

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



# Herança

---

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

---

Prof. Renan Duarte

1º semestre de 2024

# Sumário

---

## Herança

- Definição e conceito de herança em POO
- Vantagens
- Recomendações e boas práticas de uso da herança
- Tipos de herança
- *Upcasting* e *downcasting*
- Sobreposição
- Classes e métodos selados

# Introdução

---

## Conceitos relacionais

Até o momento, estudamos conceitos da POO como abstração e encapsulamento que nos permitem modelar problemas de forma mais efetiva. Contudo, todas as classes que criamos continham todas as informações e comportamentos necessários para modelar determinado problema. Cada classe criada funcionava de forma independente, sem nenhuma relação com as demais.

A partir de hoje estudaremos uma nova abordagem aos problemas utilizando **conceitos relacionais** da POO. Estes são responsáveis por possibilitar a criação de classes a partir (ou com a ajuda) de outras classes.

Na POO, os principais conceitos relacionais são a herança, a associação e as interfaces.

# Herança

---

## Conceito

O conceito de herança nada mais é do que uma possibilidade de representar algo que já existe no mundo real.

Um exemplo é, quando na escola, estudamos sobre classificações biológicas: Fazemos a divisão dos seres vivos em Reino, Filo, Classe, Ordem, Família, Gênero e Espécie.

Cada divisão mais baixa herda o que for necessário da divisão superior pois a mais baixa é um subtipo da divisão acima.

Espécie herda de Gênero, que herda de Família e assim por diante

# Herança

## Conceito



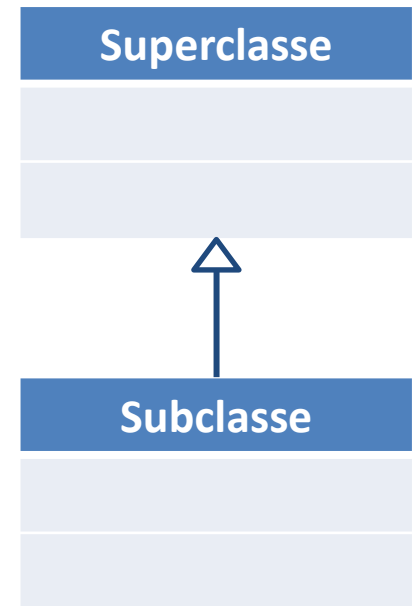
# Herança

## Definição

Na POO, herança é o 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).

Devido a isso, a subclasse consegue reaproveitar os atributos e métodos da superclasse além de poder definir seus próprios membros

Diagrama UML



# Herança

---

## Definição

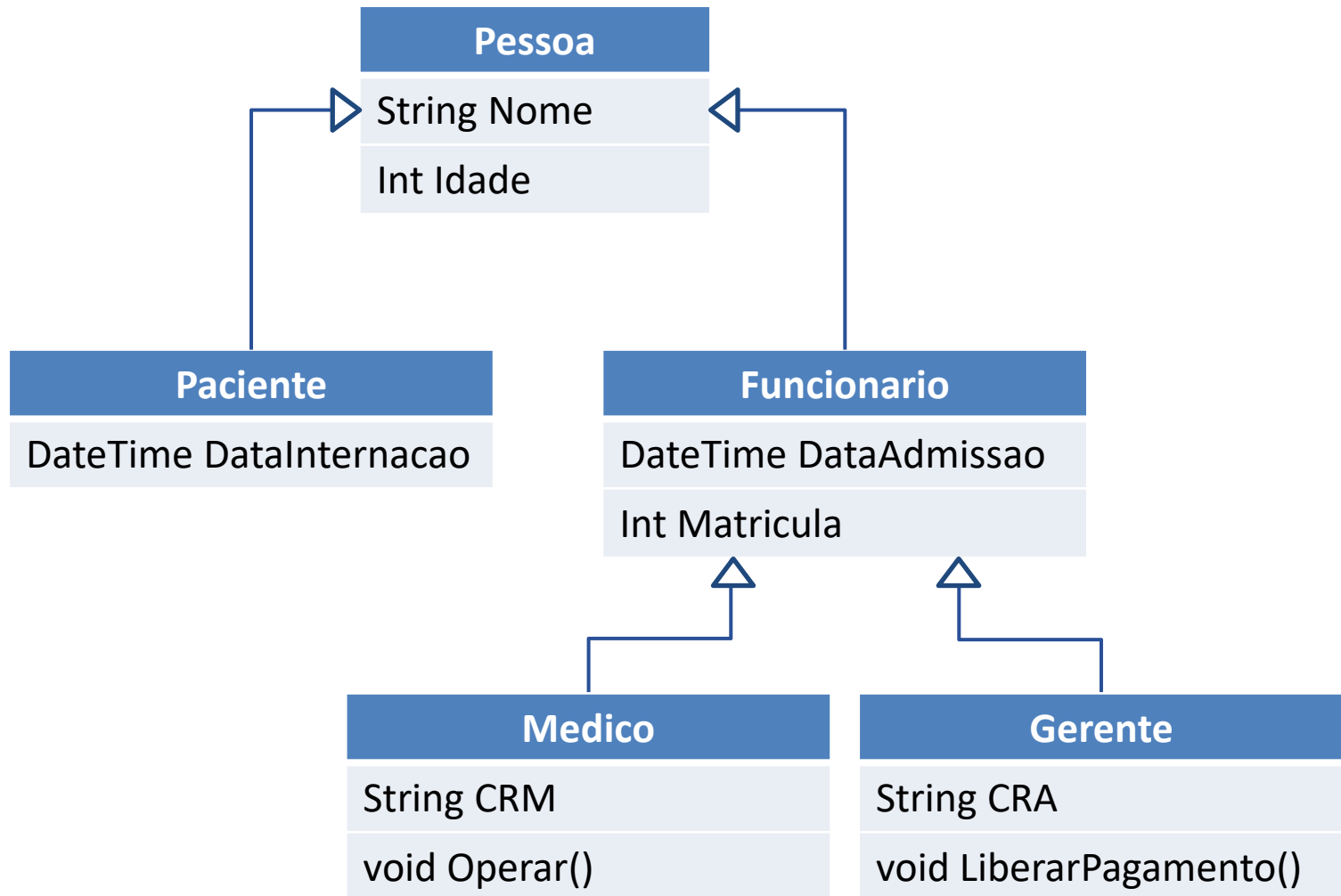
Quando utilizamos a herança, estamos dizendo que um conceito "é do tipo" de outro conceito, e esta possibilidade é vital para representar fielmente o mundo real que se está modelando.

Por exemplo, em um hospital, existem vários tipos de pessoas entre as quais podemos citar: pacientes e funcionários.

Estes últimos podem ser Médicos, Enfermeiros, Fisioterapeutas, Gerentes, entre outros.

# Herança

## Definição



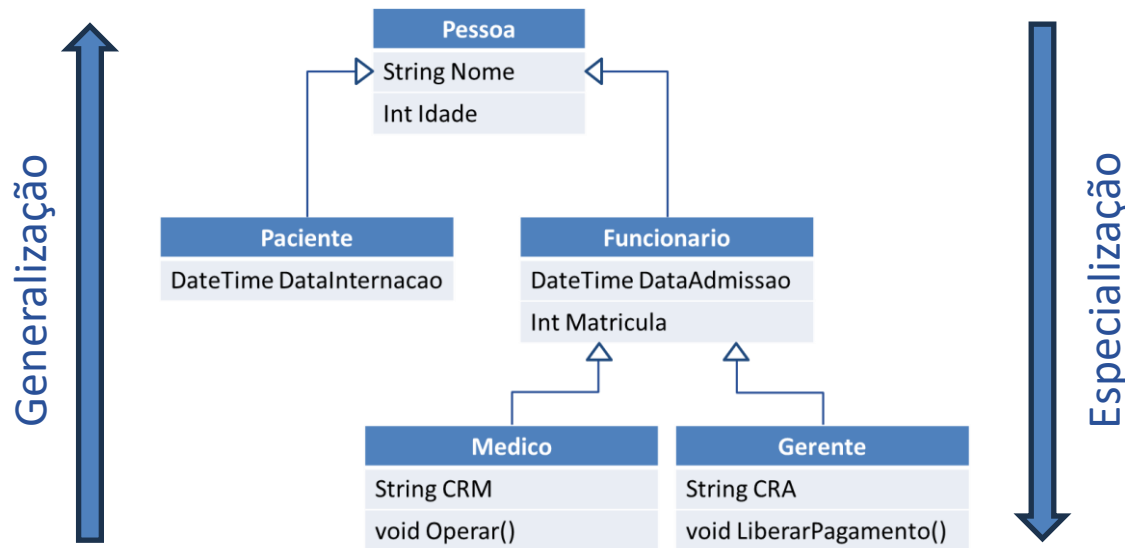


# Herança

## Generalização e especialização

O processo de definir o mais genérico nas classes bases e ir acrescentando o mais específico nas classes filhas é chamado de generalização e especialização, respectivamente.

Quanto mais acima na Hierarquia de Classe, mais genérico se torna. Quanto mais abaixo, mais específico



# Herança

---

## Vantagens

### Reutilização de Código

- Permite reutilizar conceitos e membros de outras classes
- Proporciona uma estrutura hierárquica eficiente

### Adição de Membros

- Classes derivadas podem adicionar novos membros, mantendo a consistência do código original

### Definição de Subtipos

- Facilita a criação de hierarquias de classes, organizando o código de forma lógica

### Abstração

- Promove a criação de modelos reutilizáveis e fáceis de manter

# Herança

---

## Boas práticas

A herança pode ocorrer em quantos níveis forem necessários. Contudo, uma boa quantidade de níveis é de, no máximo, 4.

Quanto mais níveis existirem, mais difícil será de entender o código pois a cada nível é gerado um distanciamento maior do conceito base.

Além disso, deve-se tomar cuidado ao decidir em que nível incluir determinado membro. Se a herança for usada só pelo reuso de código e não pelos subtipos, situações que não representam a realidade podem ocorrer.

Exemplo: Definir o atributo CRM na classe Pessoa não “quebraria” o código, mas poderia levar ao erro de atribuição de um CRM a um paciente.

# Herança

---

## Tipos

Existem dois tipos de herança: **simples** e **múltipla**

A simples ocorre quando uma subclasse tem apenas uma superclasse. Neste caso, a classe filha precisou apenas especializar e reutilizar membros de apenas um conceito.

Já a herança múltipla ocorre quando uma subclasse necessita de duas ou mais superclasses.

Exemplo: Caso se crie o cargo LiderDeEquipe no hospital e se determine que este líder pode liberar pagamentos e gerir horários de sua equipe, esta subclasse precisará dos conceitos da classe Medico e também da classe Gerente.

# Herança

---

## Herança múltipla

Observação: Em linguagens como C# e Java o conceito de herança múltipla **não é permitido**.

Isto ocorre para evitar possíveis conflitos de nomes que demandariam um compilador mais complexo capaz de resolver tais ambiguidades.

Exemplo: Classes Medico e Gerente têm atributo CargaHoraria que representa as horas que devem ser trabalhadas na função médico e gerente, respectivamente.

Ao se definir o valor do atributo CargaHoraria de um objeto do tipo LiderDeEquipe, o atributo de qual classe será mudado?

Mais informações: [https://en.wikipedia.org/wiki/Multiple\\_inheritance#The\\_diamond\\_problem](https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem)



## Exemplo 1 – Implementação da herança em C#

Para definir que uma classe é subclasse de outra, usamos “:”

```
class Animal
{
    public string Nome { get; set; }

    public Animal() { }

    public Animal(string nome) {...}

    public void Comer () {...}
}
```

Superclasse

```
class Cachorro : Animal
{
    public void Latir () {...}
}
```

→ Cachorro “extende” Animal

Subclasse

## Exemplo 2 – A palavra “*base*”


A palavra “*base*” serve para nos referirmos a superclasse

```
class Animal
{
    public string Nome { get; set; }

    public Animal(string nome)
    {
        Nome = nome;
    }
}
class Cachorro : Animal
{
    public string Raca { get; set; }

    public Cachorro(string nome, string raca) : base (nome)
    {
        Raca = raca;
    }
}
```

Chama construtor da classe base



# Herança

---

## Relembrando – Modificadores de acesso

A visibilidade dos membros de uma classe depende dos modificadores de acesso utilizados. Dentre os principais:

- *Public* → Não impõe restrições ao dado da classe, permitindo que ele seja acessado e modificado diretamente onde e quando necessário
- *Private* → Permite o acesso ao dado apenas através da própria classe (por exemplo, uma função da classe)
- *Protected* → O dado é acessível dentro de sua classe e por instâncias derivadas dessa classe (subclasses)

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



## Exercício 1

Suponha um negócio de banco que possui uma conta comum e uma conta para empresas, sendo que a conta para empresa possui todos membros da conta comum, mais um limite de empréstimo e uma operação de realizar empréstimo.

Account
- number : Integer - holder : String - balance : Double
+ withdraw(amount : Double) : void + deposit(amount : Double) : void

BusinessAccount
- number : Integer - holder : String - balance : Double - loanLimit : Double
+ withdraw(amount : Double) : void + deposit(amount : Double) : void + loan(amount : Double) : void

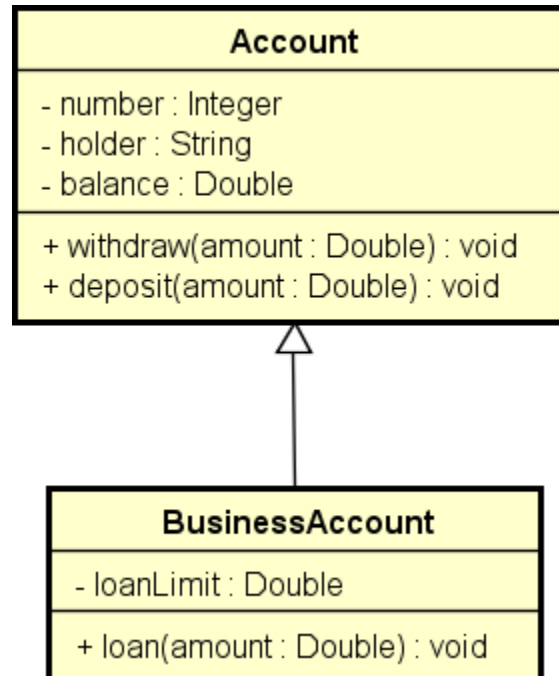
Implemente as classes e o programa principal para teste.

Obs: O saldo da conta só pode ser alterado através dos métodos das classes



## Exercício 1 – Continuação

### Diagrama UML



# Herança

---

## ***Upcasting e downcasting***

*Upcast* e *downcast* são duas operações que podem ser realizadas com objetos criados a partir de classes envolvidas em uma hierarquia de classe.

### ***Upcasting* → Promoção de uma subclasse à superclasse**

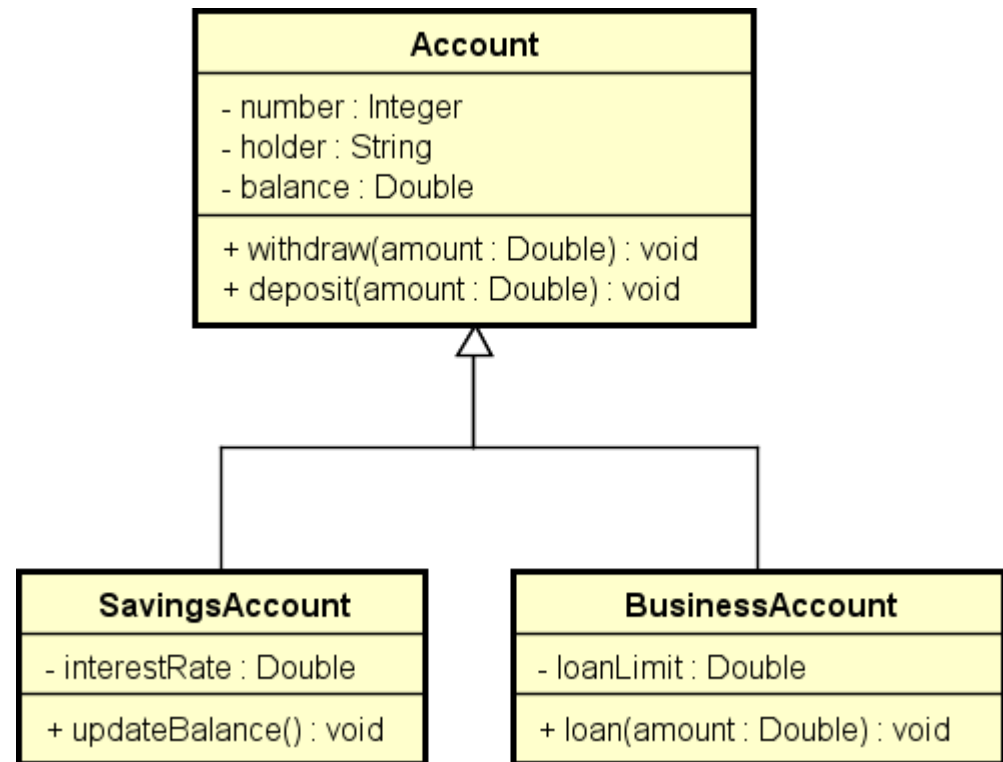
- A conversão é implícita (*just works*)
- Sempre seguro, pois um objeto derivado pode ser tratado como seu tipo base

### ***Downcasting* → Conversão de uma superclasse em subclasse**

- A conversão precisa ser explícita
- Potencialmente perigoso, pois nem sempre o objeto do tipo base é do tipo derivado

## Exemplo 3 – *Upcasting e downcasting*

Para exemplificar as operações, vamos ampliar o exercício anterior adicionando uma nova modalidade de conta do tipo poupança





## Exemplo 3 – *Upcasting e downcasting* – Continuação

// Upcasting: Criando uma nova instância de BusinessAccount diretamente como Account

```
Account acc = new BusinessAccount(1004, "Bob", 0.0, 200.0);
```

// Downcasting: Convertendo Account de volta para BusinessAccount

```
BusinessAccount bacc = (BusinessAccount)acc;
```

// Teste para verificar o tipo

```
if (acc is BusinessAccount)
{ ... }
```

```
else if (acc is SavingsAccount)
{ ... }
```



## Sobreposição

Utilizando novamente o exemplo das contas bancárias, suponha que a conta comum (e por consequência a conta empresarial) tem uma taxa de saque de 5,00 e a conta bancária tem taxa zero.

Como fazer com que o método `Withdraw` da subclasse `SavingsAccount` quebre uma regra estabelecida pela classe base?

Resposta: sobrescrevendo o método `Withdraw` na subclasse.

Para isso, precisamos utilizar no método da classe base a palavra `virtual` e na subclasse a palavra `override`



## Exemplo 4 – Sobreposição

### Classe Account

```
public virtual void Withdraw(double amount)
{
    // Realiza o saque subtraindo o valor do saldo e taxa.
    Balance -= amount + 5.00;
}
```

### Classe SavingsAccount

```
public override void Withdraw(double amount)
{
    // Realiza o saque subtraindo o valor do saldo.
    Balance -= amount;
}
```

# Herança

---

## Classes e métodos selados

Palavra chave: `sealed`

Na classe → Evita que a classe seja herdada

Nota: ainda é possível estender a funcionalidade de uma classe selada usando "*extension methods*"

```
internal sealed class SavingsAccount : Account
```

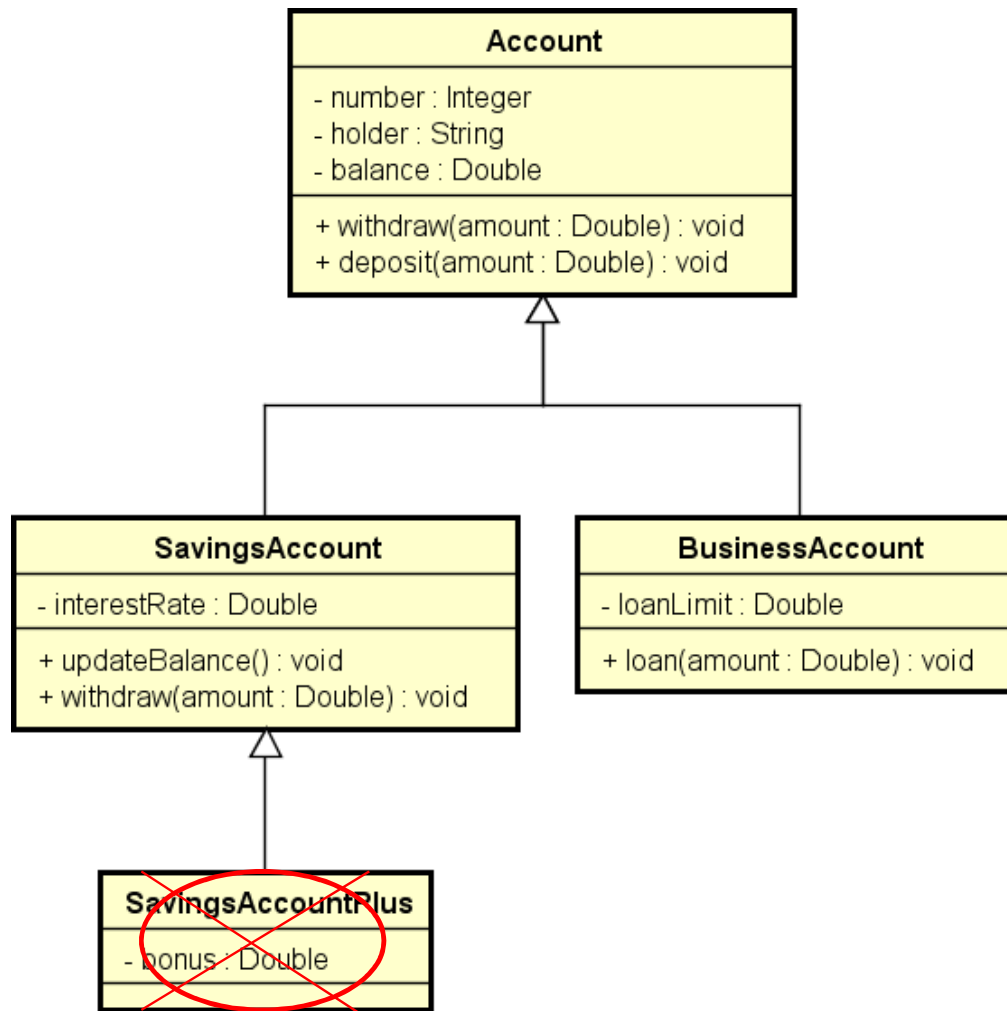
No método → Evita que um método sobreposto possa ser sobreposto novamente → Só pode ser aplicado a métodos sobrepostos

```
public sealed override void Withdraw(double amount)
```



# Herança

## Classes e métodos selados





## Exercício 2

Desenvolva um sistema de gerenciamento de funcionários que utilize os conceitos de herança. Crie uma classe **Employee** que represente um funcionário genérico. Esta classe deve ter as seguintes propriedades:

- **Id (int)**: identificador único do funcionário.
- **Name (string)**: nome do funcionário.
- **BaseSalary (double)**: salário base (todos recebem independentemente do tipo).

Crie duas classes derivadas de **Employee**: **HourContract** e **SalariedEmployee**.

**HourContract** representa um funcionário que recebe por hora trabalhada. Ela deve conter:

- **Hours (int)**: quantidade de horas trabalhadas.
- **HourValue (double)**: valor da hora de trabalho.

**SalariedEmployee** representa um funcionário que recebe um salário fixo. Ela deve conter:

- **MonthlySalary (double)**: salário mensal.

Implemente um método **Income()** em cada classe que calcule o salário do funcionário de acordo com sua forma de pagamento.

Implemente também um programa para testar as funcionalidades das suas classes.

# Revisão

---

## Próxima aula

### Enumerações e Associações

- Definição e conceito de enumerações
- Vantagens do uso de enumerações
- Definição e conceito de associação
- Tipos de associação: agregação, composição e dependência
- Exemplos

# Dúvidas?

---

[renan.duarte@gedre.ufsm.br](mailto:renan.duarte@gedre.ufsm.br)

GEDRE – Prédio 10 – CTLAB