

UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE PROCESSAMENTO DE ENERGIA ELÉTRICA



Classes e métodos abstratos

Interfaces

Curso de Engenharia de Controle e Automação
DPEE1090 - Programação orientada a objetos para automação

Prof. Renan Duarte

1º semestre de 2024

Sumário

Classes e métodos abstratos e Interfaces

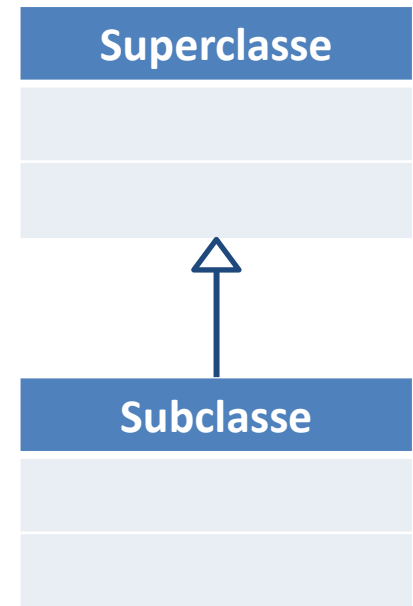
- Introdução
- Métodos abstratos
- Classes abstratas
- Interfaces

Relembrando

Herança

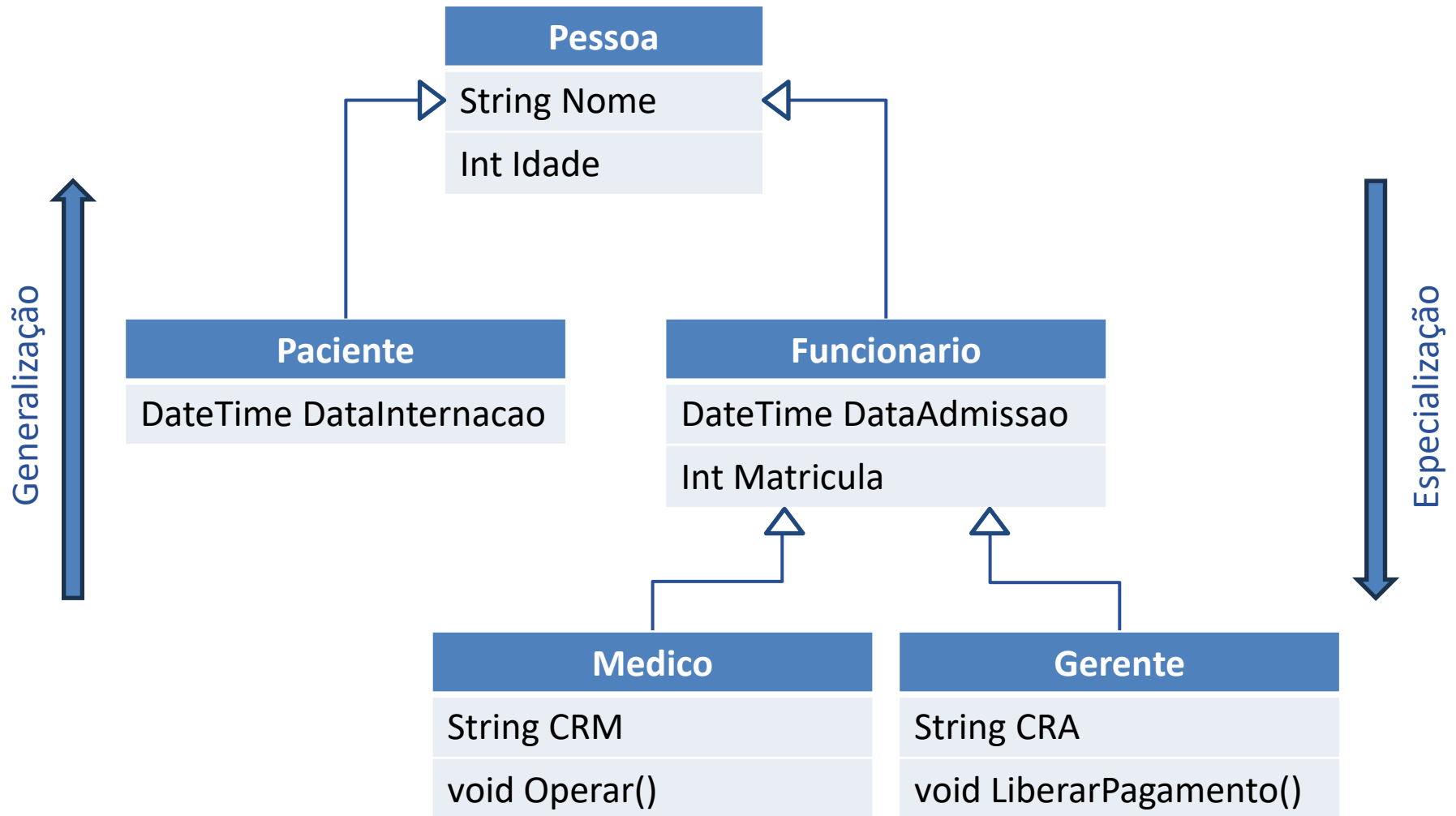
Relacionamento **entre classes** em que uma classe chamada de subclasse (ou classe filha, classe derivada) é uma extensão (um subtipo) de outra classe chamada de superclasse (ou classe pai, classe mãe, classe base).

- Subclasse estende funcionalidades da superclasse
- Generalização e especialização



Relembrando

Herança

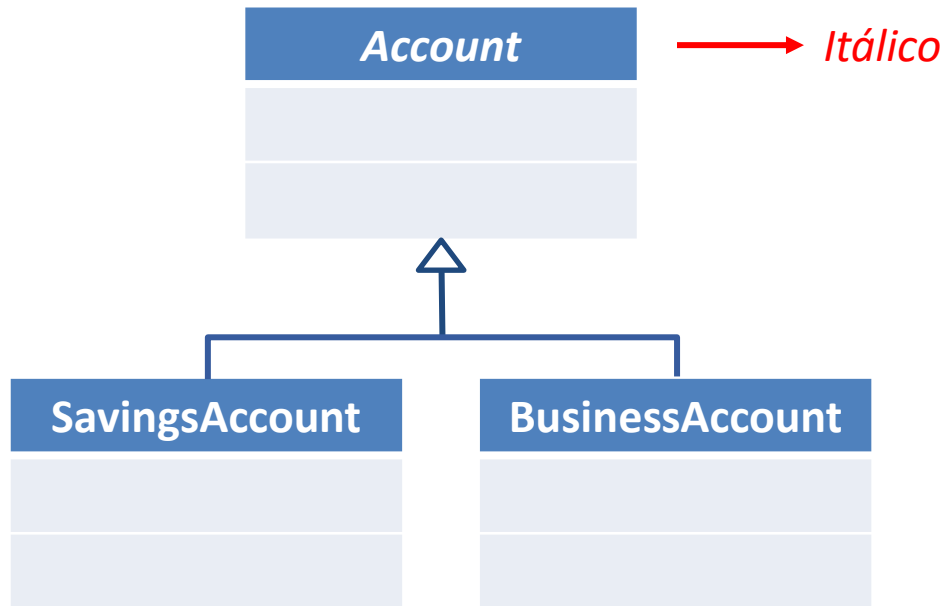


Classes abstratas

Conceito

São classes que não podem ser instanciadas

- É uma forma de garantir herança total: somente subclasses não abstratas podem ser instanciadas, mas nunca a superclasse abstrata



Classes abstratas

Conceito

As classes abstratas podem conter:

- Atributos, como toda classe;
- Métodos implementados (concretos);
- Métodos declarados mas não implementados (abstratos)
 - Geralmente sabemos o que esses métodos precisam fazer
 - Sabemos também que a execução do método pode ser diferente dependendo do subtipo de objeto em questão

Fornece alguns comportamentos padrão em métodos concretos

O programador é **FORÇADO** a implementar métodos em uma subclasse antes que qualquer objeto possa ser instanciado.

Métodos abstratos

Conceito

São métodos que são declarado sem uma implementação (sem chaves e seguido por um ponto e vírgula)

- Servem como uma assinatura que deve ser implementada pelas subclasses, definindo um contrato para que estas forneçam uma implementação específica

Assinatura de método

- A assinatura de um método é a definição que inclui o nome do método, o tipo de retorno, os tipos e a ordem dos parâmetros, mas não inclui o corpo do método nem a implementação:
 - `int sum()`
 - `int sum(int a, int b)`
- Mesmo nome mas assinaturas diferentes

Classes e métodos abstratos

Observações

Para que uma classe tenha métodos abstratos, a própria classe deve ser abstrata:

```
abstract class Vehicle {  
    public abstract void StartEngine();  
}
```

A própria classe abstrata não pode ser instanciada diretamente:

```
Vehicle myVehicle = new Vehicle(); // Erro de compilação
```

Pode ter dados e métodos concretos:

```
abstract class Vehicle {  
    public abstract void StartEngine();  
    public void FuelUp() {  
        Console.WriteLine("Filling up the tank...");  
    }  
}
```


Classes e métodos abstratos

Implementação em C# - Superclasse

```
public abstract class MyClass
{
    // Atributo da classe
    public int MyProperty { get; set; }

    // Construtor
    public MyClass(int myproperty) {
        MyProperty = myproperty;
    }

    // Método concreto
    public int GetProperty() {
        return MyProperty;
    }


    // Método abstrato
    public abstract int MyMethod();
```

→ Não existe implementação do método

Classes e métodos abstratos

Implementação em C# - Subclasses

```
public class MySubClass : MyClass
{
    // Construtor
    public MySubClass(int myproperty) : base (myproperty) {}

    // Implementação do método abstrato da superclasse
    public override int MyMethod()
    {
        return 1 + 1;  Implementação
                        do método
    }
}
```

Classes e métodos abstratos

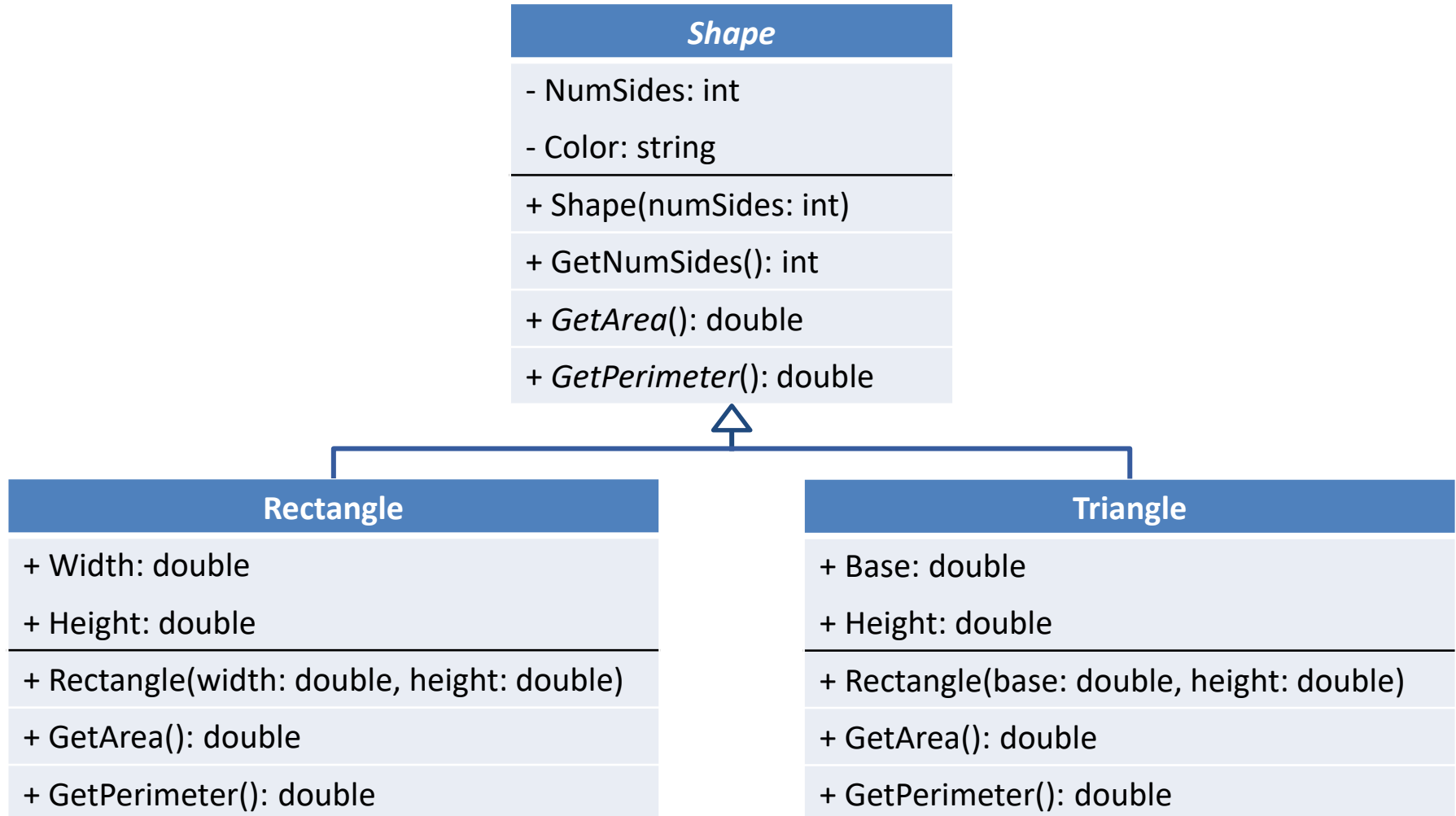
Questionamentos

Se uma classe abstrata não pode ser instanciada, por que simplesmente não criar somente as suas subclasses diretamente?

Resposta:

- Reuso de código: Mantém comportamentos e dados comuns a múltiplas classes em um único lugar
- Polimorfismo: Permite tratar de forma fácil e uniforme todos os objetos derivados de um ancestral comum (mais detalhes nas próximas aulas)

Exemplo 1 – Básico



Exercício 1

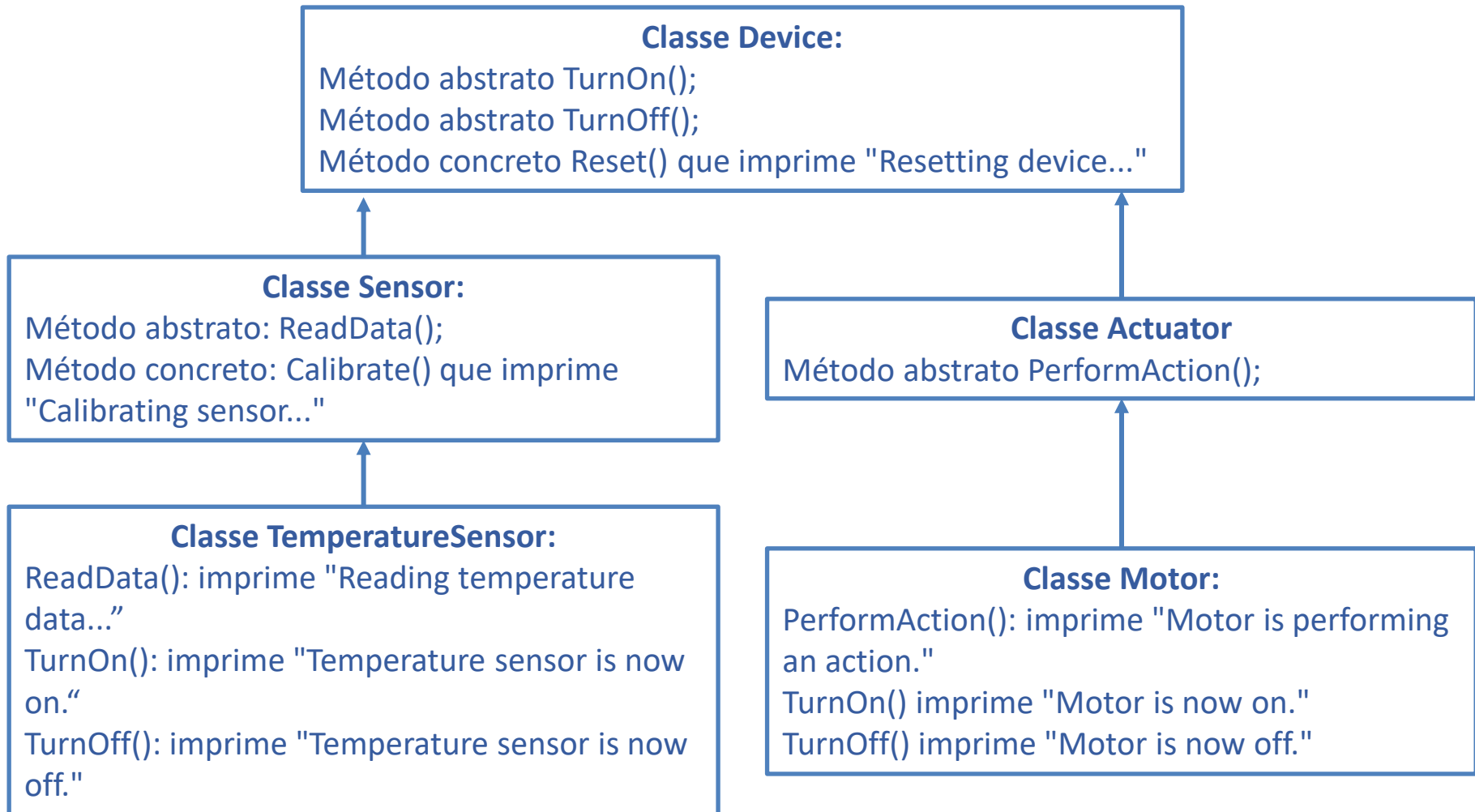
Você foi contratado para desenvolver um sistema de controle para uma linha de produção automatizada. A linha de produção contém diferentes tipos de dispositivos, como sensores e atuadores. Cada dispositivo deve ser capaz de ser ligado, desligado e reiniciado.

Sensores devem ter a capacidade de ler dados e atuadores devem realizar uma ação específica.

Utilizar classes abstratas e métodos abstratos para modelar os dispositivos na linha de produção, garantindo que todos os dispositivos compartilhem uma interface comum e que cada tipo de dispositivo implemente sua funcionalidade específica.

Os requisitos de cada classe são dados no próximo slide.

Exercício 1 – Continuação



Interfaces

Definição

Interface é um tipo que define um conjunto de operações que uma classe (ou struct) deve implementar

Ela estabelece um **contrato** que a classe (ou struct) deve obrigatoriamente cumprir

Objetivo: Criar sistemas com **baixo acoplamento e flexíveis**

```
interface IShape {  
    double Area();  
    double Perimeter ();  
}
```



Em C#, tipicamente se usa o prefixo "I" antes do nome da interface

Interfaces

Conceito

Uma interface é como uma *checklist*

- Uma classe que implementa uma interface deve implementar/definir todos os métodos declarados na interface
- Na interface, todas as declarações de métodos omitem o corpo
- Enquanto na herança **se estende** uma classe, com interfaces **se implementa** uma interface

Por que usar interfaces?

- Definir um conjunto de comportamentos
- Permitir "herança múltipla" implementando várias interfaces

Exemplo 2 – Básico

Partindo do exemplo 1, vamos definir uma interface chamada `IResizable` que deve ser cumprida por todas as classes que precisam definir objetos que podem ser redimensionados.

```
// Define uma interface chamada IResizable
interface IResizable
{
    // Declaração de um método chamado Resize que aceita um
    // parâmetro do tipo double
    // Este método deve ser implementado por qualquer classe
    // que implemente esta interface
    void Resize(double scale);
}
```

Exemplo 2 – Continuação

Agora, para que os objetos criados a partir das classes `Rectangle` e `Triangle` sejam considerados redimensionáveis, estas devem cumprir o contrato estabelecido na interface `IResizable`:

```
class Rectangle : Shape, IResizable
{
    ...
    // Implementação do método Resize() da interface
    // IResizable para redimensionar o retângulo
    public void Resize(double scale)
    {
        Width *= scale;
        Height *= scale;
    }
}
```

Interfaces

Algumas interfaces comuns em C#

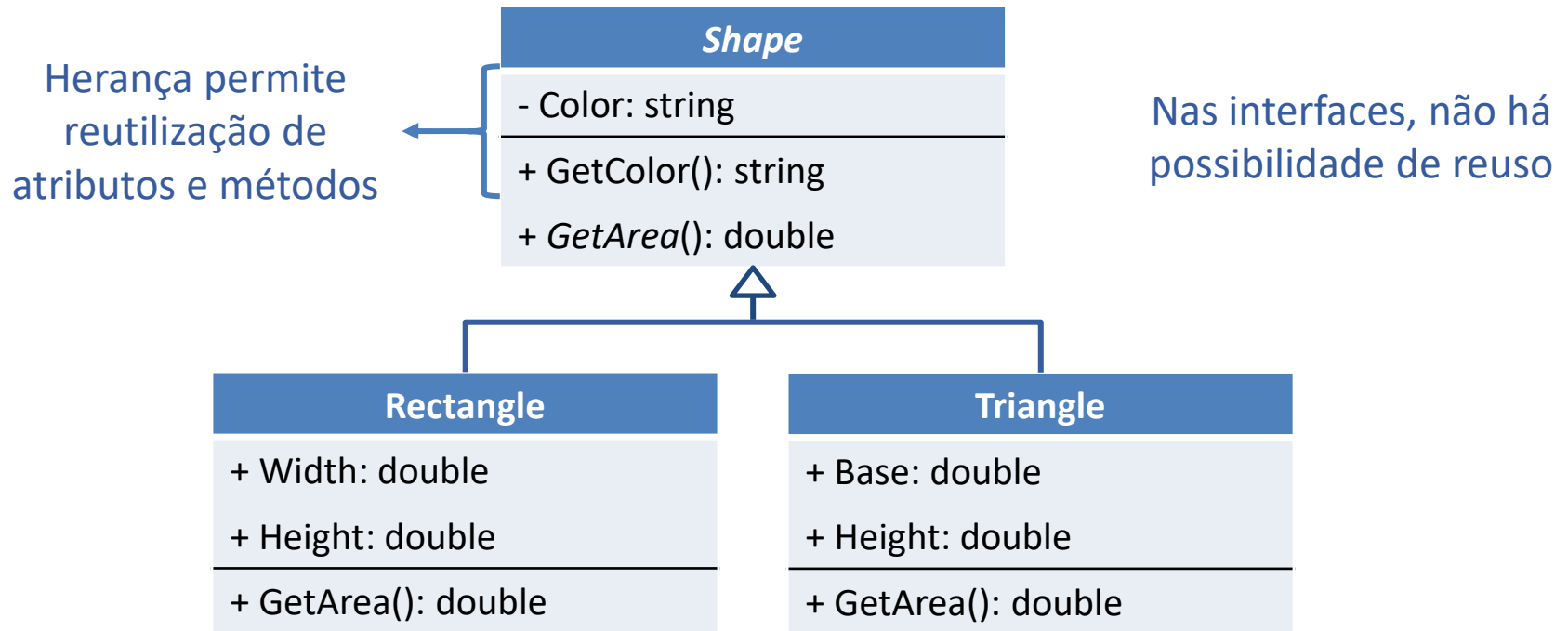
- `IEnumerable<T>` (and `IEnumerable`): Para uso com 'foreach' e LINQ, permitindo a iteração sobre uma coleção
- `IDisposable`: Para recursos que requerem limpeza, usado com a estrutura 'using'
- `IQueryable<T>`: Permite executar consultas contra fontes de dados consultáveis
- `IComparable<T>` e `IComparer<T>`: Para ordenação generalizada
- `IComparable<T>` e `IEqualityComparer<T>`: Para igualdade generalizada
- `IList<T>` e `ICollection<T>`: Para coleções mutáveis
- `IDictionary<T, K>`: Para coleções de busca (lookup)

Herança vs. interfaces

Diferença fundamental

Herança → Reuso

Interface → Contrato a ser cumprido



Herança vs. interfaces

Herdar vs. cumprir contrato

Herança:

- Quando há uma clara relação "é um(a)" (e.g., um Retângulo é uma Forma)
- Para compartilhar código comum entre classes

Interfaces:

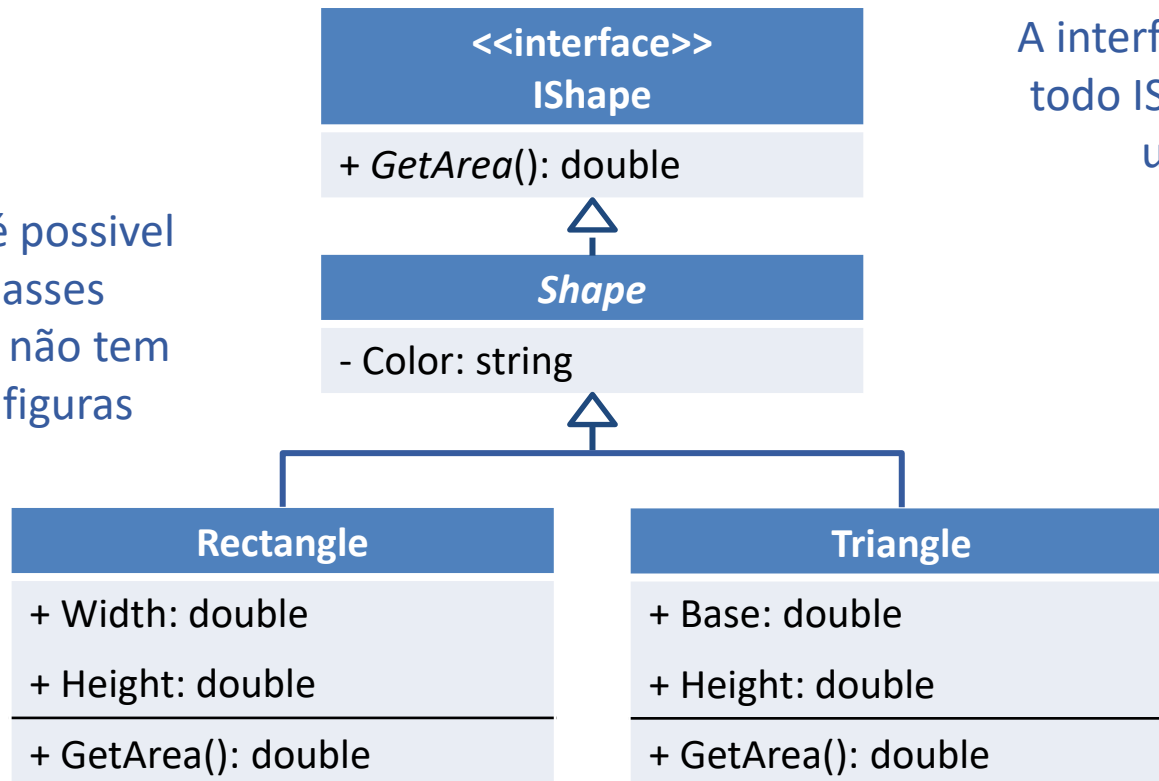
- Quando diferentes classes devem compartilhar o mesmo comportamento sem relação hierárquica
- Para permitir que uma classe implemente comportamentos de múltiplos contratos

Herança vs. interfaces

Herdar vs. cumprir contrato

E se precisarmos implementar Shape como interface mas também quisermos definir uma estrutura comum reutilizável para todas as figuras?

Dessa forma, é possível criarmos classes concretas que não tem cor mas são figuras



A interface define que todo IShape deve ter uma área

Exercício 2 – Desafio

Uma empresa de tecnologia que desenvolve veículos autônomos e carros convencionais deseja criar um sistema de gestão para esses dois tipos de veículos.

Ambos os tipos pertencem a categoria “Vehicle”, tendo atributos comuns a ambos como “Modelo”, “Cor” e “Ano” e funcionalidades definidas pela interface “IVehicle”. Os carros autônomos por sua vez, também implementam as funcionalidades requeridas pela interface “INavigation”.

O próximo slide apresenta o diagrama UML do problema.

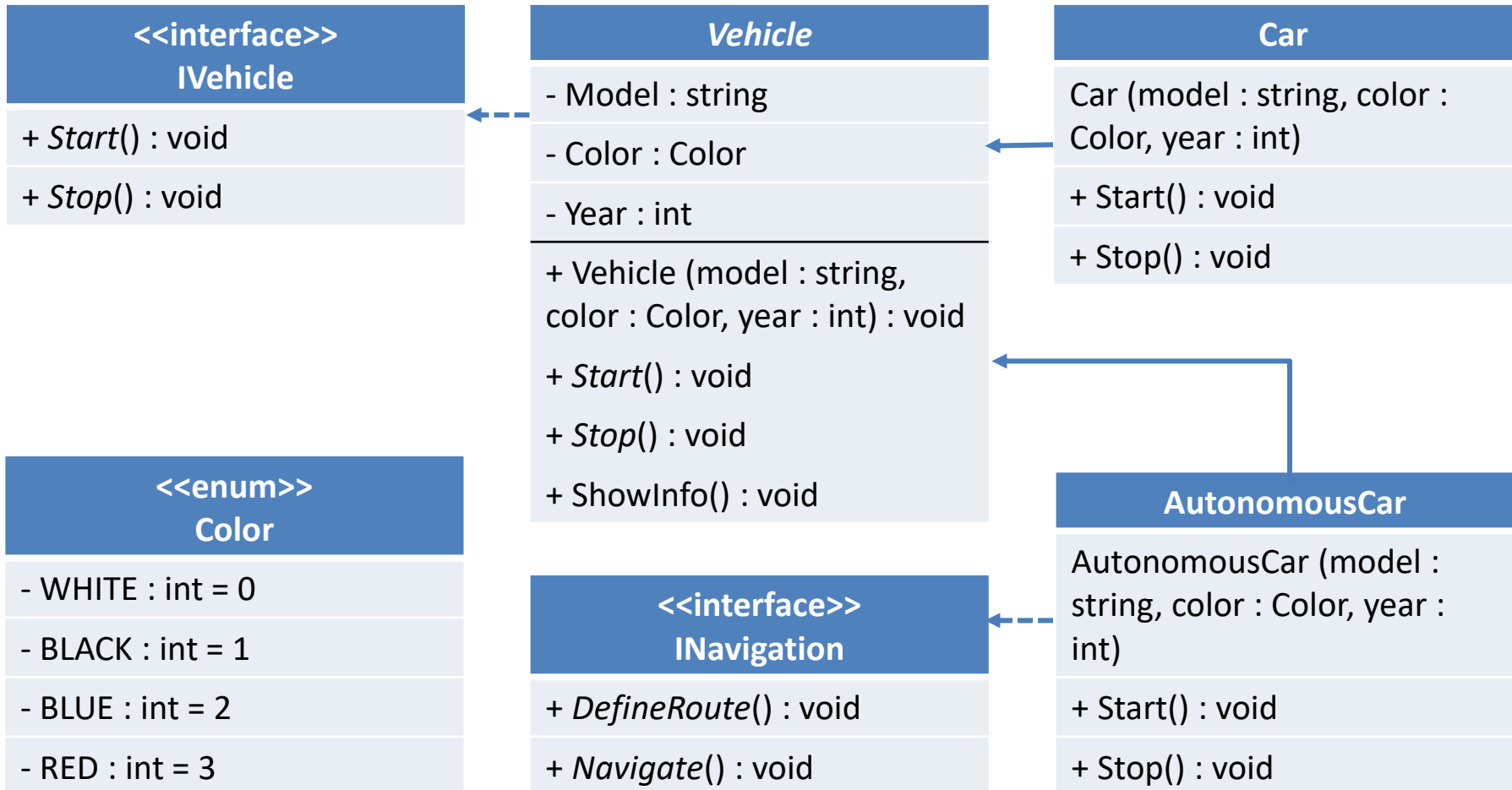
Implemente as classes, interfaces e enumerações necessárias e crie um programa para demonstrar as funcionalidades da solução

Herança vs. interfaces



tinyurl.com/2rexa7tx

Exercício 2 – Continuação



Revisão

Próxima aula

Polimorfismo

- Definição de polimorfismo e sua importância na POO
- Polimorfismo estático: sobrecarga de métodos e operadores
- Polimorfismo dinâmico: substituição de métodos em tempo de execução

Dúvidas?

renan.duarte@gedre.ufsm.br

GEDRE – Prédio 10 – CTLAB