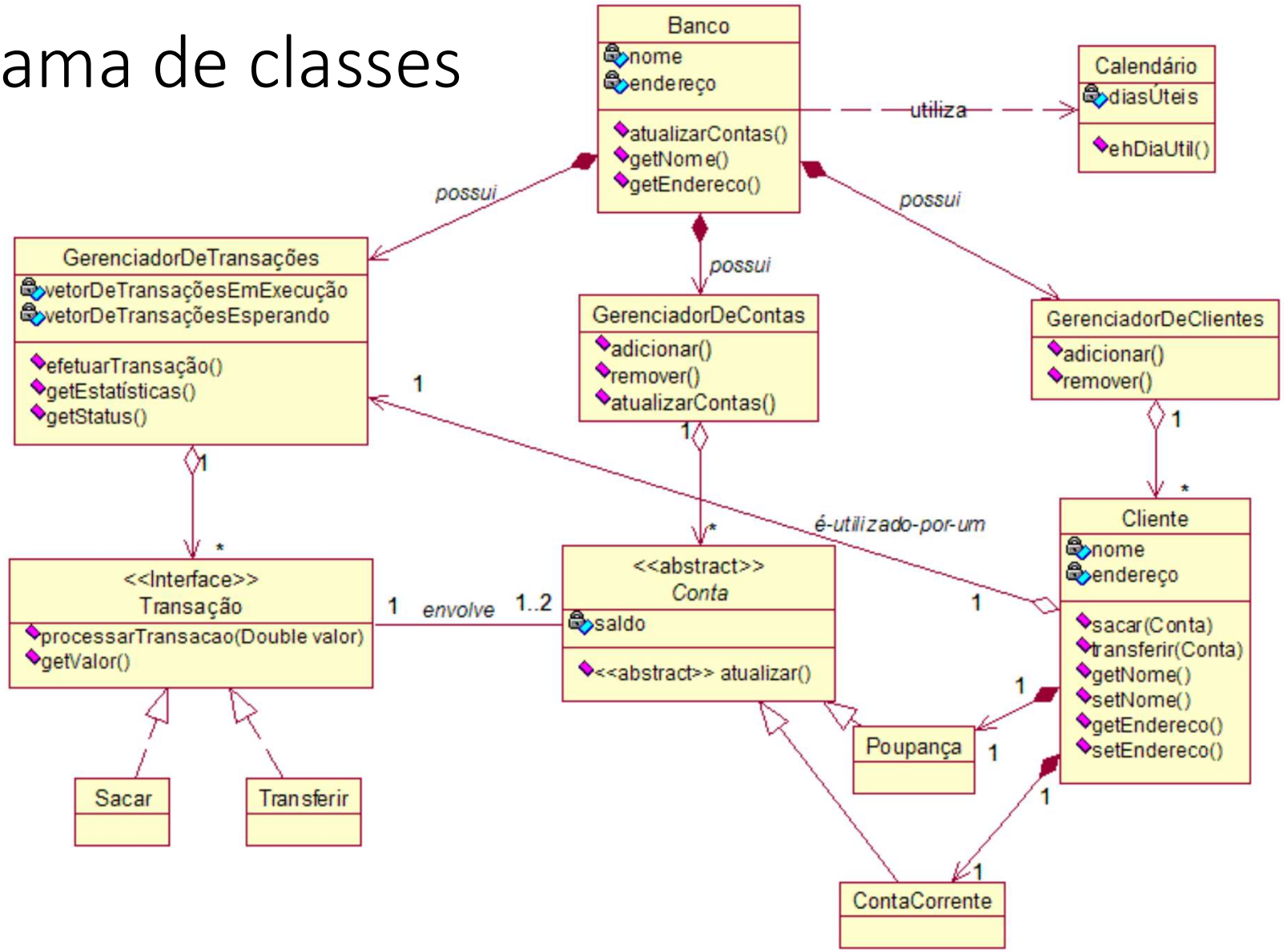
# Disciplina: POO - Programação Orientada a Objetos

# Relacionamento entre Classes

Profª. Drª. Giovana Angélica Ros Miola
giovana.miola@fatec.sp.gov.br


# Diagrama de classes

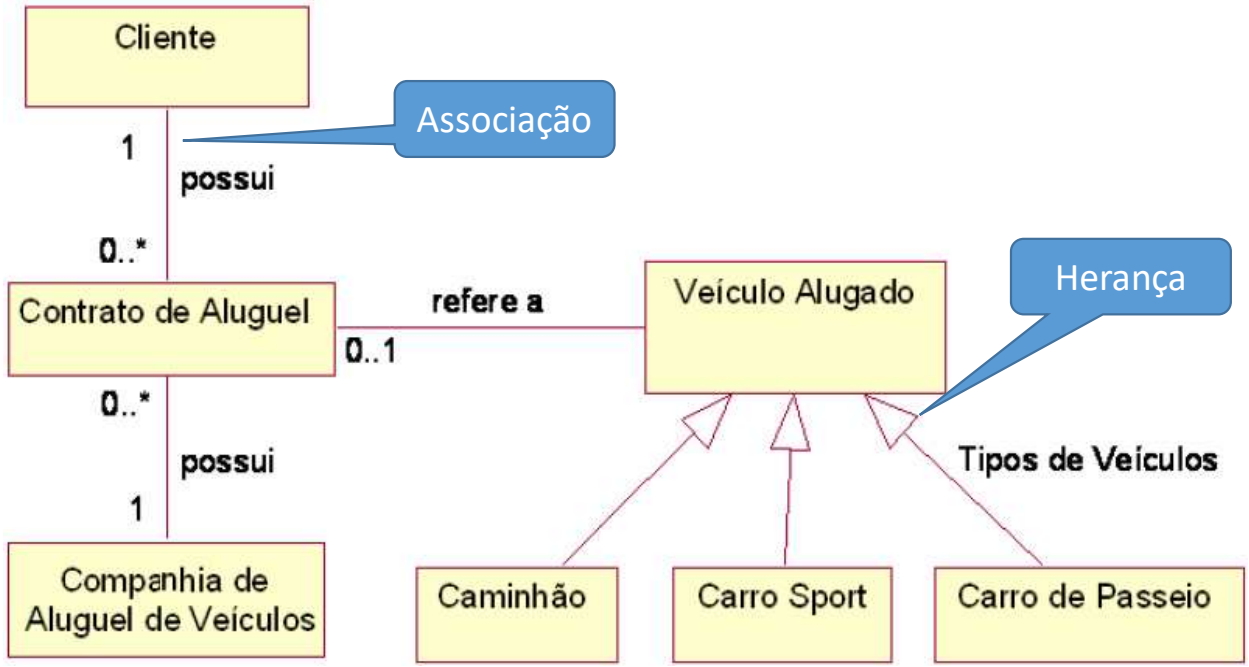


# Relacionamento entre classes

- Classes possuem relacionamentos entre elas (para comunicação)
  - Compartilham informações
  - Colaboram umas com as outras

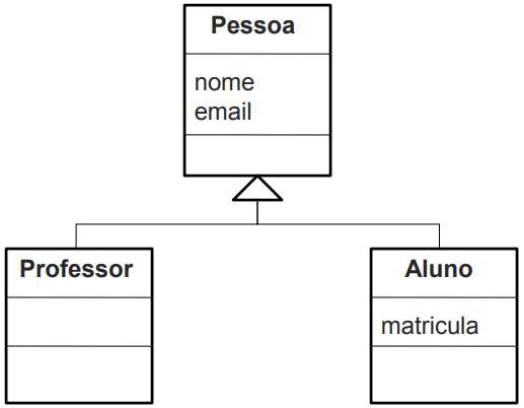
- Tipos de relacionamentos:
  - Agregação e Composição
  - Associação
  - Herança
    - Especialização/Generalização
  - Dependência

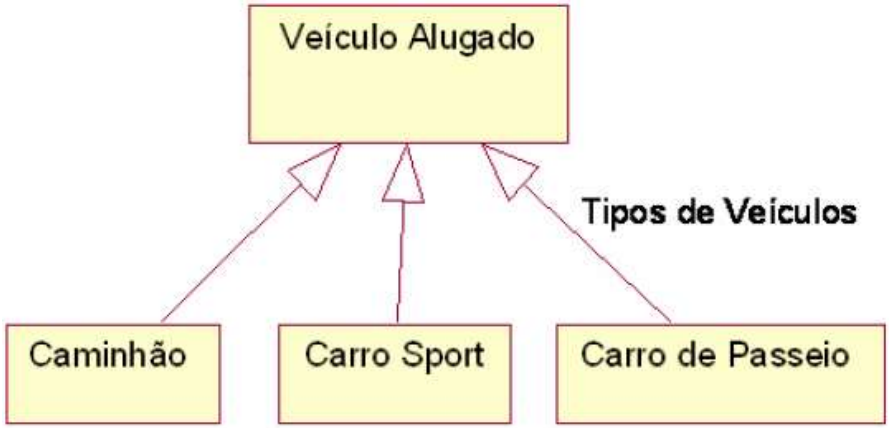
Exemplo



# Generalização

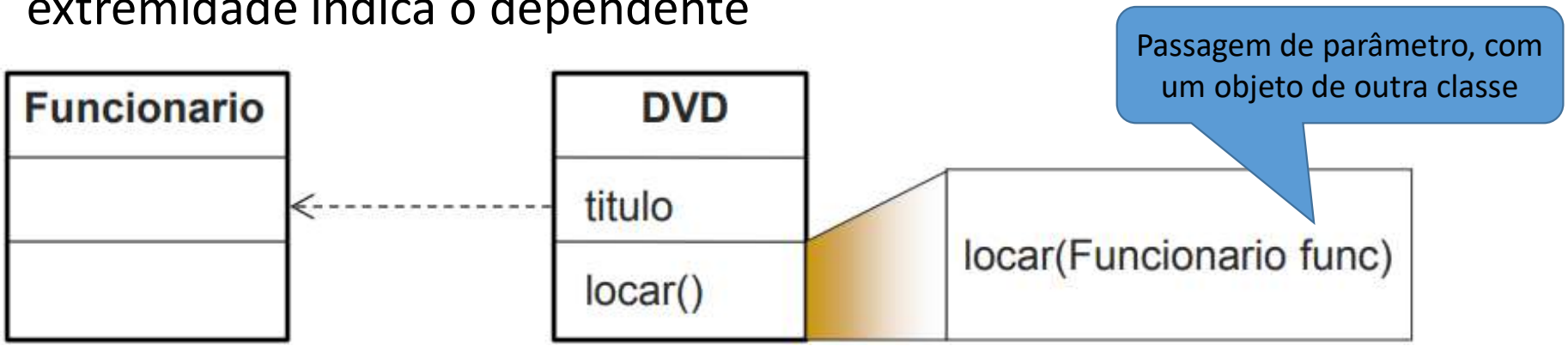
- Identificar super-classe (geral) e subclasses (especializadas)
- Semântica “é um”
- Tudo que a classe geral pode fazer, as classes específicas também podem. Atributos e métodos definidos na superclasse são herdados pelas subclasses



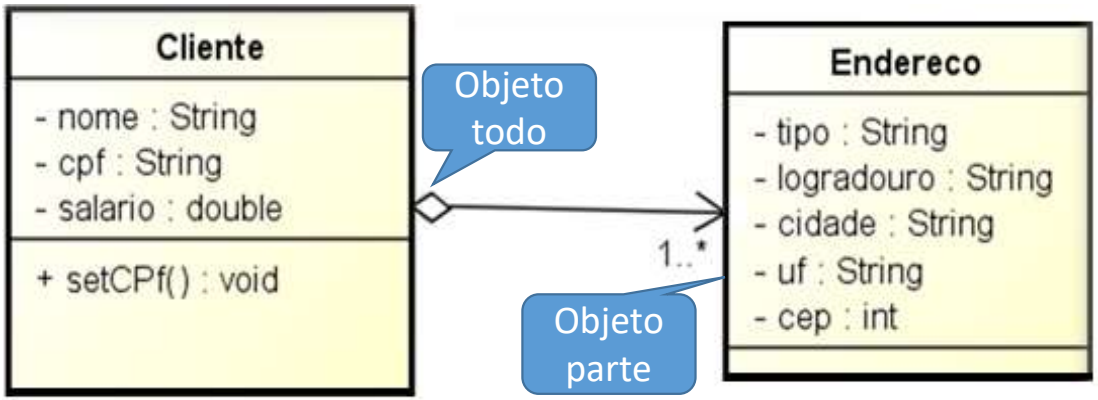
# Dependência

- Tipo menos comum de relacionamento
- Identifica uma ligação fraca entre objetos de duas classes, onde uma classe é dependente de outra classe, mas não tem uma dependência estrutural interna
- Representado por uma reta tracejada entre duas classes, a seta na extremidade indica o dependente



# Agregação

- **Agregação** é uma associação especial para demonstrar que, as informações de um objeto precisam ser complementadas pelas informações contidas em um ou mais objetos de outra classe.
  - Uma classe representa um item maior (**todo**) formado por itens menores (**parte**)
  - A **parte** pode se relacionar com o **todo** em um **determinado momento**
- Associação conhecida como relação "tem-um"
  - Um objeto da classe **todo** tem um objeto da classe **parte**
  - Os objetos "**parte**" podem existir sem **serem parte** do objeto "**todo**"



# Exemplo - Agregação

```
class Cliente
{
    string nome
    string rg;
}
```

```
class Conta
{
    int numero;
    Cliente titular;
    double saldo
}
```

```
public class Programa
{
    public static void Main(string[] args)
    {
        Cliente cli= new Cliente();
        cli.nome = "Lia";
        cli.rg = 111;
        Conta conta1= new Conta();
        conta1.titular= cli;

        // ou

        Conta conta2= new Conta();
        conta2.titular= new Cliente();
        conta2.titular.nome = "Téo";
        conta2.titular.rg = "222"
    }
}
```

# Vetor de objetos - Agregação

```
public class Conta
{
   int numero;
   List<Cliente> vetTitular;
   double saldo;

   public Conta(){
     vetTitular= new List<Cliente>();
   }
   public void AdicionarCliente(Cliente
c){
     vetTitular.Add(c);
   }
}
```

```
public class Cliente
{
   string nome
   string rg;

   public Cliente(string nome){
     this.nome = n;
   }
}
```

```
public class Programa
{
   public static void Main(string[] args)
   {
        Cliente cli1 = new Cliente("Lia");
        Cliente cli2 = new Cliente("Téo");

        Conta conta = new Conta();
        conta.AdicionarCliente(cli1);
        conta.AdicionarCliente(cli2);
    }
}
```

# Vetor de objetos

- Outros métodos:

```
lista.Remove(cli1);//remove pelo objeto
lista.RemoveAt(0);//remove pelo índice 0
int qtdeElementos = lista.Count();
```

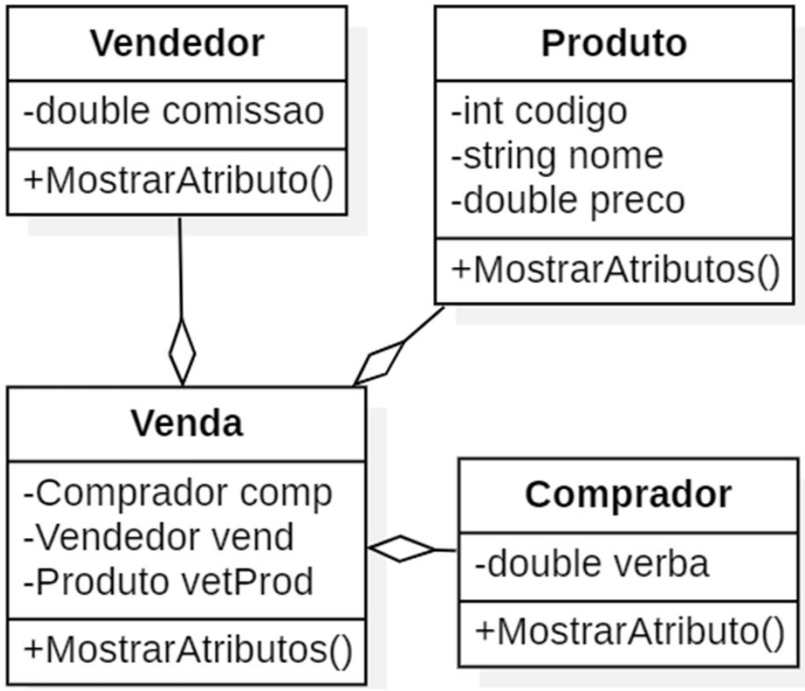
- Apresentando os elementos da lista

O vetor **deve** ter conteúdo

```
foreach(Cliente vc in vetTitular)
{
        Console.WriteLine("Titular : " + vc.Nome);
}
```

# Agregação Venda - Exercício

- Criar as classes de acordo com o trecho de diagrama de classe anexo
- A comissão do vendedor é gerada referente a 2% do preço do produto vendido
- Na instância de um comprador, conceda um valor de verba
- Quando realizar uma venda subtraia o valor da verba
- É para ser realizada mais de uma venda, para usar o vetor/List de produtos
- No cadastro de produtos, o código inicial é 500
- Use encapsulamento completo (com tratamento de valores digitados) para os atributos da classe Produto
- Realizar instâncias necessárias para testar as classes

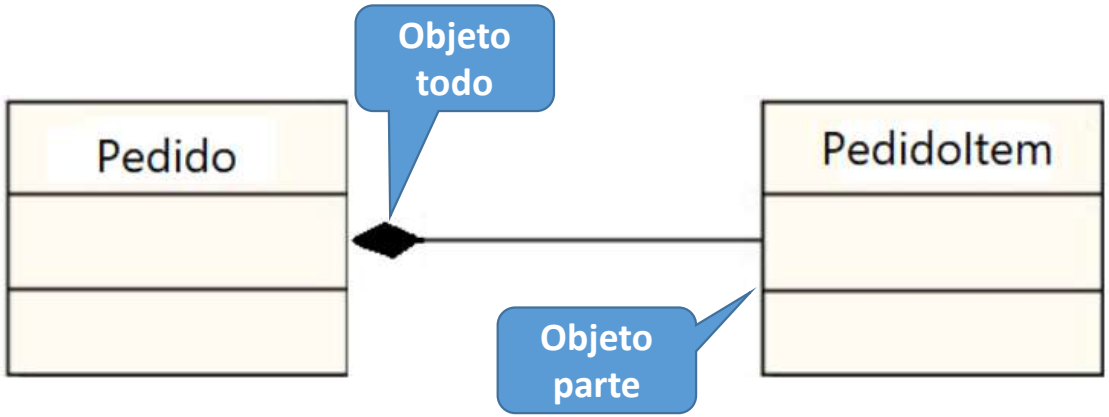


# Composição

- Composição é um tipo de associação ainda **mais forte** que agregação.
- Associação conhecida como relação “é composta de”
  - Um objeto é composto por seus objetos associados
- A composição também é um relacionamento caracterizado como parte / todo, mas neste caso, o todo é responsável pelo ciclo de vida da parte.
- A existência do **Objeto-Parte NÃO faz sentido se o Objeto-Todo não existir**, ou seja, o objeto-todo é responsável por destruir e criar suas partes.

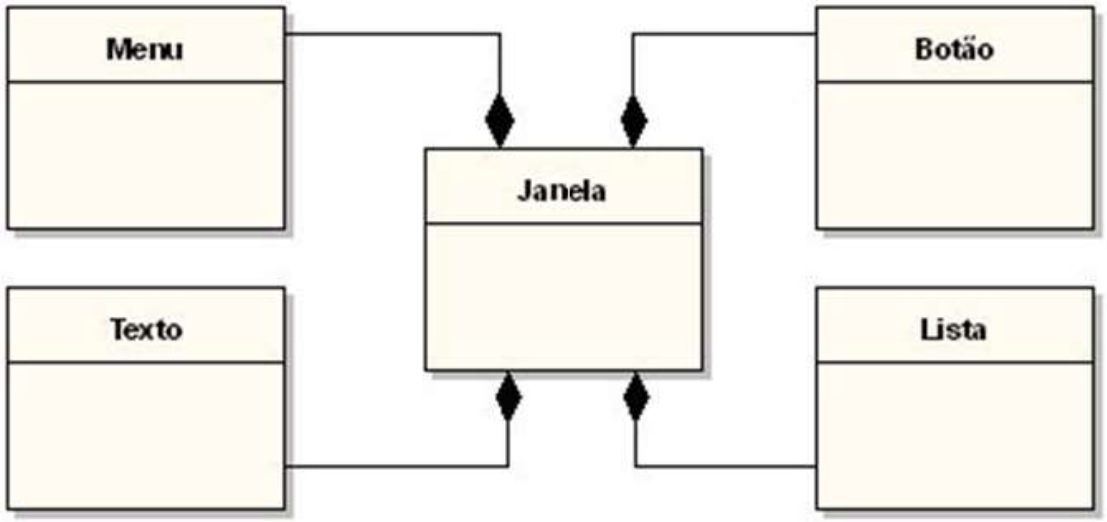
# Composição

- Neste exemplo




- Um pedido é composto por um ou vários itens, se a relação entre a classe Pedido e a classe PedidoItem for rompida, o objeto que representa a classe PedidoItem não pode existir

# Composição




```
public class A {
    private B b;
    public A( ){
        b = new B();
    }
}
```

```
public class B {
    public B( ){
    }
}
```

# Exemplo - Composição


```
public class NotaFiscal
    {
        public int NumeroNf { get; set; }
        public string Data { get; set; }
        public List<ItemNf> VetItem { get; set; }

        public NotaFiscal(int num, string dt)
        {
            NumeroNf = num;
            Data = dt;
            VetItem = new List<ItemNf>();
        }
        public void AdicionarItens(int qtde)
        {
            VetItem.Add(new ItemNf(qtde));
        }

        ~NotaFiscal()//destrutor
        {
            VetItem = null;
            Console.WriteLine("Destruindo a nota fiscal");
        }
    }
```

```
public class ItemNf
    {
        public int Quantidade { get; set; }
        public ItemNf(int qtde)//construtor
        {
            Quantidade = qtde;
        }
        ~ItemNf()//destrutor
        {
            Console.WriteLine("Destruindo o item
nota fiscal");
        }
    }
```

```
Destrutor Nota Fiscal ....
Destrutor Item de Nota Fiscal ....
Destrutor Item de Nota Fiscal ....
```

```
// função-método Main()
using ComposicaoNotaFiscal;

NotaFiscal? nf = new NotaFiscal(1, "19/09/2023");
nf.AdicionarItens(13);
nf.AdicionarItens(31);

foreach (ItemNf it in nf.VetItem)
    Console.WriteLine("Quantidade: " +
it.Quantidade);

nf = null;
GC.Collect();
```

# Destrutor

- São utilizados para destruir instâncias de classes
- Função inversa à função dos Construtores
- Construtores permitem criar regras para a criação de instâncias
- Destrutores permitem criar regras para a destruição dessas instâncias
- C# você nunca pode chamá-los explicitamente, a razão é que não se pode destruir um objeto
- Quem controla o destrutor? Resposta: O .NET Garbage Collector
- Sintaxe:

```
~NomeDaClasse{
    //alguma lógica
}
```

# Destrutor

- Só podem ser definidos em Classes
- Não podem ser herdados ou sobrecarregados
- Não possuem modificadores de acesso (public, private, protected etc.)
- Não possuem parâmetros
- Não podem ser chamados, pois são invocados automaticamente
- Uma instância pode ser destruída quando nenhum outro recurso utiliza essa instância
- O programador não possui controle de quando o Destrutor será chamado, pois isso é determinado pelo Garbage Collector
- Destrutores podem ser chamados quando uma aplicação termina seu processamento
- O GC realiza a destruição dos objetos alocados em memória

# Destrutor

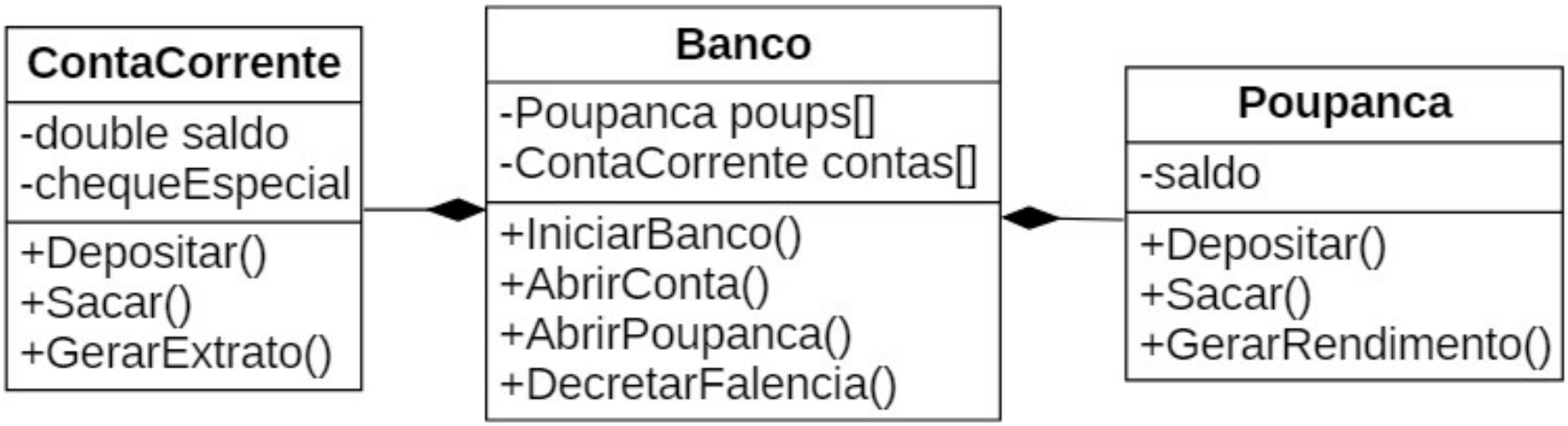
- Quando as classes trabalham com Arquivos, Conexões e Recursos de Sistemas Operacionais, devemos sempre que possível utilizar Destrutores, para limpar a memória que esses objetos utilizam.

- Não utilizar Destrutores vazios. Pois quando uma classe contêm um Destrutor ela entra na fila de Finalização do GC. Se este Destrutor estiver vazio, a classe irá estar nesta fila inutilmente, o que gera alocação de memória desnecessária.

# Comparação entre Agregação e Composição

- Ocorre quando existe uma classe que possui um atributo de outra classe (objetos ou array de objetos )
- Agregação: as partes **podem existir** sem o todo
- Composição: as partes **não existem** sem o todo
- **Não há diferença na implementação e sim no comportamento**

# Exercício – Composição - Dos métodos, crie a lógica que for conveniente a cada classe



# Exercícios - Composição