alura

01000101 01110011 01100011 01101111 01101100 01100001
00100000 01100100 01100101 00100000 01000100 01100001
01100100 01101111 01110011 00001010

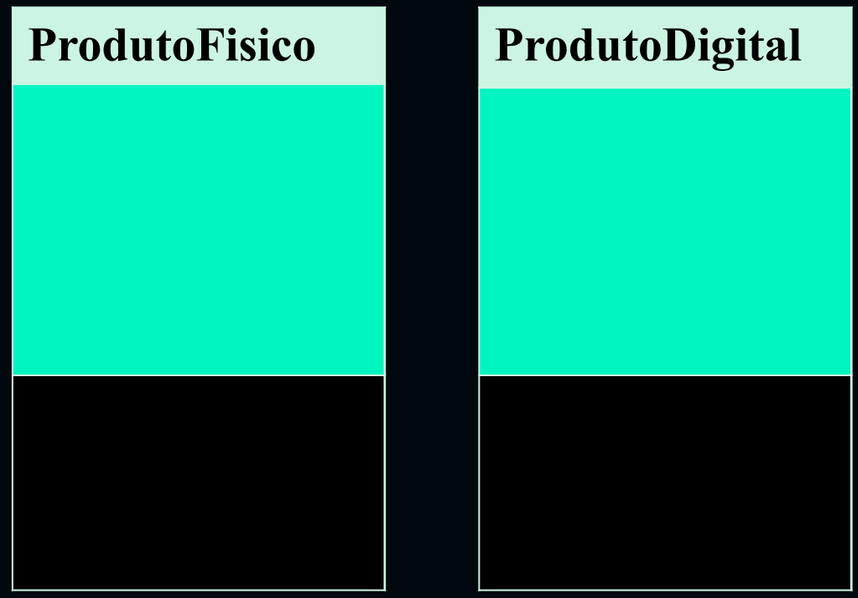
HERANÇA

Praticando C# e OO

Instrutora: Iasmin Araújo

ESCOLA_ **PROGRAMAÇÃO**

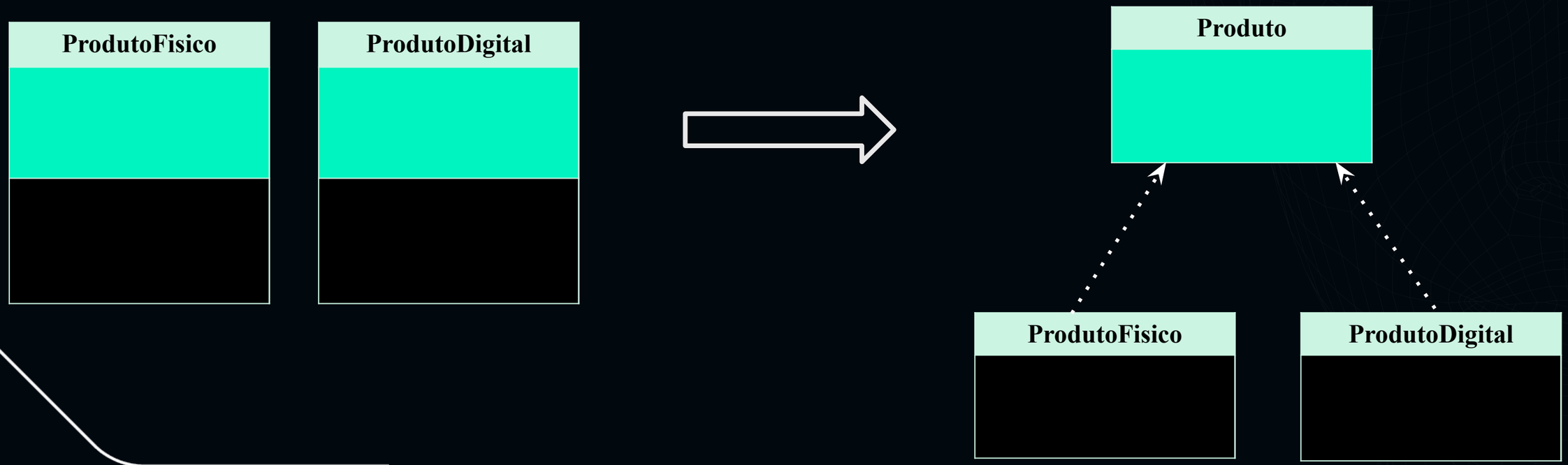
Existem casos em que classes diferentes podem conter podem conter trechos de código iguais. No entanto, essa **duplicação não é uma boa prática**. Provavelmente, ao modificarmos algo em uma classe, será necessário fazer a mesma alteração na outra, o que aumenta o risco de erros e dificulta a manutenção.



Podemos trabalhar com o conceito de **herança** para resolver esse problema de duplicação de código!

Para aplicar a **herança**, vamos criar uma classe mãe, com todos os atributos que estão duplicados. Faremos as classes filhas herdarem da classe mãe, compartilhando esse mesmo código.

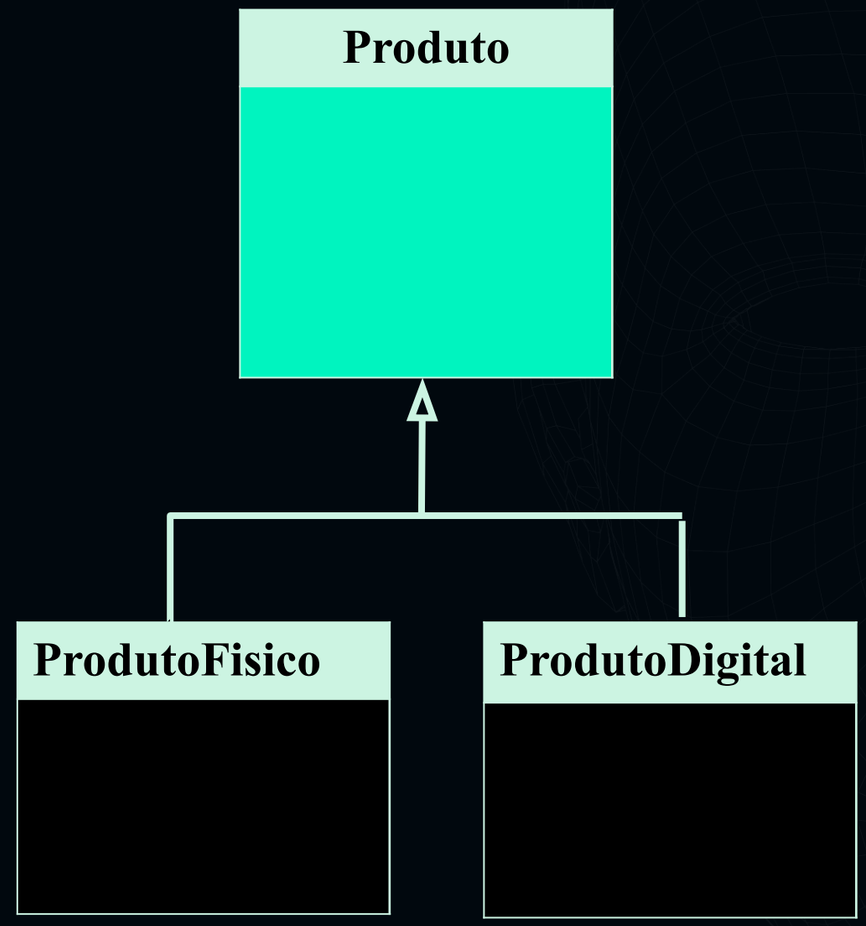
Nas classes filhas, haverá os códigos mais específicos, enquanto na classe mãe haverá o código genérico.



REPRESENTAÇÃO VISUAL

Utilizamos a representação anterior para facilitar a compreensão.

Porém, ao trabalharmos com herança, podemos representar a relação entre as classes utilizando **diagramas UML**.



```
class ProdutoFisico : Produto
{
    public int Estoque { get; }

    public bool EstaDisponivel()
    {
        return Estoque > 0;
    }
}
```

APLICANDO A HERANÇA

- ✔ Para extender de uma classe (usar a herança), basta utilizar os dois pontos.
- ✔ No exemplo ao lado, dizemos que `ProdutoFisico` é filha de `Produto`, que é a classe mãe.
- ✔ A herança modela relações do tipo **é um**. Logo, podemos dizer que `ProdutoFisico` é um `Produto`.

```
public ProdutoFisico(string nome, string
descricao, decimal preco, string imagem)
    : base(nome, descricao, preco,
imagem)
{
    this.Estoque = 0;
}
```

HERANÇA EM CONSTRUTORES

- ✔ Podemos aplicar herança em construtores, utilizando a palavra chave `base` e os dois pontos também.
- ✔ Com isso, ao construir o objeto filho, será inicialmente aplicado o construtor da classe mãe.

```
// classe Produto
    public string Nome { get; protected
set };
```

```
public ProdutoFisico(string nome, string
descricao, decimal preco, string imagem)
    {
        this.Nome = nome;
        // outras propriedades
    }
```

HERDANDO PROPRIEDADES

- ✔ Caso seja necessário alterar alguma das propriedades herdadas na classe filha, podemos declarar os sets com o modificador `protected`.
- ✔ Esse modificador permite que os membros sejam visíveis dentro da própria classe e também dentro de suas classes filhas.

CLASSES ABSTRATAS

- Em alguns casos, não faz sentido que a classe mãe possa ser instanciada.
- Com isso, ela pode apenas representar um conceito, sendo abstrata.

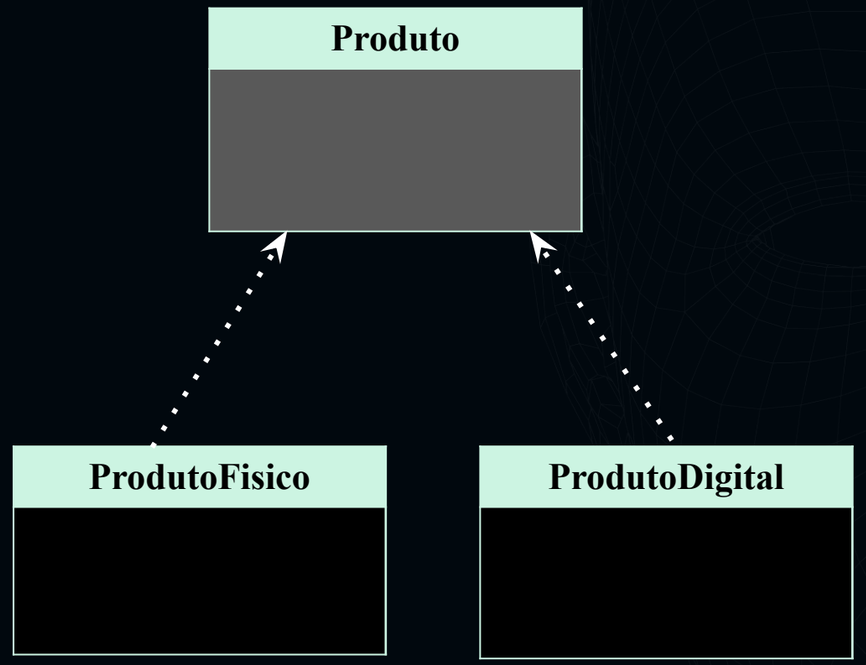
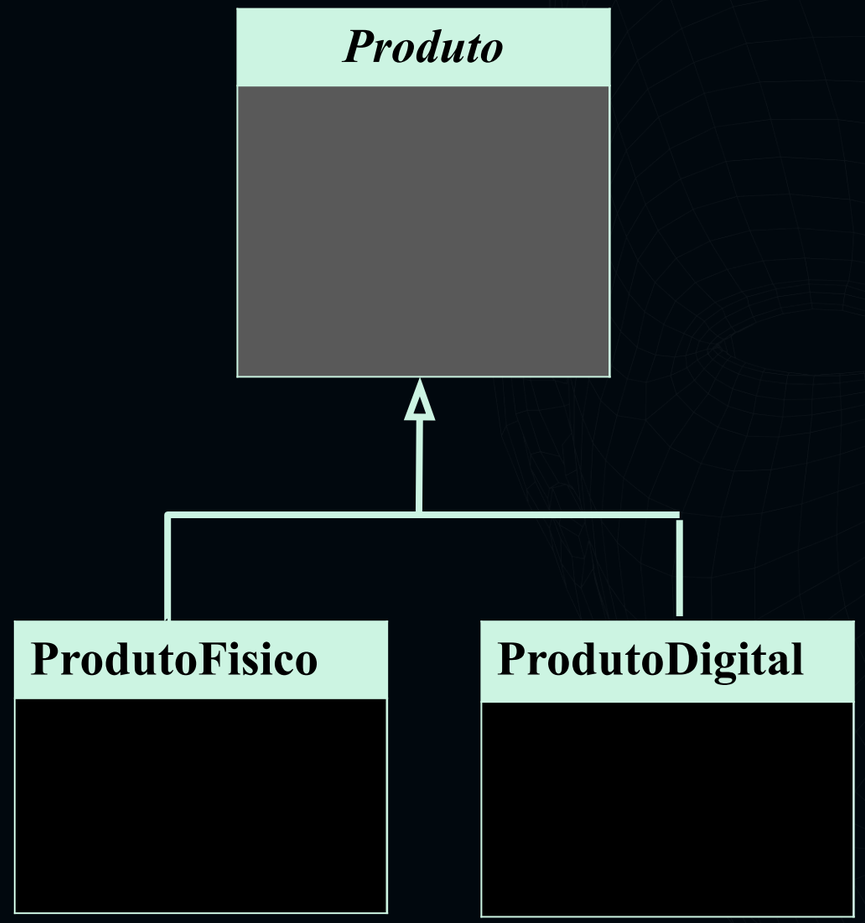


DIAGRAMA UML

- ✔ Para representar classes abstratas com diagramas UML, devemos deixá-las com o nome em itálico no desenho.



CLASSES ABSTRATAS

- ✔ Para declarar uma classe como abstrata, utilizamos o modificador `abstract` antes da palavra chave `class`.

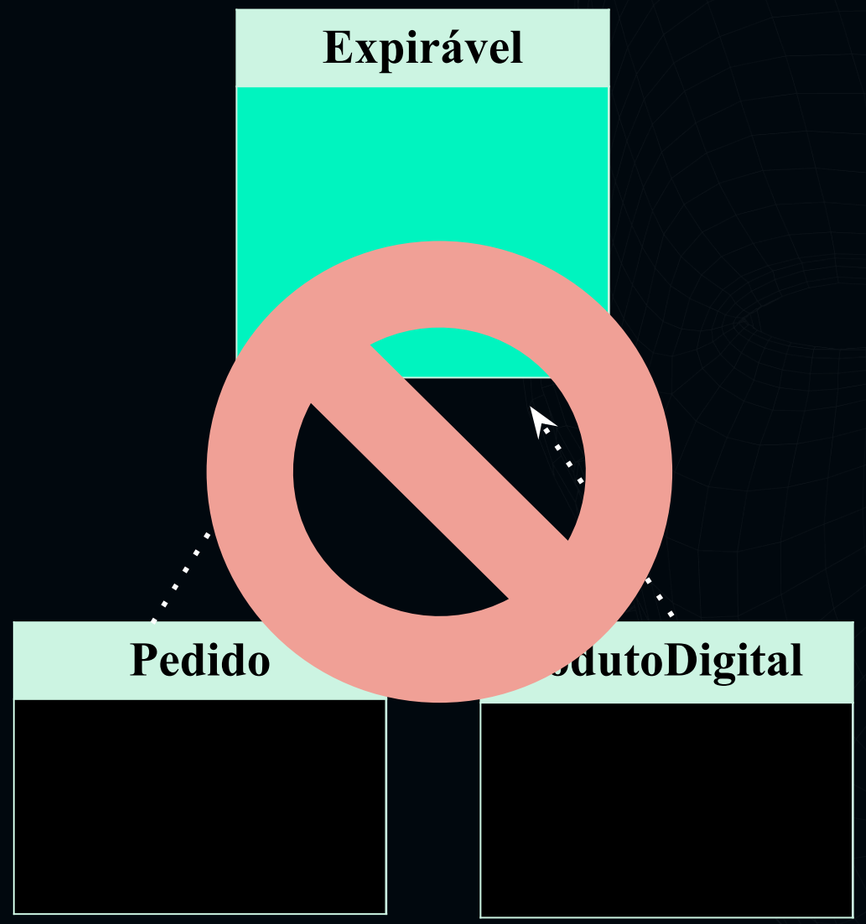
```
abstract class Produto
{
    // código abstrato da classe
}
```

```
Produto produto = new Produto("Teclado",
    "Modelo compacto e silencioso, perfeito " +
    "para produtividade diária.", 80.00m);
```

Erro de compilação ao tentar instanciar a classe abstrata `Produto`.

COMPARTILHANDO COMPORTAMENTOS

- ✔ Existem casos em que classes compartilham comportamentos, mas não faz sentido criar um relacionamento de herança para elas.
- ✔ Nesses casos, podemos optar por trabalhar com **interfaces**.



```
interface IExpiravel
{
    bool EstaExpirado();
}
```

Apenas declaramos os métodos. Eles deverão ser implementados nas classes que utilizarem a interface.

DECLARANDO INTERFACES

- ✔ Para criar interfaces, utilizamos a palavra chave `interface`.
- ✔ Em C#, é uma convenção que interfaces comecem com o prefixo `I`.
- ✔ Não utilizamos modificadores de acesso em interface. Por padrão, métodos são públicos, já que devem ser implementados em outras classes.

```
class Pedido : IExpiravel
{
    public bool EstaExpirado()
    {
        return !pago && DateTime.Now >
          Data.AddMinutes(15);
    }
}
```

```
class ProdutoDigital : Produto, IExpiravel
{

    public bool EstaExpirado()
    {
        return DateTime.Now >
          DataCompra.AddYears(2);
    }
}
```

USANDO INTERFACES

- ✔ Ao utilizarmos interfaces em uma classe, dizemos que a classe implementa a interface.
- ✔ Para fazer isso, usamos os dois pontos, assim como na herança.
- ✔ **Interfaces definem contratos:** se uma classe implementa uma interface, ela é obrigada a fornecer implementações para todos os métodos declarados na interface.

```
class Pedido : IExpiravel, IPagavel
{
    public bool EstaExpirado()
    {
        return !pago && DateTime.Now >
        Data.AddMinutes(15);
    }

    public void Pagar()
    {
        pago = true;
    }
}
```

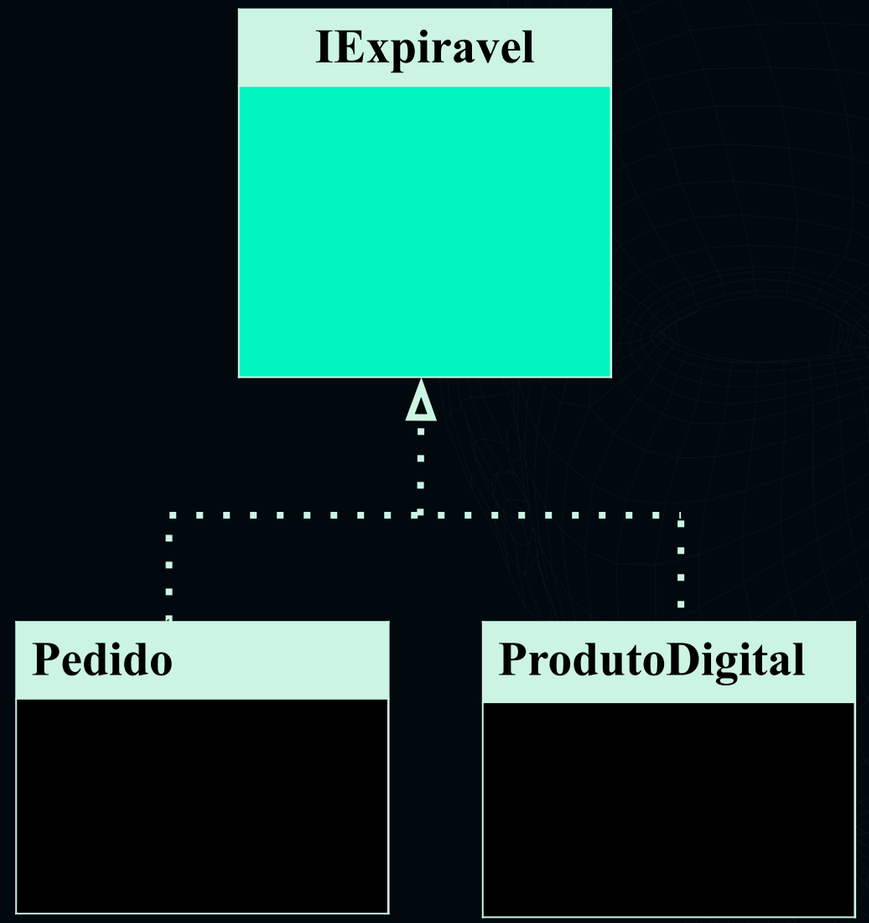
Pedido **é um** IExpirável e também **é um** IPagável.

HERANÇA MÚLTIPLA

- ✔ Uma classe pode implementar mais de uma interface ao mesmo tempo. Isso porque interfaces servem para compartilhar comportamentos.
- ✔ Já no caso de herança, em C# não é possível herdar de mais de uma classe ao mesmo tempo. Assim, dizemos que não há suporte a herança múltipla.
- ✔ Assim como a herança, interfaces modelam relações do tipo **“é um”**.

REPRESENTAÇÃO VISUAL

- ✔ Podemos representar a relação de interfaces em diagrama UML conforme o exemplo ao lado. As setas são pontilhadas, ao invés de contínuas como na herança.



```
abstract class Produto
{
    private string imagem;
    public string Nome { get; protected
set; }
    public string Descricao { get; }
    public decimal Preco { get; private
set; }
    public int Nota { get; private set; }
    public string Comentario { get;
private set; }
}
```

AGRUPANDO ATRIBUTOS E PROPRIEDADES

- ✔ Ao trabalhar com classes, podemos notar que alguns atributos não são tão relacionados com a classe a que pertencem.
- ✔ Por exemplo, ao lado, as propriedades `Nota` e `Comentario` fazem muito mais sentido se forem agrupadas em outra classe `Avaliacao`.

```
class Avaliacao
{
    public int Nota { get; private set; }
    public string Comentario { get; private
set; }
}
```

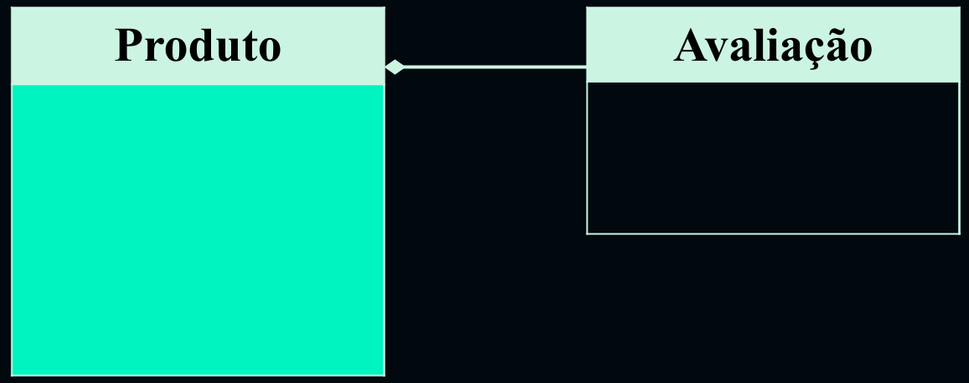
```
abstract class Produto
{
    private string imagem;
    public string Nome { get; protected set; }
    public string Descricao { get; }
    public decimal Preco { get; private set; }
    public Avaliacao Avaliacao { get; private set; }
}
```

COMPOSIÇÃO

- ✔ Ao agruparmos as informações, estamos fazendo **composição**.
- ✔ A classe `Produto` é composta por `Avaliacao`. Dizemos também que a classe `Produto` **tem uma** `Avaliacao`.
- ✔ Utilizar composição auxilia nas boas práticas, evitando os mesmos atributos duplicados em várias classes.

REPRESENTAÇÃO VISUAL

- Podemos representar a relação de composição conforme o diagrama UML ao lado.



RESUMINDO: COMO MODELAR AS CLASSES?

HERANÇA

- ★ Relação “é um”.
- ★ Utilizada quando há muito código compartilhado: as classes são naturalmente muito parecidas.

INTERFACE

- ★ Também apresenta relação “é um”.
- ★ Compartilhamento de comportamentos.
- ★ Contrato entre interface e classes que a implementam.

COMPOSIÇÃO

- ★ Relação “tem um”.
- ★ Agrupamento de atributos, possibilitando reaproveitamento de código.

alura

Compartilhe um resumo de seus novos conhecimentos em suas redes sociais.

#aprendizadoalura