CEUB

EDUCAÇÃO SUPERIOR

Programação Orientada a Objetos

ceub.br

CEUB
EDUCAÇÃO SUPERIOR

Programação Orientada a Objetos

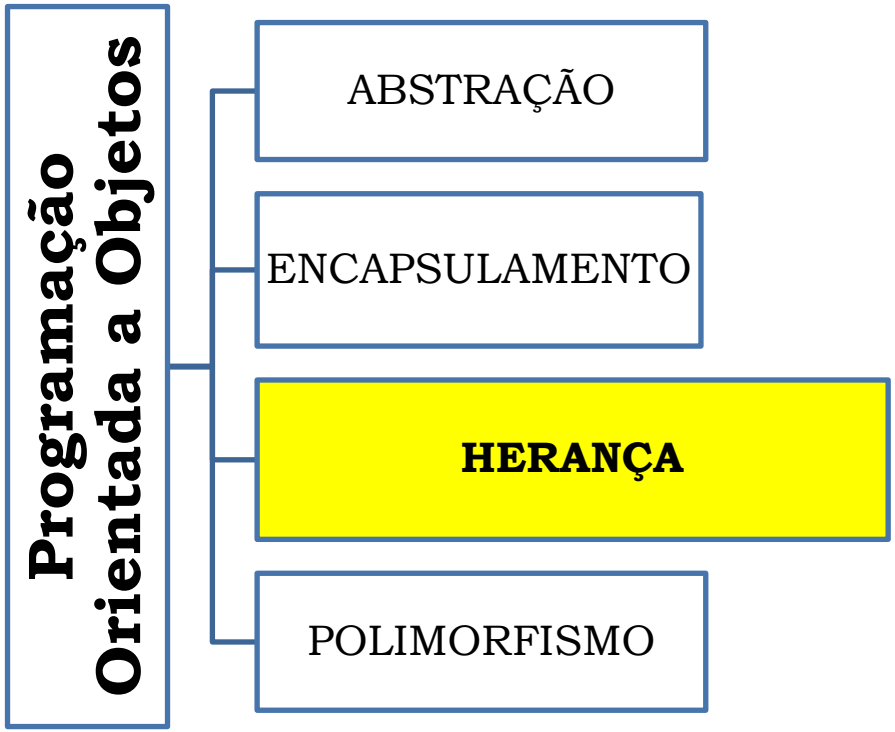
Aula 06 – Herança

Agenda

- **Conceito de Herança em POO**
- **Herança**
- **Herança em C#**

Os 4 pilares da Programação Orientada a Objetos

Para que uma linguagem possa ser enquadrada no paradigma de orientação a objetos, ela deve atender a **quatro tópicos** bastante importantes:



Os 4 pilares da Programação Orientada a Objetos

Herança

O reuso de código é uma das grandes vantagens da programação orientada a objetos. Muito disso se dá por uma questão que é conhecida como *herança*.

Essa característica otimiza a produção da aplicação em tempo e linhas de código.

A reutilização de códigos nas linguagens orientadas a objetos é uma característica que otimiza o desenvolvimento de um aplicativo tanto em economia de tempo, quanto em número de linhas de código.

Herança - Relacionamento entre objetos

- **Objetos não existem isolados**
  - **São formados por outros objetos**
  - **Objetos usam outros objetos**
  - **Um programa OO possui vários objetos que interagem entre si**
  - **Modelagem define quais objetos usamos em um programa**

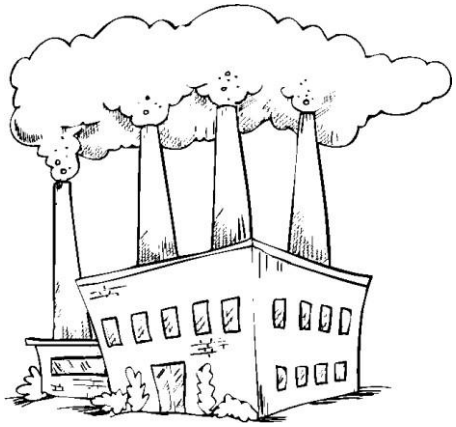
Modularidade

- Divisão dos componentes do software em unidades funcionais separadas
- Essas unidades se interagem entre si, fazendo com que o sistema funcione de forma adequada
- Vantagem:
  - Reutilização de Software

Herança

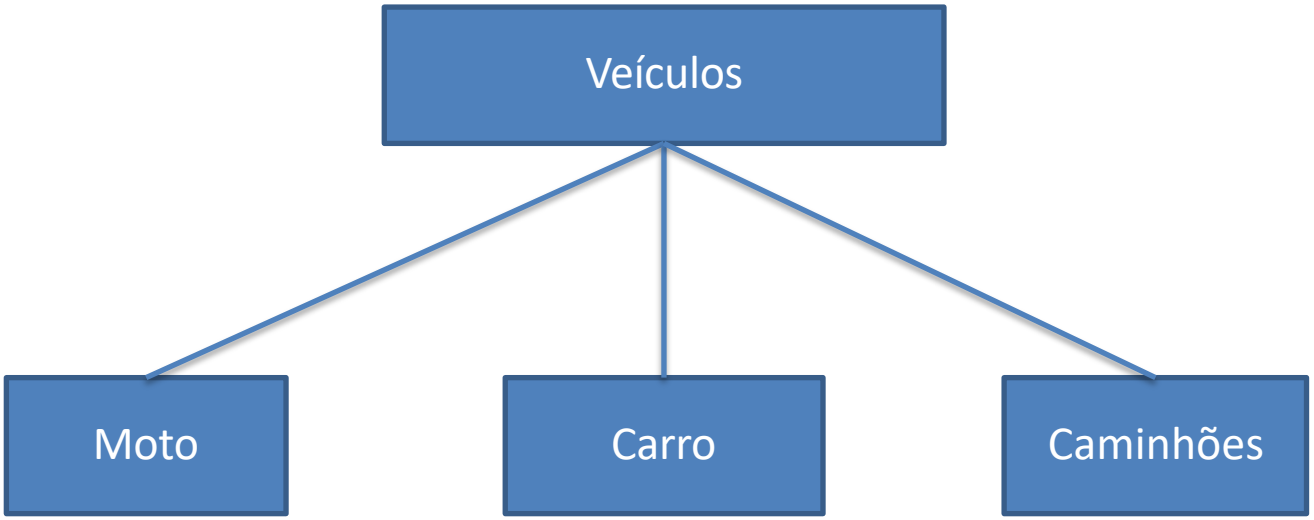
- **Dado um sistema de venda que permita o cadastro de pessoa física ou Jurídica.**
  - **Como evita a redundância?**
  - **O que eles têm em comum?**
  - **Como eles são relacionados?**

PESSOA FÍSICA
Atributos:
•Nome
• Endereco
•Telefone
•CPF



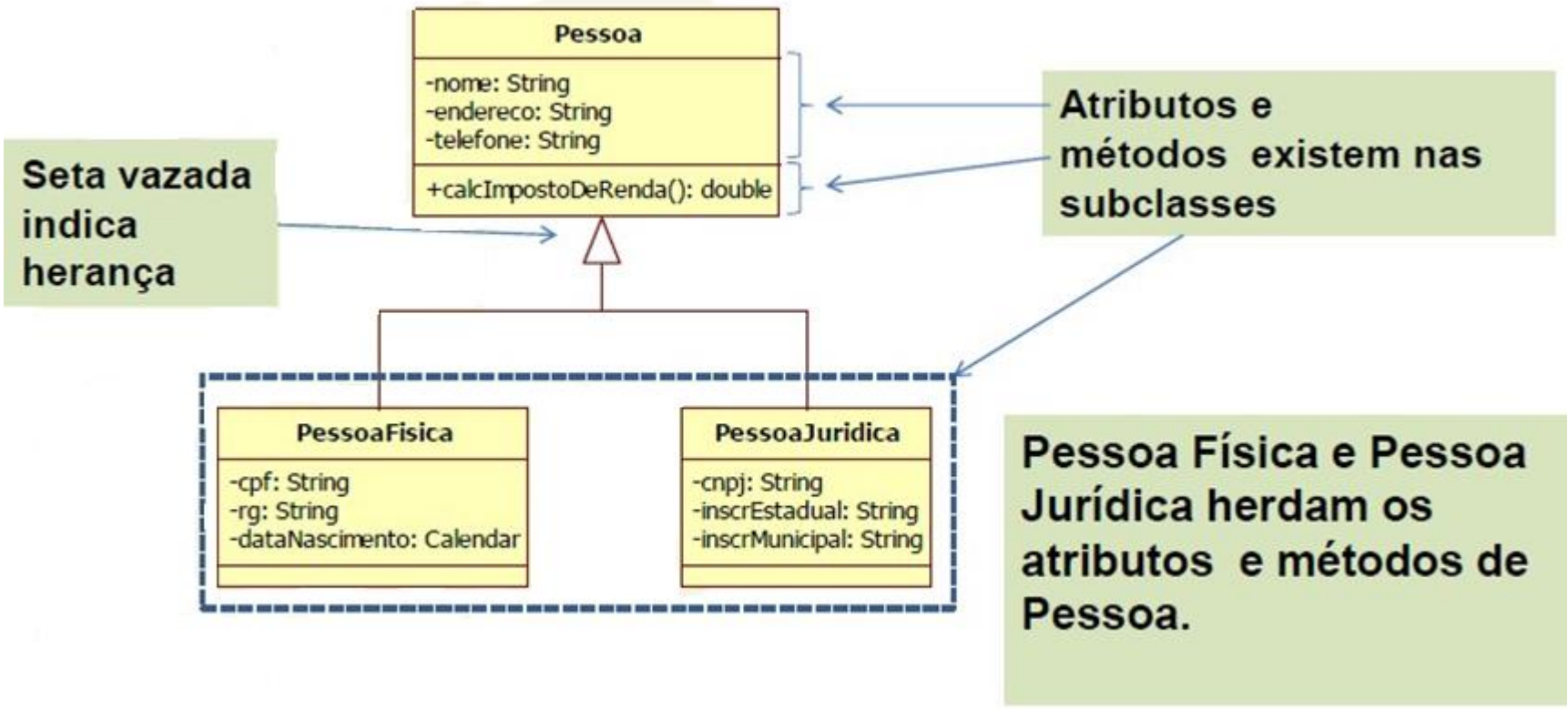
PESSOA JURÍDICA
Atributos:
•Nome
• Endereco
•Telefone
•CNPJ
•Inscrição Estadual
•Inscrição Municipal

Modularidade



Herança

- Solução: Utilização de herança



Herança em POO

- Classe genérica, classe base, superclasse, ancestral ou pai:
  - Define variáveis de instância "genéricas" e métodos
- Classe especializada, derivada, subclasse ou filha:
  - Especializa, estende ou herda os métodos "genéricos" de uma superclasse
  - Define apenas os métodos que são especializados

Herança em POO

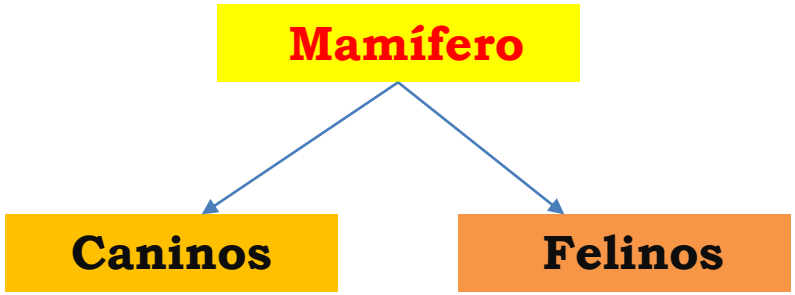
- **Herança é um conceito chave usado na orientada ao objeto para descrever uma relação entre classes**
- **Através da herança uma classe copia ou herda todas as propriedades, atributos e métodos de uma outra classe, podendo estender sua funcionalidade**
- **A herança evita a reescrita de código e especifica um relacionamento de especialização / generalização**

Herança

Por exemplo, analisemos uma classe com todas as características dos animais mamíferos.

Se tivermos uma classe Humanos e outra felinos ambas as classes herdarão as características da classe mamíferos.

Do ponto de vista de código, ao criar as classe estaremos economizando códigos e tempo de desenvolvimento.



Herança

A idéia de herança é simples, mas poderosa. Quando você quer criar uma nova classe e já existe uma classe que inclui algum código que você deseja, é possível derivar sua nova classe da classe existente.

Ao fazer isso, você pode reutilizar os atributos e métodos da classe existente sem ter que escrever (e depurar!)

Herança

Uma subclasse herda todos os membros (campos, métodos e classes aninhadas) de sua superclasse.

Os Construtores não são membros, portanto não são herdados por subclasses, mas o construtor da superclasse pode ser invocado a partir da subclasse, como o comando base().

OBS:

Na Programação orientada a objetos a herança é simples: uma classe só pode ter uma super classe. Ou seja não é possível herdar de duas classes ao mesmo tempo em C# ou em Java! Não existe a figura de Herança múltipla.

Herança - : Classe Base

Para fazermos uma classe herdar as características de uma outra, usamos a palavra reservada : logo após a definição do nome da classe.

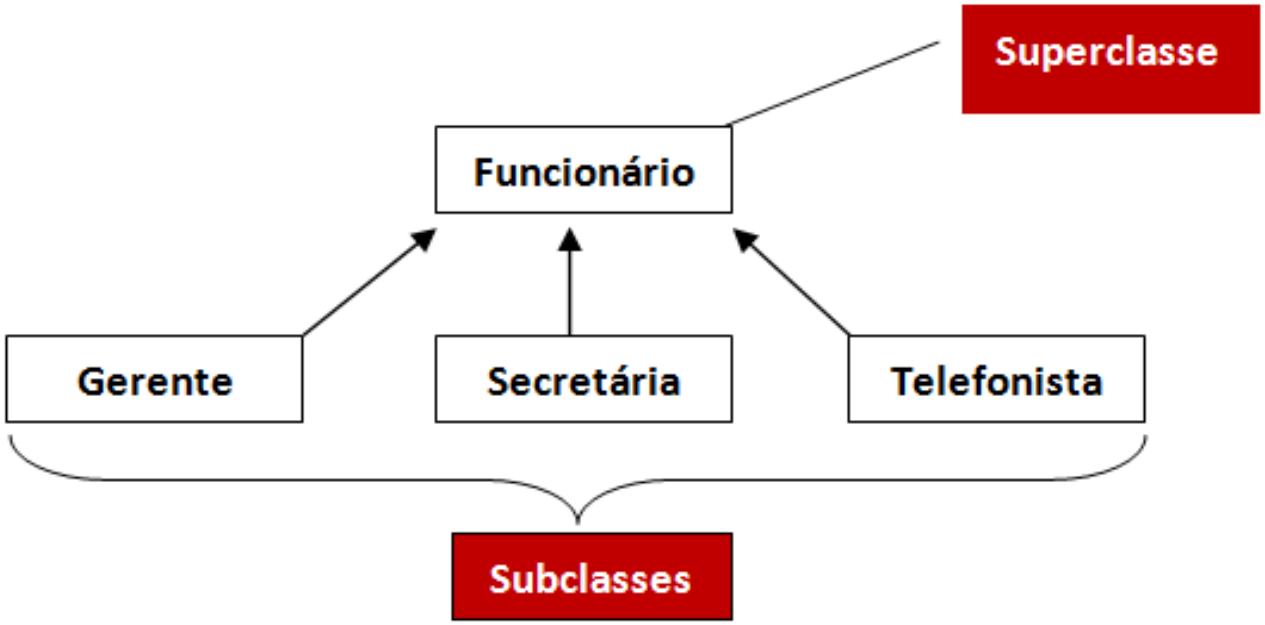
Class Motorista : Pessoa

Importante: C# permite que uma classe herde apenas as características de uma única classe, ou seja, não pode haver heranças múltiplas.

Porém, é permitido heranças em cadeias, por exemplo: se a classe Mamifero herda a classe Animal, quando fizermos a classe Cachorro herdar a classe Mamifero, a classe Cachorro também herdará as características da classe Animal.

Herança

Ex




```
class Funcionario {...}
class Gerente : Funcionario{...}
class Secretaria : Funcionario {...}
class Telefonista : Funcionario {...}
```

Herança Exemplo

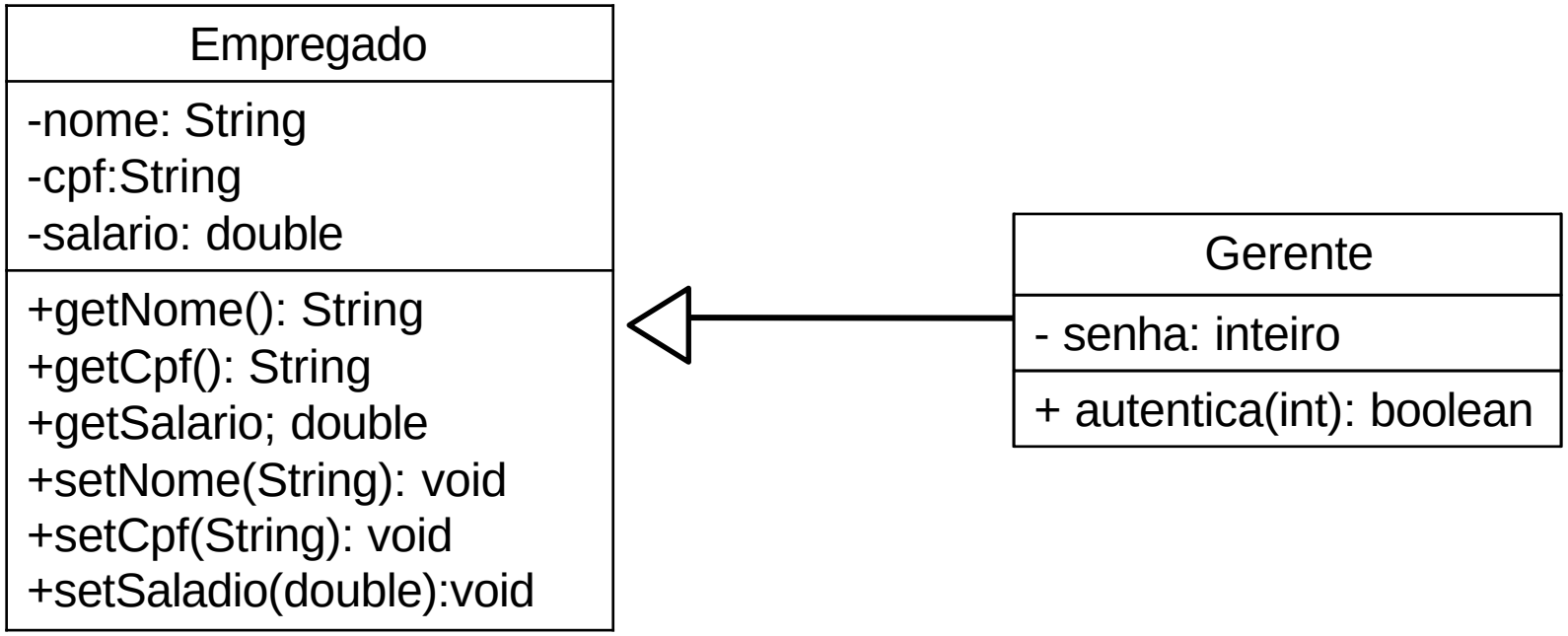
Exemplo
Vamos analisar a classe Funcionário e Gerente.

Se partirmos do pressuposto que um Gerente é uma Funcionário, podemos dizer que a Classe Gerente tem acesso aos membros da classe "pai" Empregado.

Logo para que uma classe herde de outra ela deve usar a palavra reservada : *que* informa que a classe é filha de.

HERANÇA

Exemplo de notação UML



REPETIÇÃO DE CÓDIGO

Exemplo a classe Empregado, que representa o Empregado de um banco:

```
0 referências
class Empregado
{
    private string nome;
    private string cpf;
    private double salario;

    0 referências
    public string Nome { get => nome; set => nome = value; }
    0 referências
    public string Cpf { get => cpf; set => cpf = value; }
    0 referências
    public double Salario { get => salario; set => salario = value; }
}
```

REPETIÇÃO DE CÓDIGO

Além de um funcionário comum, há também outros cargos, como os gerentes.

- **Os gerentes guardam a mesma informação que um funcionário comum, mas possuem outras informações, além de ter funcionalidades um pouco diferentes**

- **Vamos supor que, nesse banco, o gerente possui também uma senha numérica que permite o acesso ao sistema interno do banco**

REPETIÇÃO DE CÓDIGO

Classe Gerente:

```
0 referências
class Gerente
{
    private string nome;
    private string cpf;
    private double salario;

    0 referências
    public string Nome { get => nome; set => nome = value; }
    0 referências
    public string Cpf { get => cpf; set => cpf = value; }
    0 referências
    public double Salario { get => salario; set => salario = value; }
}
```

REPETIÇÃO DE CÓDIGO

Ao invés de criar duas classes diferentes, uma para Funcionario e outra para gerente, poderíamos ter deixado a classe Funcionario mais genérica, mantendo nela senha de acesso.

- **Caso o funcionário não fosse um gerente, deixaríamos este atributo vazio (não atribuiríamos valor a ele).**

Mas e em relação aos métodos?

- **A classe Gerente tem o método autentica, que não faz sentido ser acionado em um funcionário que não é gerente.**

REPETIÇÃO DE CÓDIGO

- Se tivéssemos um outro tipo de funcionário, que tem características diferentes do funcionário comum, precisaríamos criar uma outra classe, e copiar o código novamente!

- Ou ainda, se um precisássemos adicionar uma nova informação (ex: data de nascimento) para todos os funcionários?
  - Todas as classes teriam que ser alteradas para receber essa informação

- SOLUÇÃO: Centralizar as informações principais do funcionário em um único lugar!

HERANÇA

- Existe uma maneira, em C#, de relacionarmos uma classe de tal maneira que uma delas herda tudo que a outra tem.
- Isto é uma relação de classe mãe e classe filha.
- No nosso caso, gostaríamos de fazer com que o Gerente tivesse tudo que um Funcionario tem:
  - Gostaríamos que ela fosse uma extensão de
    - Empregado
- Fazemos isto através da palavra chave dois pontos :

HERANÇA

```
2 referências
class Gerente : Empregado
{
    private int senha;
    1 referência
    public int Senha { get => senha; set => senha = value; }
    0 referências
    public String autentica(int testarSenha)
    {
        if (testarSenha == Senha)
        {
            return ("Acesso Permitido");
        }
        else
        {
            return ("Acesso Negado");
        }
    }
}
```

HERANÇA

Todo momento que criarmos um objeto do tipo Gerente, este objeto possuirá também os atributos definidos na classe Empregado, pois agora um Gerente é um Empregado!

HERANÇA

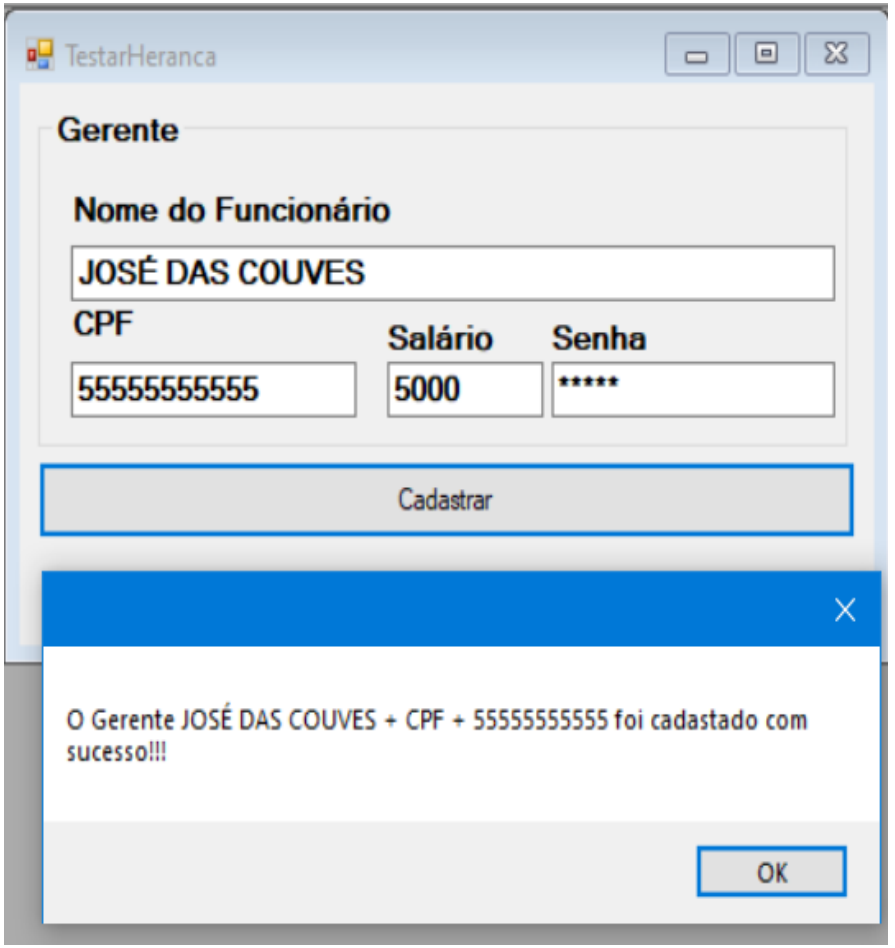
Termos utilizados:

Classes que fornecem Herança	Classes que herdam de outras
Superclasse	Subclasse
Pai	Filha
Tipo	Subtipo

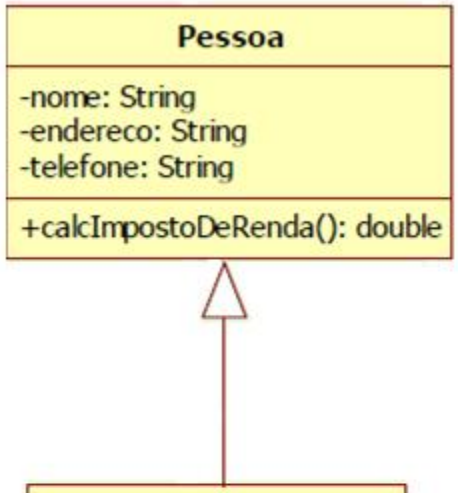
HERANÇA

Exemplo de Teste da classe:

```
4 referencias
public partial class TestarHeranca : Form
{
    1 referência
    public TestarHeranca()
    {
        InitializeComponent();
    }

    1 referência
    private void Cadastrar_Click(object sender, EventArgs e)
    {
        Gerente gerenteBanco = new Gerente();
        gerenteBanco.Nome = txtNome.Text;
        gerenteBanco.Cpf = txtCPF.Text;
        gerenteBanco.Senha = (txtSenha.Text);

        MessageBox.Show("O Gerente " + gerenteBanco.Nome +
                        " + CPF + " + gerenteBanco.Cpf +
                        " foi cadastado com sucesso!!!");
    }
}
```



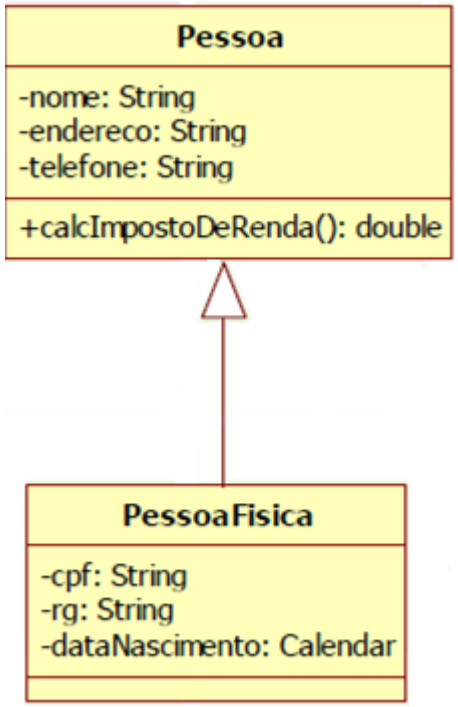
Hierarquia de Herança




```
class Pessoa {
        private String nome, endereco;
        private String telefone;
        public String getNome(){
            return nome;
        }
        public void setNome(String
nome) {
            this.nome = nome;
        }
    }
    class PessoaFisica : Pessoa {
        private String cpf, rg;
        private DateTime dtNascimento;
        public String getCpf() {
            return cpf;
        }
        public void setCpf(String cpf){
            this.cpf = cpf;
        }
    }
```

Usamos : para indicar herança em C#

Hierarquia de Herança



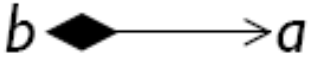
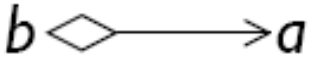
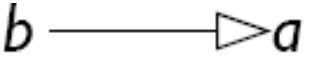

```
static void Main(string[] args){  Pessoa
   p = new Pessoa();  PessoaFisica pf = new
   PessoaFisica();  p.setNome("João");
   pf.setNome("José");
   pf.setCpf("032...");
   Console.WriteLine(p.getNome());
   Console.WriteLine(pf.getNome());
   Console.WriteLine(pf.getCpf());
}
```

Métodos herdados da classe Pessoa

Todas as características existentes na classe Pessoa também existem na classe PessoaFisica

Herança - Relacionamento entre objetos

São 04 as suas principais formas de relacionamento entre objetos:

- *Composição:* *a "é parte essencial de" b* 
- *Agregação:* *a "é parte de" b* 
- *Associação:* *a "é usado por" b* 
- *Herança:* *b "é" a (substituição pura)* 

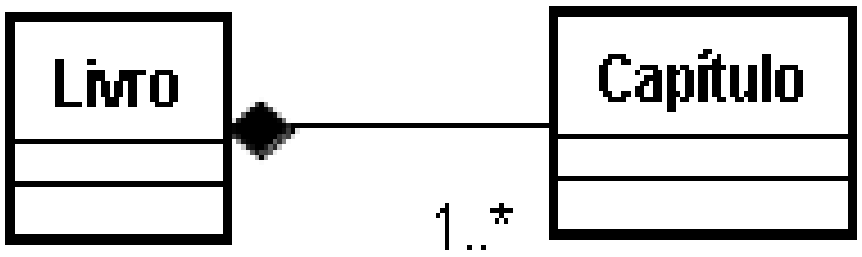
  ou *b "é um tipo de" a (substituição útil, extensão)*

Relacionamento entre objetos

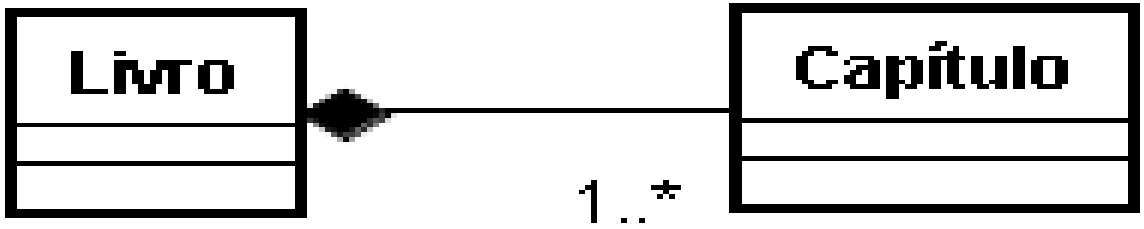
- Objetos possuem relacionamentos
  - Composição
    - Um objeto pode ser formado por outros objetos
    - Casa, livro, jardim, agenda de contatos, etc
  - Agregação
    - Um objeto pode conter outros objetos
    - Carro (motor, pneu, porta)
  - Associação
    - Objetos podem usar outros objetos
    - Trem usa estrada de ferro

Composição

- **Um livro é composto de capítulos**
  - **Capítulo é parte essencial de livro**
    - **Se não existir capítulo, não existe livro**
    - **Capítulo não existe fora de livro**
- **Linha com losângulo preenchido na classe "dominante"**
  - **Livro é composto de 1 ou mais capítulos**



Composição



Os atributos são derivados dos relacionamentos. Não existem no diagrama

```
public class Livro {

  private Capitulo[] capitulos;

  public Livro(int qtdCapitulo){
     capitulos = new Capitulo[qtdCapitulo];
  }
}
```

```
public class Capitulo {

  private Livro livro;

  /* Definição da classe Capítulo */
```

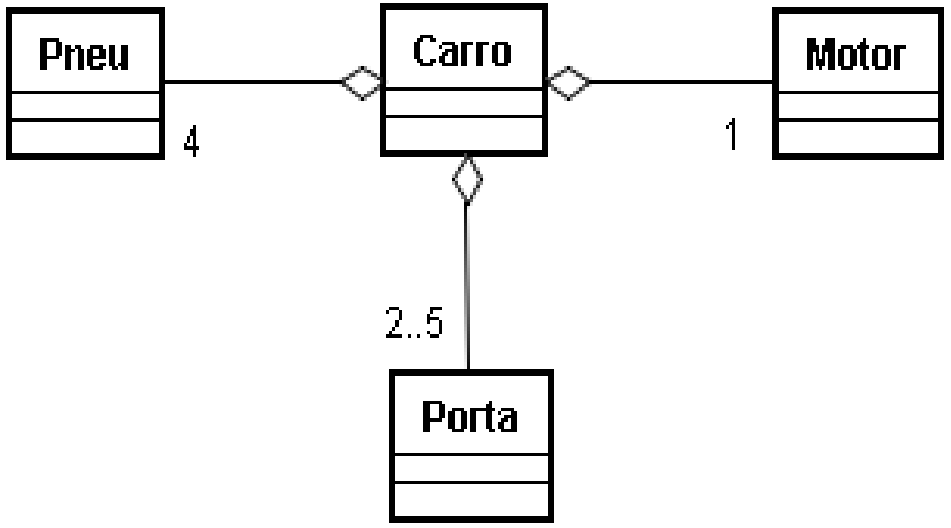
Referência pode ou não ser bidirecional. Capitulo não precisa ter o atributo livro

Agregação

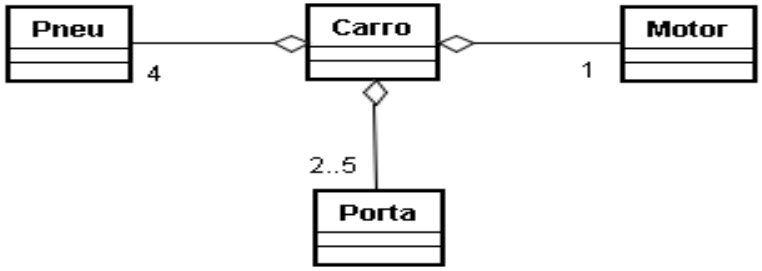
Carro possui Pneu, Motor e Porta

- Não são partes essenciais do carro
  - Retirando as portas um carro continua sendo um carro
  - Pneus/portas existem como objetos independentes

Linha com losângulo vazio na classe "dominante"



Agregação




```
public class Motor {
  /* ... */
}
```

```
public class Pneu{
  /* ... */
}
```

```
public class Porta{
  /* ... */
}
```

ATENÇÃO
Pode ser implementado de mais de uma forma

```
public class Carro {

  private Motor motor; private Porta portas[]; private Pneu pneus[];
  /* ... */
}
```

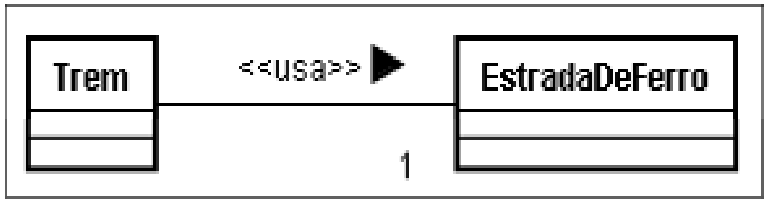
```
public class Carro {

  private Motor motor; private Porta portas[];
  private Pneu p1, p2, p3, p4;
  /* ... */
}
```

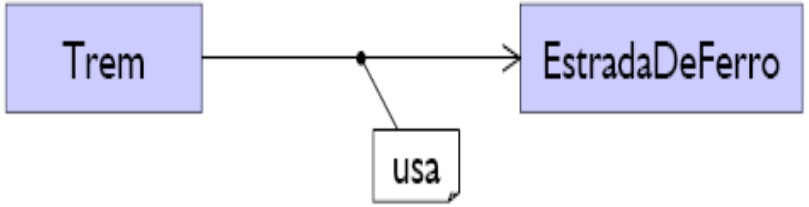
Associação

Criação de uma nova classe fazendo uso de outras usando classes (acoplamento menor).

Objetos que usam outros objetos

- **Podem ser implementados como atributos**






```
class Trem
{
    EstradaDeFerro estrada = new EstradaDeFerro();
    ...
}
```

CEUB
EDUCAÇÃO SUPERIOR

OBS: Nomes

Pode-se especificar o nome do atributo

- **Obrigar existência do atributo**
- **Carro tem um atributo privado motor do tipo Motor**

```
public class Carro {
        private Motor motor;
        /* ... */
}
```

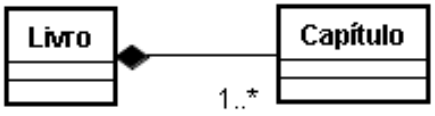
OBS 2: Nomes

Coleções

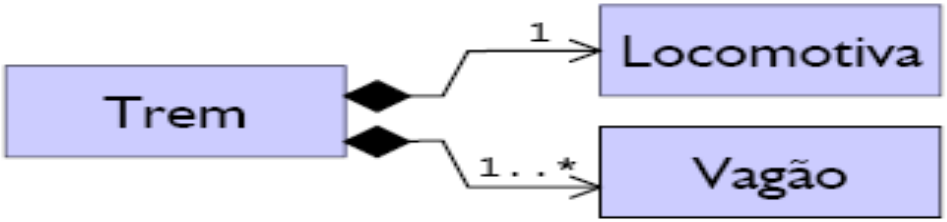
– Atributos com multiplicidade * podem ser implementados de mais de uma forma Array é uma delas.

Ex.

```
public class Livro {  private
    Capitulo cap[];
}
```



Ex2.




```
import java.util.*;
class Trem
{
    Locomotiva locomotiva = new Locomotiva();
    Collection<Vagao> vagao = new ArrayList<Vagao>();
    ...
}
```

Subtipos

- Uma classe que herda de outra é um subtipo
  - Herança representa relacionamento "É UM"
    - Carro "É UM" veículo;
    - Pessoa Física "É UMA" Pessoa;
    - Gato "É UM" Animal.
- Podemos ter uma variável do tipo Pessoa que referencia um objeto do tipo PessoaFisica

Subtipos

- **Uma referência a Pessoa sempre poderá apontar para uma instância de PessoaFisica, porque PessoaFiscia É UMA Pessoa;**

- **O que torna possível a superclasse referenciar uma instância da subclasse é a certeza de que a subclasse pode fazer tudo que a superclasse pode fazer.**

Subtipos

- **A sintaxe para declarar uma classe que herda de outra em C#**
  é:
  - **class DerivedClass : BaseClass {**
  - **…**
  - **}**
- **Sub-classes podem ser normalmente classes base para outras**
  heranças:
  - **class DerivedSubClass : DerivedClass {**
  - **…**
  - **}**
- **As classes herdam apenas de uma classe base**

Subtipos

Os construtores da classe ancestral podem (e devem) ser chamados pelo construtor da classe derivada com a palavra reservada *base*

```
class Empregado
{
    private string nome;
    protected double salarioFixo;
    1 referência
    public Empregado(string aNome, double aSalarioFixo)
    {
        nome = aNome;
        salarioFixo = aSalarioFixo;
    }
}
```

```
class Gerente : Empregado
{
    private double gratificacao;
    0 referências
    public Gerente(string aNome, double aSalarioFixo, double
    aGratificacao) : base(aNome, aSalarioFixo)
    {
        gratificacao = aGratificacao;
    }
}
```

CEUB
EDUCAÇÃO SUPERIOR

System.Object

- **A classe System.Object é a classe ancestral de qualquer classe em C#, mesmo que não declarada**

```
class Empregado { ...
}
class Empregado : System.Object { ...
}
```

- **Métodos herdados de System.Object**
  - **Equals – Testa se dois objetos são iguais**
  - **GetHashCode – Retorna o código de hash para o objeto**
  - **GetType – Retorna informação sobre a classe do objeto**
  - **ToString – Retorna o objeto como string**

Herança Exemplo

Exemplo
Vamos analisar a classe Pessoa e Motorista.

Se partirmos do pressuposto que um motorista é uma pessoa, podemos dizer que a Classe Motorista tem acesso aos membros da classe "pai" Pessoa.

Logo para que uma classe herde de outra ela deve usar a palavra reservada *:* *que* informa que a classe é filha de.

Já palavra reservada *base* chama o construtor da classe pai dentro do próprio construtor de classe

CONSTRUTOR

É da responsabilidade da subclasse inicializar os atributos definidos na sua classe, assim como os atributos que herda das suas superclasses.

O construtor da subclasse pode delegar a inicialização dos atributos herdados para a superclasse, chamando, implícita ou explicitamente, o construtor da superclasse.

Um construtor da subclasse pode fazer uma chamada explícita de um construtor da superclasse através do base()

CONSTRUTOR

Se o construtor da superclasse tiver N parâmetros, estes devem ser passados na chamada explícita:
base(param1,...,paramN).

Pode se chamar outro construtor da classe usando o this(parametros).

Se nenhum construtor da superclasse é chamado, ou se nenhum construtor da classe é chamado, como primeira instrução do construtor, o construtor sem argumentos da superclasse é implicitamente chamado antes de qualquer instrução no construtor.

Se a superclasse não tiver um construtor sem argumentos, é necessário chamar explicitamente um construtor da superclasse.

CONSTRUTOR

Pode se chamar outro construtor da classe usando o this(parametros).

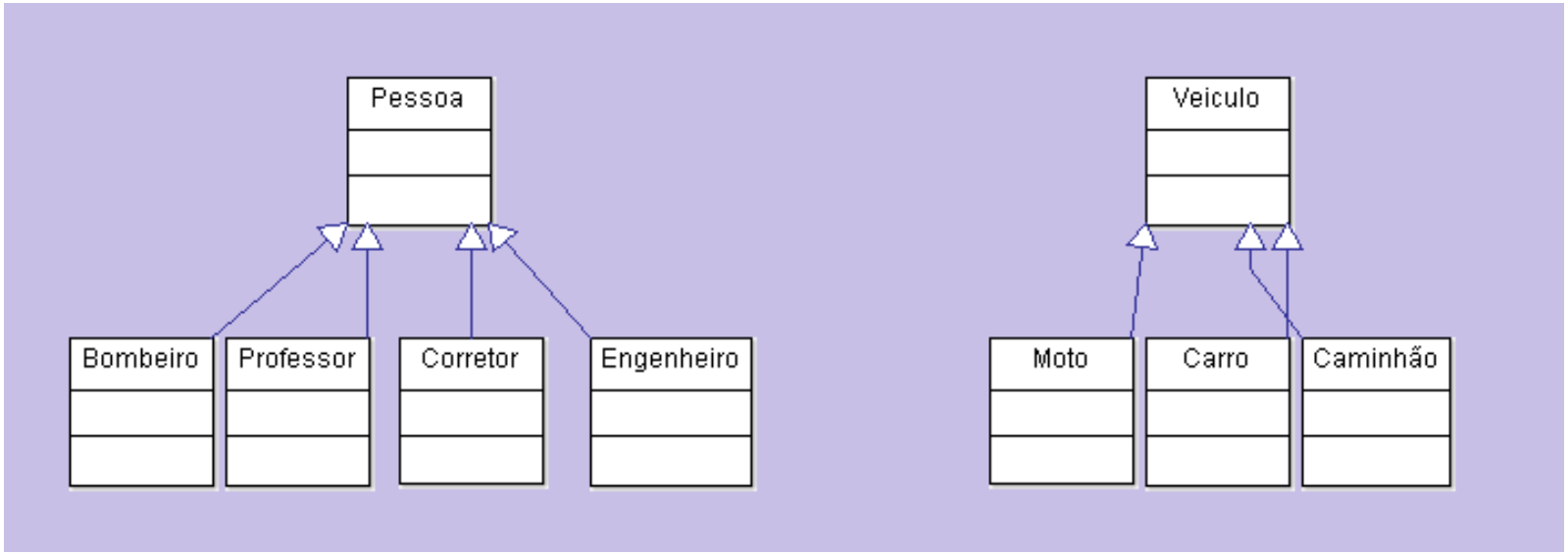
Se nenhum construtor da superclasse é chamado, ou se nenhum construtor da classe é chamado, como primeira instrução do construtor, o construtor sem argumentos da superclasse é implicitamente chamado antes de qualquer instrução no construtor.

Se a superclasse não tiver um construtor sem argumentos, é necessário chamar explicitamente um construtor da superclasse.

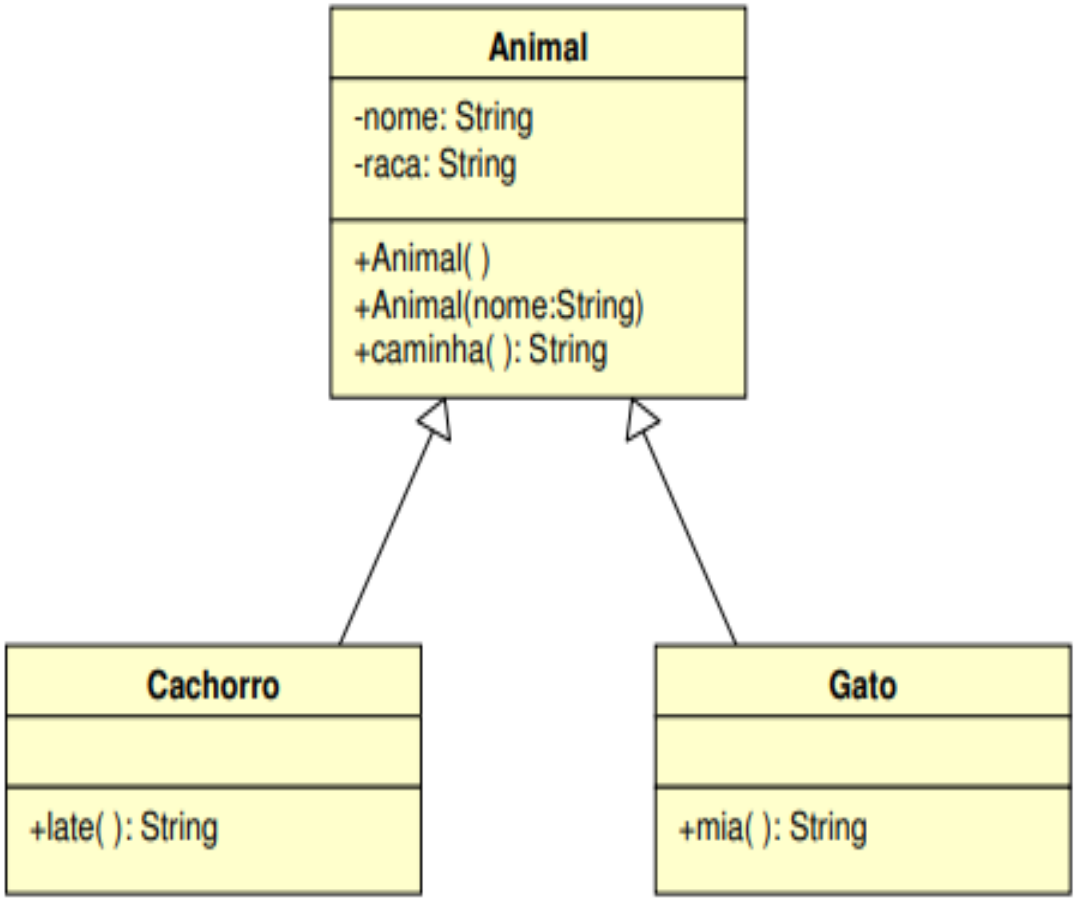
EXERCÍCIO

Dado os diagramas de classes abaixo implementar:

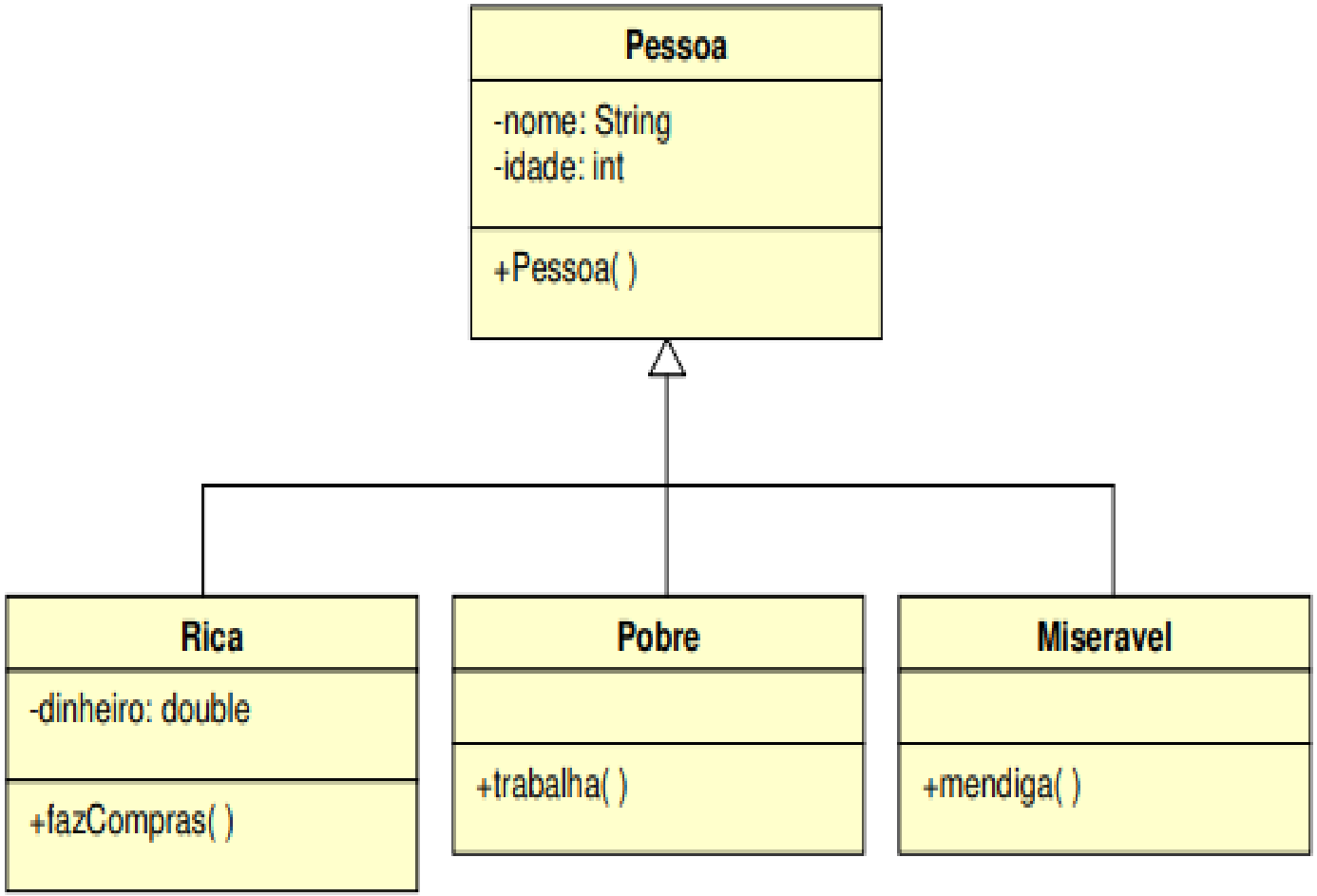
- **As classes definindo atributos (abstração) para as mesmas e fazendo o encapsulamento**
- **Implementar para cada classe um construtor com parâmetros e o default;**
- **Implementar a herança conforme a figura abaixo;**
- **Instanciar cada uma das classes informando seus atributos na tela com System.out.println('...')**



2 - Implemente os diagramas de classe abaixo:



2 - Implemente os diagramas de classe abaixo:



REFERÊNCIAS

http://www.hardware.com.br/artigos/programacao-orientada-objetos/

http://www.fontes.pro.br/educacional/materialpag inas/C#/arquivos/jdbc/jdbc.php

http://www.dm.ufscar.br/~waldeck/curso/C#

PORTAL EDUCAÇÃO - Cursos Online : Mais de 900 cursos online com certificado http://www.portaleducacao.com.br/informatica/ar tigos/7852/moderadores-de-acesso#ixzz2AAmxO3JD

http://www.slideshare.net/regispires/C#-08- modificadores-acesso-e-membros-de-classe- presentation

https://www.devmedia.com.br/abstracao-encapsulamento-e-heranca-pilares-da-poo-em-C#/26366

Referências:

Flanagan, D. (2012). JavaScript: O Guia Definitivo (6ª ed.). Bookman.

https://developer.mozilla.org/en-US/docs/Web/JavaScript