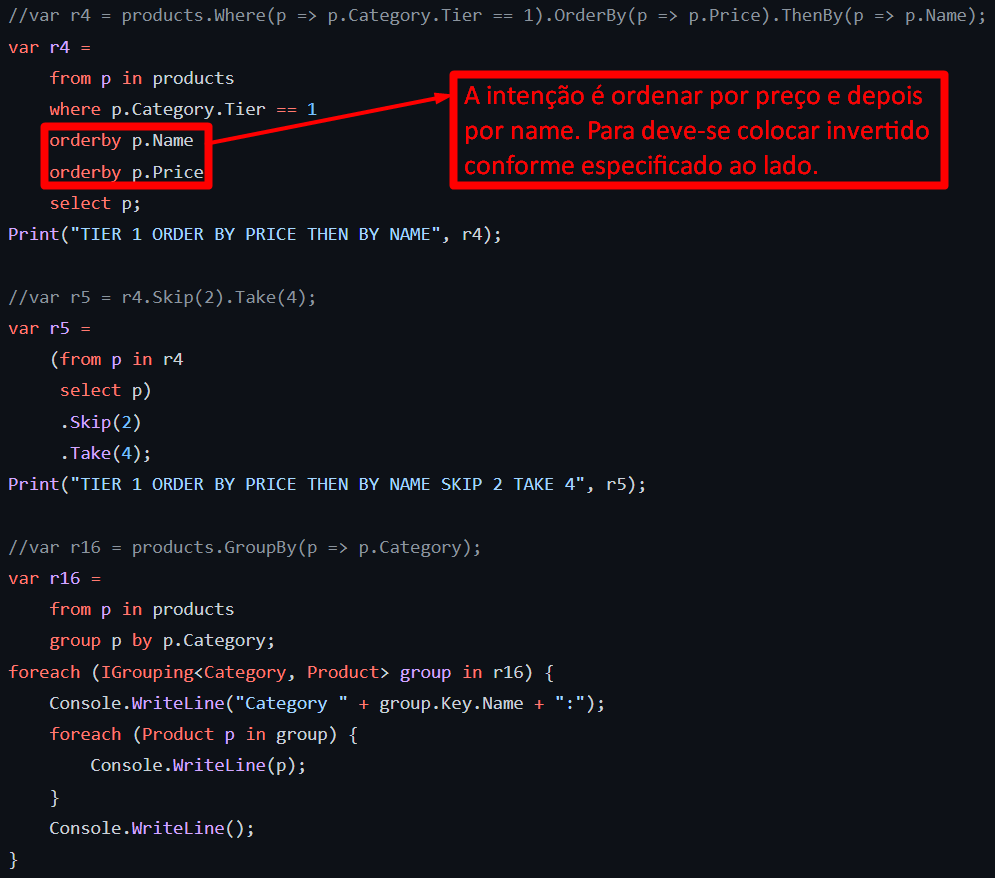
* Projeção: Seleção das colunas que devem ser exibidas:





# Link com Sintaxe Similar ao SQL





Projeto Sistema Web com ASP.NET Core MVC e Entity Framework

# Introdução ASP.NET Core MVC

* Framwork para desenvolvimento de aplicações WEB;
* Roda tanto no .NET Framework quento no .NET Core;
* Estrutura definida de:
  + Controllers;
  + Views;
  + Models – View Models ;
* https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-6.0

Aplicações WEB

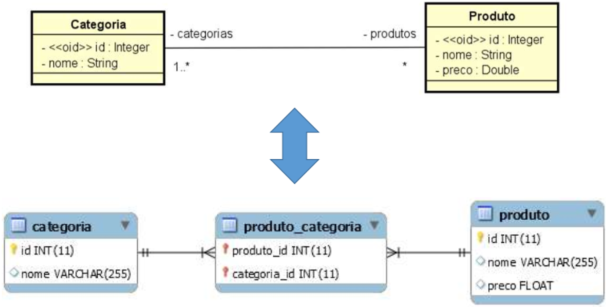
* Uma aplicação web pode funcionar de dois modos, são eles:
  + Web Services:
    - Dividido em Backend e Frontend (HTML, CSS, JS e frameworks) ;
    - O frontend é responsável por desenhar as páginas e se comunica com backend através de API Rest (HTTP, AJAX);
    - O backend é composto por Model e Controllers que se comunicam através de objetos;
    - Os controladores recebem informações recebem requisições do frontend, converte em objetos e devolvem o resultado em formato de texto (JSON, normalmente);
  + Template Engine:
    - O Backend possui mais responsabilidades. Inclusive constroi a tela e envia ao frontend no formato HTML;
    - As requisiçoes são feitas ao controlador que conversa com o model e devolve a informação ao sistema de templates (que monta a página de resposta);

# MVC – Model, View, Controller

* O MVC é um padrão que consiste em dividir a aplicação em 3 partes:
  + Model:
    - Mantem estrutura dos dados e suas transformações (Domain Model). Podemos defini-lo como “o sistema”.
    - Possui a inteligência do negócio (regras de negócio);
    - Composto por Entities, Services e, dependendo da arquitetura, repositories (acessam o banco de dados);
  + Controllers:
    - Tem a função de receber e tratar as interações do usuário com o sistema;
    - Faz o meio de campo entre o sistema e as Views
  + View:
    - Definir a estrutura e comportamentos das telas
* Arquitetura em geral:

# Entity Framework

* Por muitos anos, uma grande dificuldade de se criar sistemas orientados a objetos foi a comunicação com banco de dados relacional;
* Solução: Mapeamento Objeto - Relacional (ORM)
  + Permite programar em nível de objetos e comunicar de forma transparente com um banco de dados relacional;
  + O entity framework é um ORM dos mais conhecidos;
* Providers são as implementações para os bancos de dados específicos.
* <https://docs.microsoft.com/en-us/ef/>

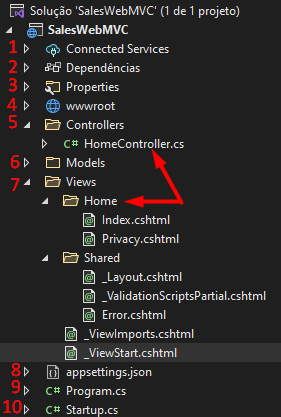


* Principais classes:
  + DbContext: Objeto que encapsula uma seção com o banco de dados para um determinado modelo de dados específico (representado por DbSet’s) – conexão e seção no banco de dados;
    - Consulta e salva entidades no banco de dados;
    - Define as entidades que farão parte do modelo de dados do sistema;
    - Define diversas configurações;
    - Combinação dos padrões Unity of Work e Repository
      * Unity of Work: Mantém uma lista de objetos afetados por uma transação e coordena a escrita de mudanças e trata possíveis problemas de concorrência;
      * Repository: Define um objeto capaz de realizar operações de acesso a dados (consulta, salvar, atualizar e deletar) para uma entidade;
  + DbSet<Tentity>: Representa uma coleção de entidades de um dado tipo em um contexto. Tipicamente, corresponde a uma tabela no banco de dados;
* Processo de execução de operações:



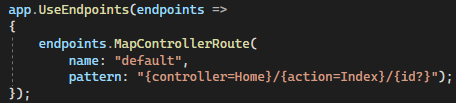
# Estrutura do Projeto

* 1 – Serviços em que o projeto pode estar conctados
* 2 – Dependências (Pacotes e imports);
* 3 – Propiedades do Projetos;
* 4 – wwwroot – Recursos do frontend (css, js e etc)
* 5 – Controladores
* 6 – Modelos
* 7 – Views (Telas da aplicação).
  + Subpastas correspondem respectivamente a cada tela:
    - Já vem com uma pasta home que compreende os arquivos responsáveis por gerar essa tela inicial da aplicação. Esta view está associada a um controlador HomeController (dentro da pasta controllers) que controla as ações relacionadas a aplicação básica;
    - Cada página são arquivos com o formato cshtml (aceita tanto html quanto C#)
  + Ainda possui uma pasta Shared que são páginas compartilhadas por toda a aplicação (layout básico);
  + Arquivo \_ViewStart, define a tela que dá início a aplicação;
* 8 – appsetting.json – configuração de recursos externos (acessos a banco, por exemplo);
* 9 – Program – Entrepoint da aplicação que aciona a classe Startup.cs;
* 10 – Startup.cs; - Configuração da aplicação;

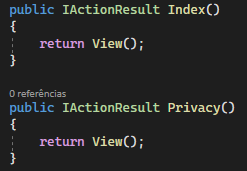


# Primeiro Controlador e Teste da Página Razor

* Padrão de rotas: Controlador → Ação → Id (opcional).
  + Isso pode ser verificado no arquivo Setup.cs (últimas linhas);
  + Caso seja digitado apenas o endereço no navegador (“endereço”:“porta”) esse arquivo determina que será chamada a ação Index()



* Cada método do controlador, mapeia uma ação que retorna um IactionResult (resultado de uma ação). Logo, digitando “endereço”:“porta”/Home/Index, executa o método abaixo;



* Quando é executada um desses métodos o framework constrói a página (pasta View) que possui o mesmo nome do método executado acima;
* No método é possível passar um dicionário ViewData com uma chave e uma valor para mesma. Esse valor capturado pela view e apresentado na tela de acordo com as tags html em que foram passadas:
* Obs: Utilizando o @{ … } é permitido utilizar código C# dentro da estrutura.
* Natural Template: Quando digitamos um determinado caminho, estamos chamando diretamente o controlador, que por sua vez encaminhará requisição para a View.
* A interface IActionResult generaliza as seguintes especificações. O que determina qual tipo (Classe) será retornado é o MethodBuilder.

