**=========== Programação Orientada a Objetos ===========**

-> Paradigma onde tenta aproximar a programação do mundo real.

4 pilares:

1° Encapsulamento: forma de esconder a implementação da classe;

2° Herança: maneira de herdar atributos e comportamentos de uma classe;

3° Polimorfismo: maneira que um objeto muda seu comportamento de diferentes formas;

4° Abstração: pensar na forma de aproximar do mundo real

Orientação a objetos visa a melhor manutenção de um sistema e o rápido desenvolvimento de maneira rápida.

**=========== Classe, objeto, atributos e métodos ===========**

Classe: seria o molde do objeto pensado;

Objeto: seria a instância da classe, ou seja, a criação;

Atributos: características que fazem parte da classe;

Métodos: Comportamentos;

**=========== Constructor ===========**

Serve para inicializar objetos e falar o que é necessário ser inicializado;

Uma classe pode possuir mais de 1 contrutor (conceito de sobrecarga)

A ideia disso é justamente para quando a classe for instanciada ela ter alguns atributos já definido;

**=========== Modificadores de acessos ===========**

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers

Serve para controlar a visibilidade da classe, seus métodos e atributos dentro do projeto;

public: não existe restrição / é visível para todos;

private: acesso limitado a classes, sempre quando uma classe é criada ela por default é private

internal: acesso limitado dentro do assembly (projetos externos não tem acesso)

protect: acesso limitado a membros de mesma classe e herança.

**=========== Propriedades e encapsulamento ===========**

Encapsulamento: Processo de esconder os membros de uma classe para um acesso exterior;

Get = retorna o valor de uma propriedade;

Set = atribuição de valor para uma propriedade;

**=========== Modelos de dominios ===========**

Modelo: seria a models, a representação do mundo real;

Modelo anêmico: são models "pobres", que não possui validação;

Modelos ricos: são models que consegue tratar situações dentro da própria classe;

**=========== Métodos e Hierarquia de projeto ===========**

Método = bloco de código, com instruções, possui modificadores e assinatura (tipo, nome, modificador de acesso e params).

Hierarquia de um projeto:

Namespace {

classe {

método {

}

}

}

Método Main: ponto de entrada a aplicação, chamada pela CLR (Common Language Runtime);

**=========== Sobrecarga / Overload ===========**

Método com o mesmo nome, mas assinatura diferente.

ex:

public int Somar (int num1, int num2) {}

public int Somar (int num1, int num2, int num3){}

**=========== Parâmetro opcional ===========**

\* Parâmetros que possuem um valor constante, mas podem sem modificados, caso não seja modificado é atribuido o valor padrão.

exemplos:

public int Somar (int num1, int x = 20, int z = 30) {}

**=========== Métodos Estáticos ===========**

É usado a palavra reservada static;

Basicamente, quando usado o static é indicado que o item não precisa ser instanciado para ser utilizado;

Ou seja, é possível utilizar o metódo, atributo, classe de qualquer lugar

Um exemplo é o metedo WriteLine ou ReadLine que não precisa de instância.

**=========== Herança ===========**

É um dos pilares da OO, basicamente possibilita a relação de classes, ou seja, uma classe (filha) herda os atributos, métodos, construtores, de uma classe pai.

Serve para reutilizar código.

É possível sobrescrever um método pai

para isso basta colocar a palavra virtual no método pai

public virtual void Falar(){}

E na classe filho para sobrescrever é preciso utilizar o override.

public override void Falar(){}

A palavra base() serve para reaproveitar algumas coisas da classe pai (!!!!!!estudar melhor!!!!!!)

**=========== As and Is ===========**

As: é usado para executar a conversão entre tipos de referência compatível ou tipo anulados.

Is: é para verificar se o objeto é compatível com o tipo especificado.

**=========== Diagrama UML ===========**

Serve para representar a classe e documentar.

|  |
| --- |
| NOME DA CLASSE |
| ATRIBUTOS |
| MÉTODOS |

+ -> public

- -> private

# -> protect

**Principais relacionamentos:**

Associação (vínculo entre classes);

Composição (tem um);

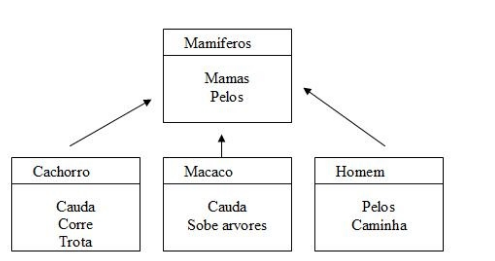
Agregação (tem um);

Herança (é um);

**=========== Herança ===========**

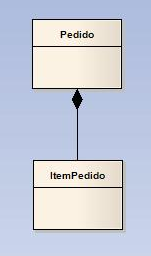
**Generalização:** classe pai (atributos em comum)

**Especialização:** classe filha (atributos distintos)



**=========== Composição ===========**

Ocorre quando o objeto de uma classe é composto de um ou mais objetos de outra classe.

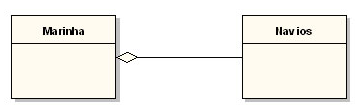


Possui uma relação de todo – parte, ou seja, o objeto só vai existir se o pai existir, por exemplo o ItemPedido só vai existir se o Pedido existir, caso contrário, não faz sentido.

**=========== Agregação ===========**

Mesmo conceito de composição

Todo – parte independente, ou seja, o filho vai existir mesmo sem o pai.



Por exemplo a marinha existe sem navios, e os navios não precisa da marinha para existir.

**=========== Acoplamento ===========**

Basicamente se refere ao quanto o seu sistema possui dependências.

**Forte Acoplamento:** é quando o sistema é muito rígido, está muito interligado, ou seja, isso é ruim, pois é difícil de dar manutenção em sistemas fortemente acoplado, pelo fato de tudo estar interligado, então para alterar uma coisa acaba influenciando em outras.

**Fraco Acoplamento:** é quando o sistema não possui dependências, sendo fácil entendimento e sendo fácil a manutenção, pois cada coisa está em seu lugar.

**Níveis de acoplamento:**

**Herança:** Forte acoplamento, gerando muitas dependências.

**Composição:** Acoplamento mais fraco que a herança.

**Agregação**: Acoplamento mais fraco de todos.

**\*PREFIRA USAR COMPOSIÇÃO, NO LUGAR DE HERANÇA.\***

**Problemas da herança:**

* Viola um dos pilares da O.O, no caso o encapsulamento, pois a herança expões a implementação da classe pai para os filhos.
* Viola a prática de desenvolvimento de um sistema de fraco acoplamento, levando em consideração que a herança gera um forte acoplamento entre as classes, sendo difícil a manutenção.

**Vantagens da composição:**

* Menor dependência.
* Mantém o princípio da SRP (responsabilidade única).
* Mantém o encapsulamento.

**Exemplo porque é melhor a composição do que a herança.**

Imagine o cenário de uma classe animal possuindo os métodos: Andar(), Comer() e Dormir().

Agora imagine a criação das classes filhas Homem e Gato.

Teoricamente podemos aplicar a herança nessas classes, pois o Gato e o Homem, fazem tudo isso.

Agora imagina a criação da classe Sardinha, temos um problema, pois a Sardinha é um animal, mas ela não anda, ela nada.

Então temos problema de abstração e esse problema impacta nas classes filhas.

Para resolvermos isso é necessário usar composição.

Ainda teremos a classe Animal, mas com os métodos Comer() e Dormir() e teremos as classes Gato, Homem e Sardinha.

Mas agora teremos outras 2 classes a ComportamentoAndar() e ComportamentoNadar().

Para implementarmos basta ir à classe por exemplo na classe Gato.

Class Gato{

Animal gato = new Animal();

ComportamentoAndar andar = new ComportamentoAndar();

}

Assim corrigimos o erro da abstração e deixamos o sistema fracamente acoplado (sem dependências).

**=========== Classe Abstrata ===========**

São classes que não podemos instanciar, para isso basta colocar a palavra reservada **abstract**.

Ficando

abstract class Test {}

Geralmente a classe abstrata serve de base, ou seja, para ser usada na herança.

Exemplo de uma classe abstrata:

abstract class Forma{

public void Desenhar(){

} // esse método pode ser sobrescrito.

Public abstract CalcularArea(); // esse método é obrigatório ser implementado pela classe filha.

}

**=========== Classe Sealed ===========**

Torna a classe selada, ou seja, ninguém pode herdar dela. É o oposto das classes abstratas.