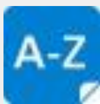
UNIDADE 4 – Reuso de Classes - Herança

Objetivos da Unidade*:

- **Conhecer o conceito de herança e aprender a implementar esse conceito em Java.**
- **Aprender a utilização da herança pela aplicação dos conceitos de reuso de classes.**

4.1 Conceito de herança

Uma das vantagens das linguagens orientadas a objeto é a possibilidade de se reutilizar código. Mas, quando falamos em reutilização de código, precisamos de recursos muito mais poderosos do que simplesmente copiar e alterar o código.



Herança
É um mecanismo que permite que uma classe herde todo o comportamento e os atributos de outra classe (CADENHEAD; LEMAY, 2005, p. 13) Em uma herança, a classe da qual outras classes herdam é chamada de classe pai, classe base ou superclasse. Já a classe que herda de uma outra é chamada de classe filha, classe derivada ou subclasse

Um dos conceitos de orientação a objetos que possibilita a reutilização de código é o conceito de herança. Pelo conceito de herança é possível criar uma nova classe a partir de outra classe já existente.

Para ilustrar o conceito de herança vamos criar uma classe para representar as contas especiais de um banco. Em nosso exemplo, uma conta especial é um tipo de conta que permite que o cliente efetue saques acima de seu saldo até um limite, ou seja, permite que o cliente fique com saldo negativo até um dado limite. Assim, criaremos uma classe *ContaEspecial* que herdará da classe Conta que criamos em aulas anteriores.

Adotaremos essa estratégia, já que uma *ContaEspecial* é um tipo de Conta que tem, além de todos os atributos comuns a todas as contas, o atributo limite. Sendo assim, deve-se utilizar a palavra reservada **extends** para que *ContaEspecial* herde de Conta suas características. A Figura 4.1 exibe o código da classe da classe *ContaEspecial*. Note na definição da classe a utilização da palavra *extends*.

```
public class ContaEspecial extends Conta {
    private double limite;

    public double getLimite() {
        return limite;
    }

    public void setLimite(double limite) {
        this.limite = limite;
    }

}
```

Figura 27: Código da classe *ContaEspecial* que herda da classe Conta

Nesse caso dizemos que ContaEspecial é uma **subclasse** ou **classe filha** de Conta. Podemos também dizer que Conta é **ancestral** ou **classe pai** de ContaEspecial. Note que ContaEspecial deine um tipo mais especializado de conta. Assim, ao mecanismo de criar novas classes herdando de outras é dado o nome de **especialização**.

Agora suponha que tenhamos um outro tipo de conta: a *ContaPoupanca*. A *ContaPoupanca* tem tudo o que a *Conta* tem com um método a mais que permite atribuir um reajuste percentual ao saldo. Agora teríamos duas classes herdando da classe Conta Nesse contexto podemos dizer que a classe *Conta* generaliza os conceitos de *ContaEspecial* e *ContaPoupanca*.

A Figura 28 exibe o código da classe *ContaPoupanca*. Note que ela herda as características da classe *Conta* e apenas implementa um novo método: *reajustar*.

```
public class ContaPoupanca extends Conta{
    public void rejustar(double percentual){
        double saldoAtual = this.getSaldo();//Obtenho o saldo atual da conta
        double reajuste = saldoAtual * percentual;//Calculo o reajuste
        this.depositar(reajuste);//deposito o reajuste na conta;
    }
}
```

Figura 28: Código da classe *ContaPoupanca* que herda da classe Conta

Quando visualizamos uma hierarquia partindo da classe pai para ilhas, dizemos que houve uma **especialização** da superclasse. Quando visualizamos partindo das classes ilhas para as classes ancestrais, dizemos que houve uma **generalização** das **subclasses**.

Após definir as classes *ContaPoupanca* e *ContaEspecial*, conforme ilustrado nas Figuras 27 e 28, o NetBeans nos sinalizará com erros nas duas classes pois, na aula anterior

definimos construtores para a classe *Conta* que é ancestral das duas que criamos agora e não deffinimos construtores para as duas subclasses que acabamos de gerar.

Caso você não tenha deinido um construtor em sua superclasse, não será obrigado a definir construtores para as subclasses, pois Java utilizará o construtor padrão para a superclasse e para as subclasses. Porém, caso haja algum construtor deinido na superclasse, obrigatoriamente você precisará criar ao menos um construtor para cada subclasse. Vale ressaltar que os construtores das subclasses utilizarão os construtores das superclasses pelo uso da palavra reservada *super*.

A Figura 29 exibe o construtor criado para a classe *ContaEspecial* enquanto a Figura 30 exibe o construtor da classe *ContaPoupanca*.

```
public ContaEspecial(int numero, String nome_titular, double limite) {
    super(numero, nome_titular);
    this.limite = limite;
}
```

Figura 29: Construtor da classe *ContaEspecial*

```
public ContaPoupanca(int numero, String nome_titular){
    super(numero,nome_titular);
}
```

Figura 30: Construtor da classe *ContaPoupanca*

Note que o construtor da classe ContaEspecial recebe como parâmetros o número, o nome do titular e o limite. Então, pelo uso da palavra super, esse construtor “chama” o construtor da classe Conta (que criamos em aula anterior) repassando o número e o nome do titular e, depois, atribui ao limite o valor recebido como parâmetro. Já o construtor da classe ContaPoupanca apenas se utiliza do construtor da superclasse, pois ele não recebe atributos além dos já tratados pela superclasse.

4.2 Visibilidade de Classes e pacotes

Quando estudamos **encapsulamento** aprendemos que devemos preferencialmente manter os atributos com nível de acesso privado (*private*) de forma que para acessá-los outras classes precisem utilizar métodos.

• ***public*** (público): indica que o método ou o atributo são acessíveis por qualquer classe, ou seja, que podem ser usados por qualquer classe, independentemente de estarem no mesmo pacote ou estarem na mesma hierarquia;

• ***private*** (privado): indica que o método ou o atributo são acessíveis apenas pela própria classe, ou seja, só podem ser utilizados por métodos da própria classe;

• ***protected*** (protegido): indica que o atributo ou o método são acessíveis pela própria classe, por classes do mesmo pacote ou classes da mesma hierarquia (estudaremos hierarquia de classes quando tratarmos de herança).

Mas, vimos também que há um nível de acesso protegido (*protected*) que faz com que o atributo se comporte como público para classes da mesma hierarquia ou do mesmo pacote e como privado para as demais classes. Definir alguns atributos da superclasse como *protected* pode trazer algumas facilidades para implementar métodos das subclasses.

Por exemplo, note que na implementação do método *reajustar* da classe *ContaPoupanca* apresentado na Figura 4.2, tivemos de obter o saldo, calcular o reajuste e depois depositar esse reajuste na conta para que fosse somado ao *saldo*, pois o atributo *saldo* foi definido na superclasse *Conta* como privado, não permitindo que o alterássemos diretamente de dentro da classe *ContaPoupanca*.

A Figura 31 apresenta uma nova implementação para o método *reajustar* da classe *ContaPoupanca*, considerando que o atributo *saldo* teve sua definição alterada para protegido (*protected*).

```
public class ContaPoupanca extends Conta {

    public void rejustar(double percentual) {
        //Recalculo o saldo acessando diretamente o atributo
        saldo = saldo + saldo * percentual;
    }
}
```

Figura 31: Nova implementação do método *reajustar* considerando o atributo saldo como protected

Note que na nova implementação podemos acessar diretamente o atributo, o que a torna mais simples.

Apesar de potencialmente facilitar a implementação de métodos nas subclasses, a utilização de atributos protegidos é perigosa, pois o atributo ficará acessível a todas as classes que estejam no mesmo pacote e não somente às subclasses. Assim, pense bastante sobre as vantagens e desvantagens antes de se decidir por deinir um atributo como *protected.*

4.2 Terminologia

Abaixo um exemplo para melhor entendimento da terminologia do conceito de herança desenvolvida na Unidade IV.

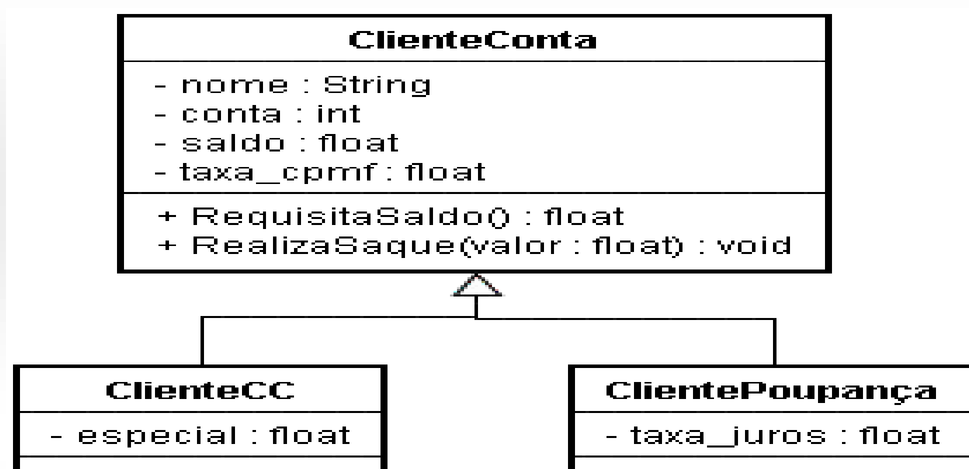


Figura 32: Diagrama de classes utilizando herança

- A classe ClienteConta é chamada de classe mãe, superclasse das classe ClienteCC e ClientePoupança;
- As classes ClienteCC e ClientePoupança são chamadas de subclasses ou classes filhas da classe ClienteConta;
- As classes ClienteCC e ClientePoupança são especializações da classe ClienteConta;
- A classe ClienteConta é uma generalização das classes ClienteCC e ClientePoupança;

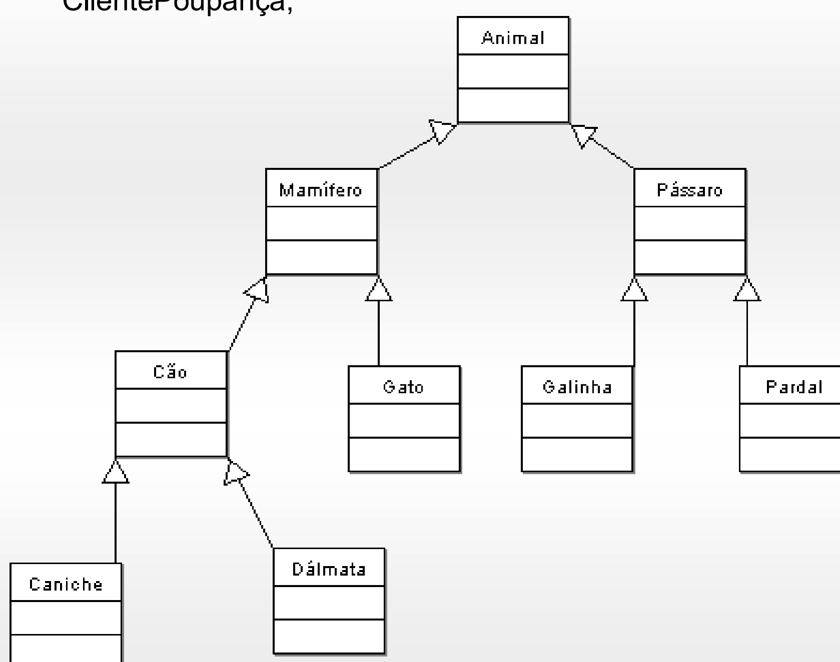


Figura 33: Herança – classes filhas devem ser do mesmo tipo das superclasses

Resumo e exemplos

O conceito de herança nos permite criar uma classe a partir de outra. Assim, quando temos um conjunto de classes com características comuns, utilizamos o conceito de herança para agrupar essas características em vez de repetirmos suas implementações várias vezes. Segue pontos principais a serem revisados:

Herança.

a. Inspiração no mundo real

- Pais transmitem aos filhos suas características e comportamento.

b. Permite criar classes que aproveitam atributos e métodos de classes existentes

- Objetivo de uso: reutilização de código com flexibilidade

c. Exemplo de diagrama e ligação entre classes.

- Classes: Estudante e Professor herdam de Pessoa.

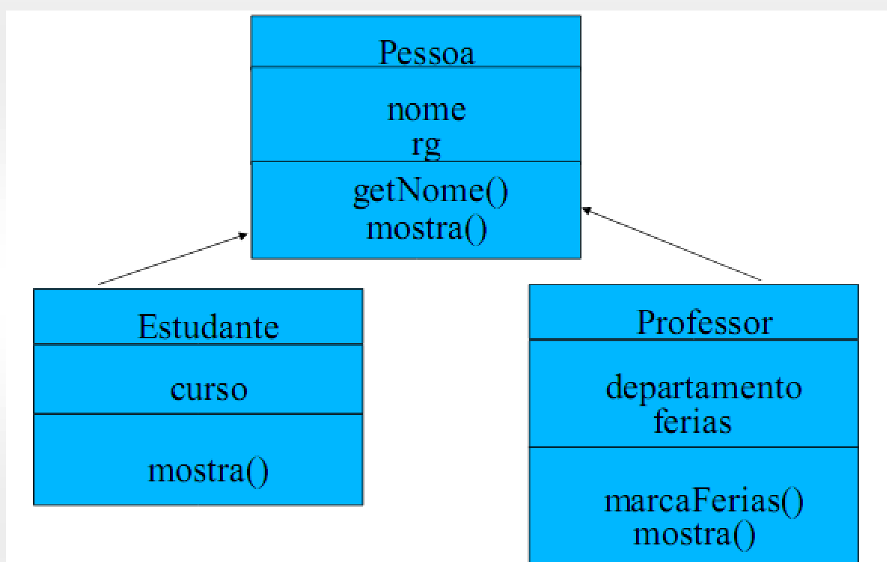


Figura 34: Estudante e Professor herdam de Pessoa

d. Terminologia da herança.

- Classes: Super-classe, sub-classe, classe derivada, classe-mãe ou classe-pai, são termos utilizados para designar herança entre classes.

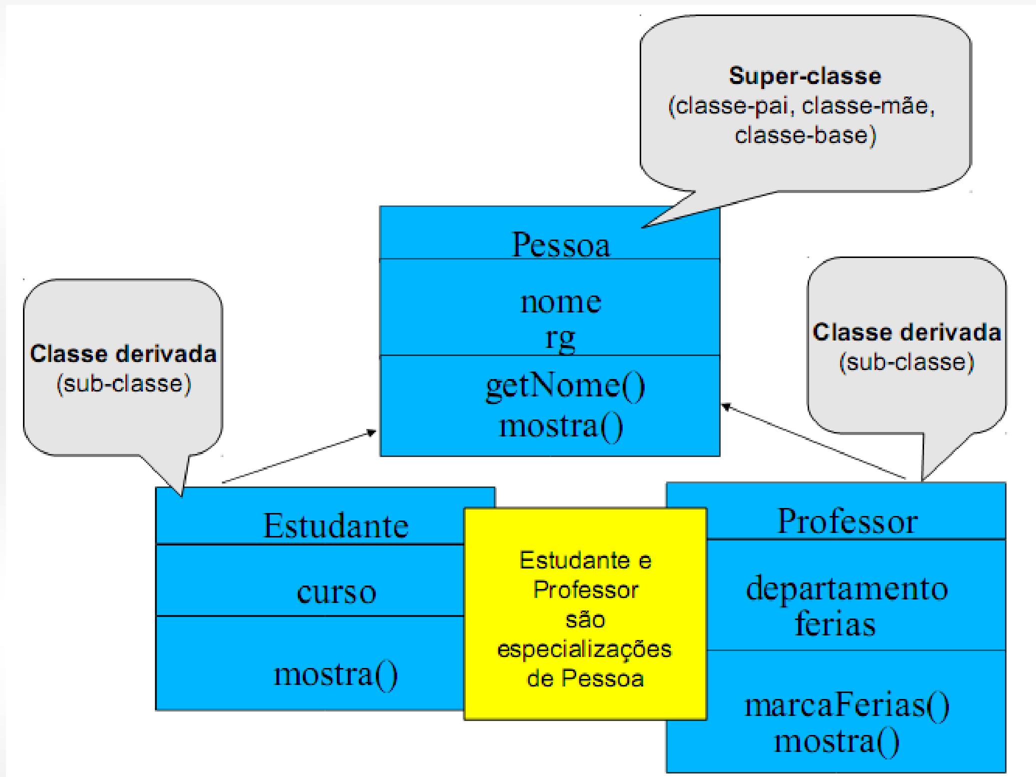


Figura 35: Terminologia da herança e seus relacionamentos.

e. Atributos (campos ou variáveis) são herdados.

- Classes: Estudante e Professor herdam atributos de Pessoa.

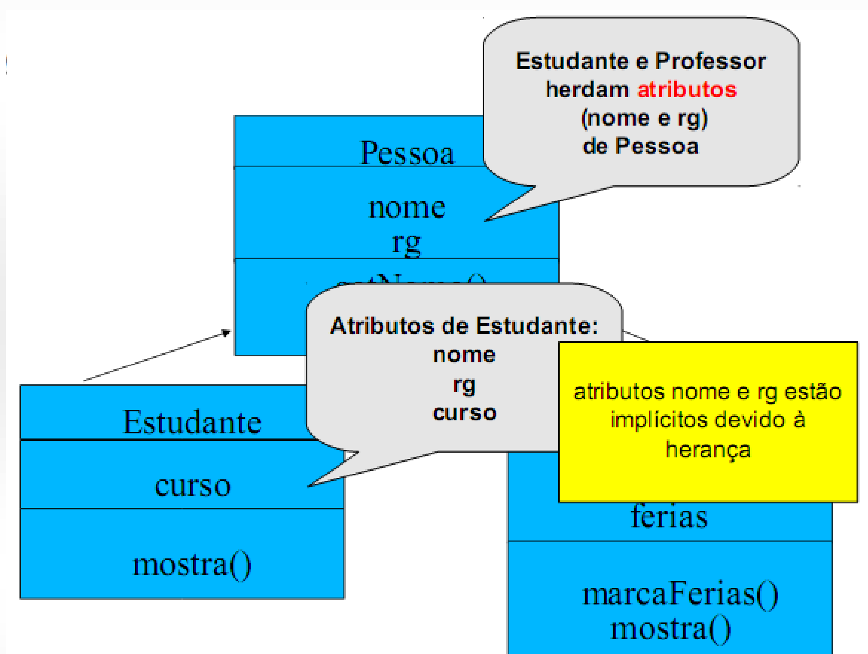


Figura 36: Estudante e professor herdam atributos de Pessoa

f. Métodos (ações ou funcionalidades) são herdados.

- Classes: Estudante e Professor herdam métodos de Pessoa.

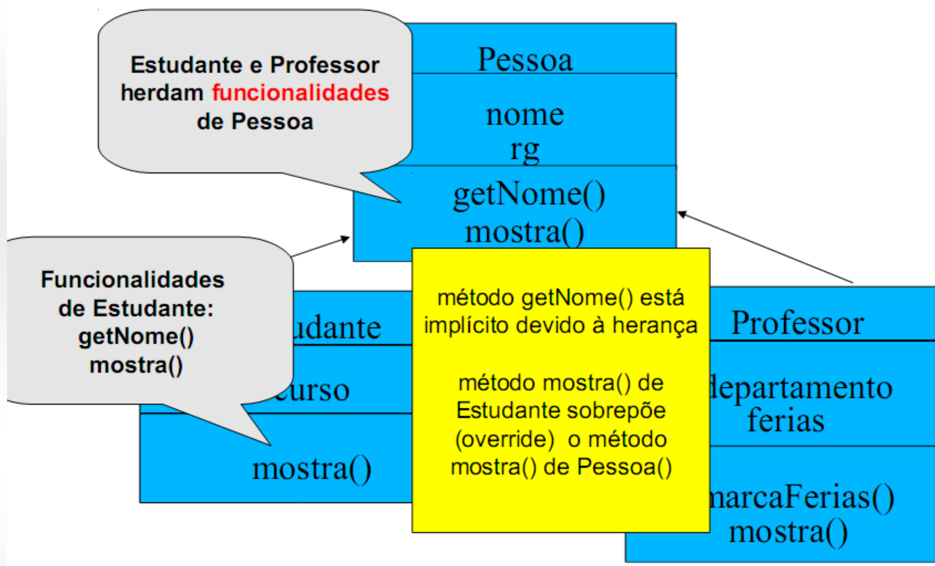
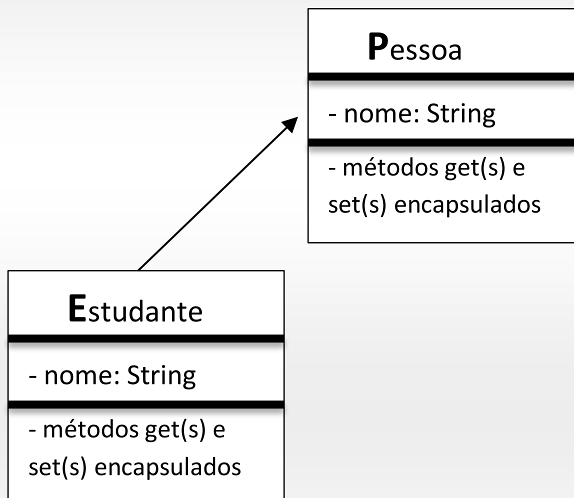


Figura 37: Estudante e professor herdam métodos de Pessoa

g. Exemplos de código utilizando herança.

- Classes: Estudante herda de Pessoa.

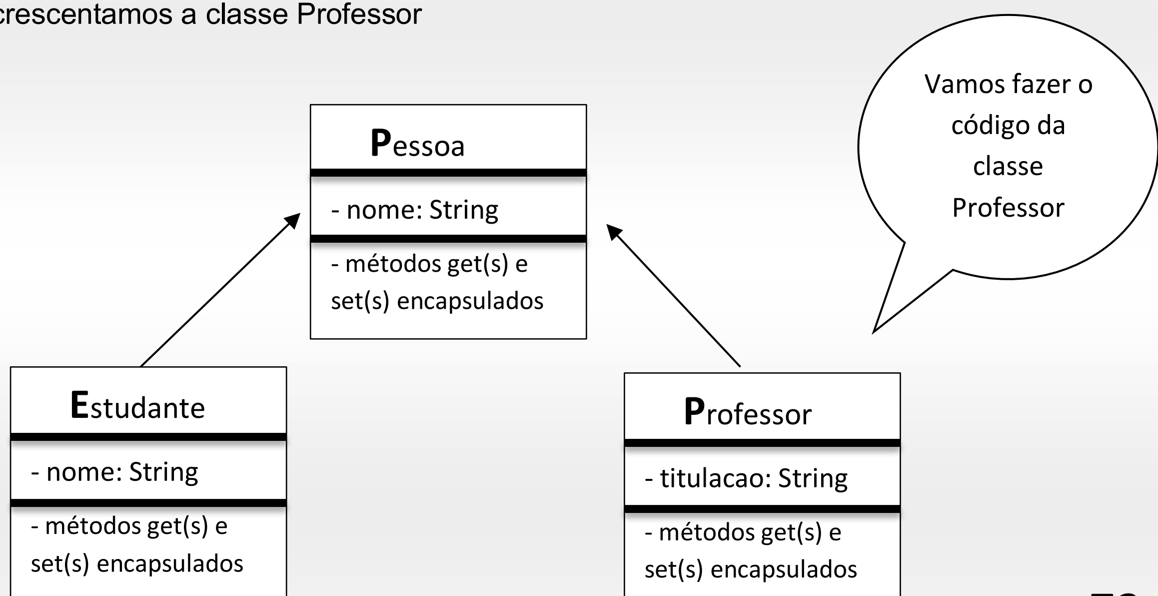



```
package unidadeIV;
public class Pessoa {
     private String nome;
   public String getNome() {
      return nome;
   }
   public void setNome(String nome) {
      this.nome = nome;
   }
}
```

```
package unidadeIV;
public class Estudante extends Pessoa {   //aqui , estudante herda de Pessoa (extends)

    private String curso;

   public String getCurso() {
      return nome;
   }
   public void setCurso(String curso) {
      this.curso = curso;
   }

@Override
   public String toString() {
      return  "Curso: " + getCurso() + "\n"+
               "Nome: " + getNome();    //aqui uso getNome() que está na classe Pessoa, posso
                                         //utilizar porque Estudante é extensão de Pessoa.
   }
}
```

```
package unidadeIV;
public class TestaEstudante{

   public static void main(String[] args) {

      Estudante e1 = new Estudante();

      e1.setNome("Maria");       //método setNome() herdado de Pessoa
      e1.setCurso("Informática");

      System.out.println(e1);
   }
}
```

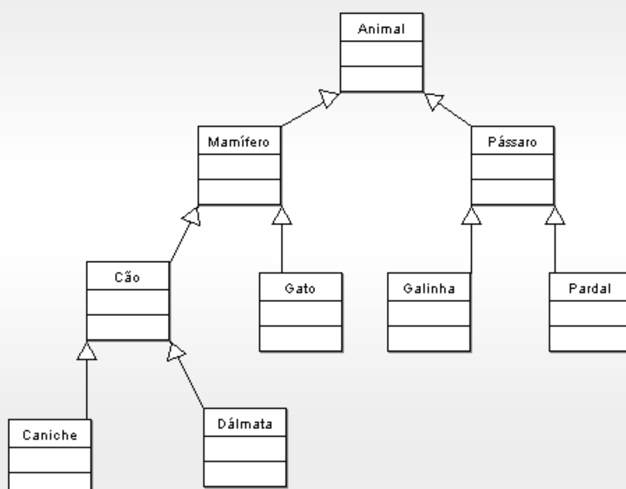
Acrescentamos a classe Professor

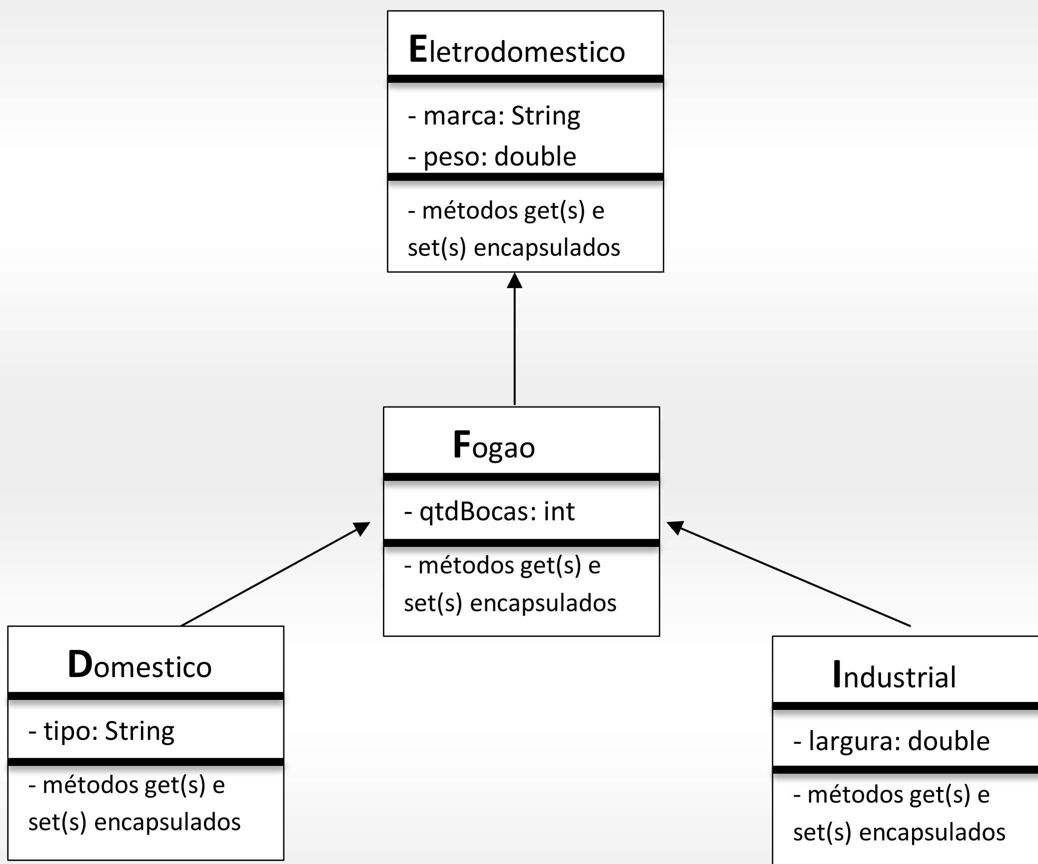


```
package unidadeIV;

public class Professor extends Pessoa {   //aqui , estudante herda de Pessoa (extends)

 private String titulacao;

  public String getTitulacao() {
      return titulacao;
   }

   public void setTitulacao(String titulacao) {
      this.titulacao = titulacao;
   }

@Override
   public String toString() {
      return  "Nome: " + getNome()+ "\n"+    //Método getNome() herdado de Pessoa
               "Titulação " + getTitulacao();
   }

}
```

```
package unidadeIV;

public class TestaProfessor{

   public static void main(String[] args) {

      Professor p1 = new Professor();

      p1.setNome("Paulo");       //método setNome() herdado de Pessoa
      p1.setTitulacao("Mestre");

      System.out.println(p1);
   }
}
```

* Outro exemplo com mais de dois níveis.






```
package unidadeIV;
public class Eletrodomestico {
    private String marca;
    private double peso;
  public String getMarca() {
     return marca;   }

  public void setMarca(String marca) {
     this.marca = marca;   }

  public double getPeso() {
     return peso;   }

  public void setPeso(double peso) {
     this.peso = peso;  }
}
```

```
package unidadeIV;
public class Fogao extends Eletrodomestico {
    private int qtdBocas;
  public int getQtdBocas() {
     return qtdBocas;   }

  public void setQtdBocas(int qtdBocas) {
     this. qtdBocas = qtdBocas;   }
}
```

```
public class Domestico extends Fogao {

   private String tipo;

   public String getTipo() {
      return tipo;   }

   public void setTipo(String tipo) {
      this. tipo = tipo;   }

   public String toString() {
   return  "Marca: " + getMarca()+ "\n"+   //Método getMarca() herdado de Fogao que herdou de
                                             Eletrodomestico
             "Peso: " + getPeso()  + "\n"+    //Método getPeso() herdado de Fogao que herdou de
                                               Eletrodomestico
             "Tipo " + getTipo()  + "\n"+

            "Quantidade de Bocas: " + getQtdBocas();   //Método getQtdBocas() herdado de Fogao
   } }
```

```
public class Industrial extends Fogao {

   private double largura;

   public double getLargura() {
      return largura;   }

   public void setLargura(double largura) {
      this. largura = largura;   }

   public String toString() {
   return  "Marca: " + getMarca()+ "\n"+   //Método getMarca() herdado de Fogao que herdou de
                                             Eletrodomestico
             "Peso: " + getPeso()  + "\n"+    //Método getPeso() herdado de Fogao que herdou de
                                               Eletrodomestico
             "Largura " + getLargura()  + "\n"+
            "Quantidade de Bocas: " + getQtdBocas();   //Método getQtdBocas() herdado de Fogao
   }  }
```

```
public class Executa{
    public static void main(String[] args) {

     Domestico d1 = new Domestico();

     d1.setMarca("Electrolux");  //métodos herdados
     d1.setPeso("25");
     d1.setQtdBocas("4");
     d1.setTipo("cooktop");

      Industrial ind1 = new Industrial();
      ind1.setMarca("TRON");   //métodos herdados
      ind1.setPeso("100");
      ind1.setQtdBocas("4");
      ind1.setLargura("100");

     System.out.println(d1);
     System.out.println(ind1);
   } }
```

REFERÊNCIAS:

CARVALHO, Victor Albani.; TEIXEIRA, Giovany Frossard **Programação Orientada a Objetos.** Disponível em: <http://redeetec.mec.gov.br/images/stories/pdf/eixo_infor_comun/tec_inf/081112_progr_obj.pdf >. Acesso em: 20 jul. 2013.

BORSOI, Beatriz Terezinha.; BRITO, Robison Cris **Linguagem de Programação Comercial. Parte I.** Disponível em: <http://redeetec.mec.gov.br/images/stories/pdf/eixo_infor_comun/tec_inf/081112_progr_obj.pdf >. Acesso em: 20 jul. 2013.