

Disciplina POO
Programação Orientada a Objetos
Linguagem Java
Apostila

PROFESSORA DRA. LILIANE JACON

Porto Velho, outubro/2020

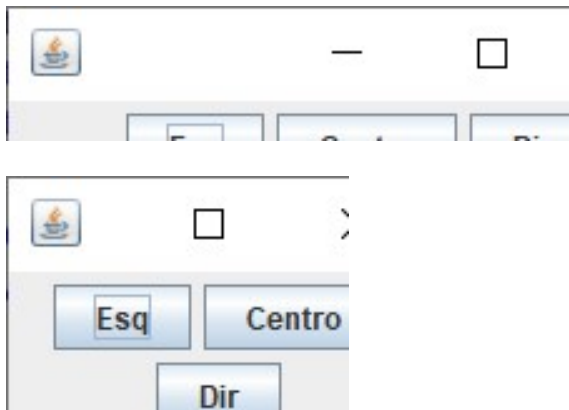
1. GERENCIADORES DE LAYOUT'S

Os gerenciadores de Layout são fornecidos a fim de organizar componentes GUI (graphical user interfaces) para apresentação. O método `setLayout` permite que você defina o gerenciador de Layout desejado.

FlowLayout()

É o gerenciador de layout mais simples. Os componentes são colocados da esquerda para a direita, na ordem que são adicionados. Quando a borda for alcançada, os componentes continuarão a ser exibidos na próxima linha.

Caso a janela seja redimensionada pelo usuário, os componentes são remanejados, conforme ilustração abaixo:



```
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JButton;
```

```
public class JanelaFlowLayout extends JFrame
```

```
{
    private JButton BotaoEsq;
    private JButton BotaoDir;
    private JButton BotaoCentro;
```

```
public JanelaFlowLayout() {
    setLayout ( new FlowLayout());
    BotaoEsq = new JButton("Esq");           add(BotaoEsq);
    BotaoCentro = new JButton("Centro");      add(BotaoCentro);
    BotaoDir = new JButton("Dir");           add(BotaoDir);
}
```

```
public static void main(String args[])
{
    JanelaFlowLayout f = new JanelaFlowLayout();
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setSize(300,75);
    f.setVisible(true);
}
}
```

GridLayout()

Organiza os componentes nas linhas e colunas, como se fossem posicionados numa matriz bidimensional (linha x coluna).

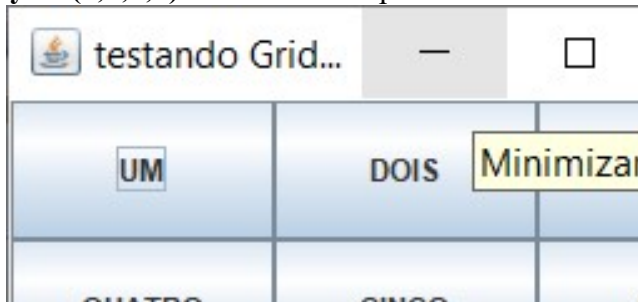
Os componentes são adicionados na parte superior esquerda e prosseguindo da esquerda para a direita até a linha estar cheia.

No exemplo abaixo, os botões foram inseridos na sequência 1,2,3,4,5 e 6.

GridLayout(3,2)



GridLayout(2,3,5,5) // lacunas de 5 por 5



```
import java.awt.GridLayout; import javax.swing.JFrame;
import javax.swing.JButton;
public class JanelaGridLayout extends JFrame {
    private JButton UM, DOIS, TRES, QUATRO, CINCO, SEIS;
    public JanelaGridLayout()
    {
        super("testando GridLayout");
        setLayout(new GridLayout(3,2)); // TESTAR COM (2,3,5,5)

        UM = new JButton("UM");          DOIS = new JButton("DOIS");
        TRES = new JButton("TRES");        QUATRO = new JButton("QUATRO");
        CINCO = new JButton("CINCO");      SEIS = new JButton("SEIS");

        add(UM);          add(DOIS);      add(TRES);
        add(QUATRO);      add(CINCO);      add(SEIS);
    }
    public static void main(String args[]) {
        JanelaGridLayout g = new JanelaGridLayout();
        g.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        g.setSize(300,200);
        g.setVisible(true);
    }
}
```

}

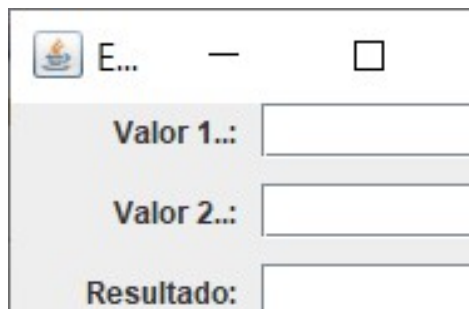
BorderLayout()

É o gerenciador padrão para JFrames. Organiza os componentes em cinco áreas NORTH, SOUTH, EAST, WEST e CENTER.



O BorderLayout permite adicionar apenas um único componente por região/área. Caso você deseje adicionar mais de um componente numa determinada área/região, primeiramente você deve criar um JPanel. No JPanel recém-criado, você adiciona quantos componentes forem necessários. Em seguida, você adiciona o JPanel na região/área desejada (norte, sul, leste, oeste ou centro).

O exemplo abaixo ilustra a utilização dos layout's BorderLayout e GridLayout.



```
import javax.swing.JButton; import javax.swing.JFrame;
import java.awt.GridLayout; import java.awt.BorderLayout;
import javax.swing.JPanel; import javax.swing.JTextField;
import javax.swing.JLabel; import javax.swing.SwingConstants;
import java.awt.event.*;
```

```
public class JanelaBorderLayout extends JFrame implements ActionListener {
    private JTextField valor1, valor2, resultado;
    private JLabel label1, label2, label3;
    private JButton limpa;
    private JButton adicao, subtracao, multiplicacao, divisao;

    public JanelaBorderLayout() {
        super ("Exemplo calculadora");
        //cria os componentes
        setLayout ( new BorderLayout()); // janela principal
    }
```

```

valor1=new JTextField(5);
valor2=new JTextField(5);
resultado = new JTextField(5);
label1 = new JLabel("Valor 1..:",SwingConstants.RIGHT);
label2 = new JLabel("Valor 2..:",SwingConstants.RIGHT);
label3 = new JLabel("Resultado:",SwingConstants.RIGHT);
limpa = new JButton("Clear");
adicao = new JButton("+");
subtracao = new JButton("-");
multiplicacao = new JButton("*");
divisao = new JButton("/");

//cria os paineis
JPanel centro = new JPanel(new GridLayout(3,2,10,10)); // Painei da região Central
JPanel leste = new JPanel(new GridLayout(4,1)); // painei da região Leste

//adciona os componentes
centro.add(label1);
centro.add(valor1);
centro.add(label2);
centro.add(valor2);
centro.add(label3);
centro.add(resultado);
add(centro,BorderLayout.CENTER);

leste.add(adicao);
leste.add(subtracao);
leste.add(multiplicacao);
leste.add(divisao);
add(leste,BorderLayout.EAST);

add(limpa,BorderLayout.SOUTH);

        limpa.addActionListener(this);
            adicao.addActionListener(this);
            subtracao.addActionListener(this);
            multiplicacao.addActionListener(this);
            divisao.addActionListener(this);
    } // fecha constructor JanelaBorderLayout

public void actionPerformed(ActionEvent e) {
    double valorfinal;
    if (e.getSource()==limpa){
        valor1.setText("");
        valor2.setText("");
        resultado.setText("");
    }
    if (e.getSource()==adicao){
        valorfinal = Float.parseFloat(valor1.getText()+
Float.parseFloat(valor2.getText());

```

```

        resultado.setText(String.format("%s",valorfinal));
    }
    if (e.getSource()==subtracao){
        valorfinal = Float.parseFloat(valor1.getText()) - Float.parseFloat(valor2.getText());
        resultado.setText(String.format("%s",valorfinal));
    }
    if (e.getSource()==multiplicacao){
        valorfinal = Float.parseFloat(valor1.getText()) * Float.parseFloat(valor2.getText());
        resultado.setText(String.format("%s",valorfinal));
    }
    if (e.getSource()==divisao){
        valorfinal = Float.parseFloat(valor1.getText()) / Float.parseFloat(valor2.getText());
        resultado.setText(String.format("%s",valorfinal));
    }
}

} // fecha o evento do clique dos botoes

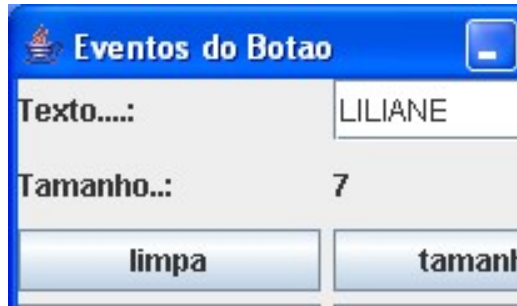
public static void main(String args[]) {
    JanelaBorderLayout f = new JanelaBorderLayout();
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setSize(250,160);
    f.setVisible(true);
}
} // fecha JanelaBorderLayout

```

2. EVENTOS DE BOTÕES

a) Criando uma nova classe para tratamento dos eventos

<pre>import javax.swing.JButton; import javax.swing.JFrame; import java.awt.GridLayout; import javax.swing.JTextField;</pre>	<pre>import javax.swing.JLabel; import java.awt.event.ActionListener; import java.awt.event.ActionEvent;</pre>
--	--



public class Exemplo extends JFrame implements ActionListener

```
{  
    private JTextField texto;  
    private JLabel tam;  
    private JButton limpa,maiusculo,minusculo,tamanho;
```

public Exemplo() //constructor

```
{  
    super ("Eventos do Botao");  
    setLayout(new GridLayout(4,2,5,5));  
  
    //cria os componentes  
    texto=new JTextField(15);  
    limpa = new JButton("limpa");  
    maiusculo = new JButton("maiusculo");  
    minusculo = new JButton("minúsculo ");  
    tamanho = new JButton("tamanho");  
  
    //adciona os componentes  
    JLabel lab1 = new JLabel("Texto....: "); add(lab1); add(texto);  
    JLabel lab2 = new JLabel("Tamanho..: "); add(lab2);  
    tam = new JLabel(); add(tam);  
  
    // botoes  
    add(limpa);          add(tamanho);  
    add(maiusculo);      add(minusculo);
```

```
    limpa.addActionListener(this);  
    tamanho.addActionListener(this);  
    maiusculo.addActionListener(this);  
    minusculo.addActionListener(this);  
} // fecha constructor
```

```

public void actionPerformed(ActionEvent e) {
    String frase = texto.getText();

    if (e.getSource()==limpa)
    {
        texto.setText("");
        tam.setText("");
    }
    else if (e.getSource()==tamanho)
        tam.setText(String.format("%s",frase.length()));
    else if (e.getSource()==maiusculo)
        texto.setText(String.format("%s",frase.toUpperCase()));
    else if (e.getSource()==minusculo)
        texto.setText(String.format("%s",frase.toLowerCase()));
} // fecha actionPerformed

public static void main(String args[])
{
    Exemplo x = new Exemplo();
    x.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    x.setSize(350,200);
    x.setVisible(true);
}

} // fecha class ExemploTexto2

```

3. Definição de Objetos, Classes, Instância e Métodos

Objeto = é a união inseparável entre uma estrutura de dados e todas as operações (métodos) associadas a esta estrutura

A estrutura de dados é denominada ATRIBUTOS

As operações são denominadas MÉTODOS.

Exemplo: Objeto PEDRO

Atributos: Nome=Pedro, Idade = 23

Métodos: CalcularIdade, InformarNome etc

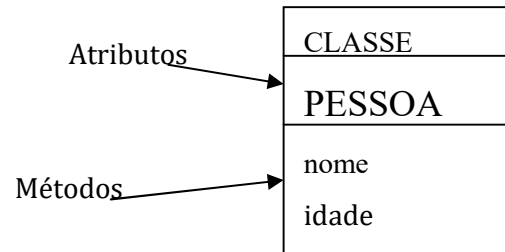
Uma aplicação orientada a objetos é decomposta em objetos, não em procedimentos ou subprogramas.

Portanto, um objeto é composto por uma coleção de dados privados e um conjunto de métodos que atuam sobre estes dados.

Classes, Instâncias e Mensagens

CLASSE: é um grupamento de objetos que revelam profundas semelhanças entre si, tanto no aspecto estrutural como funcional.

Uma classe pode ser vista como uma declaração de tipo, uma descrição de uma coleção de possíveis objetos.



Exemplos de Objetos Pessoa:

- João, 19

```
public class pessoa{  
    // atributos  
    private String nome;  
    private int idade;  
  
    // metodos  
    public void setNome( String nome) {  
        this.nome = nome; }  
  
    public void setIdade( int idade) {  
        this.idade = idade; }  
  
    public String getNome() {  
        return nome; }  
  
    public int getIdade() {  
        return idade; }  
} // fecha classe pessoa
```

testaPessoa.java

```
1 import java.util.Scanner;  
2 public class testaPessoa  
3 {  
4     public static void main (String args[])  
5     {  
6         pessoa pX = new pessoa();  
7         pessoa pZ = new pessoa();  
8         pZ.setNome("Jose Romualdo da Silva");  
9         pZ.setIdade(15);  
10        System.out.printf("nome Z= %s\n Idade= %d\n" ,pZ.getNome() ,pZ.getIdade() );  
11  
12        Scanner input = new Scanner( System.in );  
13        System.out.printf("Entre com o nome: ");  
14        String nome = input.nextLine();  
15        pX.setNome(nome);  
16        System.out.printf("\nEntre com a idade: ");
```

```

17 String frase = input.nextLine();
18 pX.setIdade(Integer.parseInt(frase));
19 System.out.println();
20 System.out.printf("nome X=%s\n Idade=%d \n ", pX.getNome() ,pX.getIdade() );
21 }
}

```

No exemplo anterior:

Pessoa = classe

pX e **pZ** são objetos da classe Pessoa

Podemos ter diversos objetos criados a partir de uma mesma classe. Esses objetos terão as mesmas características entre si, descritos pela classe. Desta forma, dizemos que cada um desses objetos é uma instância desta classe.

Cada objeto é uma instância de uma classe.

Objetos são criados durante a execução do programa, e ocupam espaço na memória.

Métodos

Um método é ativado pelo envio de uma mensagem ao objeto. O envio de uma mensagem a um objeto corresponde a uma chamada de uma função. Em ambos os casos algum trecho de código será executado e, provavelmente, modificará alguns dados.

Métodos Construtores / Sobrecarga de Construtores

Um *constructor* é um método especial que inicia o objeto antes que ele seja utilizado.

Um constructor deve ter o mesmo nome de sua classe.

A chamada do constructor é indicada pelo nome da classe seguido pelos parênteses.

Por exemplo, no programa **testaPessoa.Java**, nas linhas 6 e 7, são criados os objetos **pX** e **pZ**, respectivamente. A palavra chave *new* chama o constructor da classe para realizar a inicialização. Como um método, um constructor especifica em sua lista de parâmetros os dados que ele requer para realizar sua tarefa.

A seguir, apresentamos novamente as classes pessoa e testaPessoa, mas agora utilizando construtores.

public class Pessoa

```

{
    // atributos
    private String nome;
    private int idade;

    public Pessoa()
    { nome=""; idade=0;
    }

    public Pessoa(String nome,int idade) {
        this.nome=nome;
        this.idade=idade;
    }
}

```

Construtor SEM
parâmetros

Construtor COM
parâmetros

```

public void setNome( String nome) {
    this.nome = nome; }

public void setIdade( int idade) {
    this.idade = idade; }

public String getNome() {
    return nome; }

public int getIdade() {
    return idade; }
} // fecha classe pessoa

```

Observe agora que o objeto **pX** é inicializado *sem* parâmetros, e que o objeto **pZ** é inicializado *com* parâmetros.

```

public class testaPessoa
{
    public static void main (String args[])
    {
        pessoa pX = new pessoa();
        pessoa pZ = new pessoa("Jose Romualdo da Silva",15);
        System.out.printf("nome Z= %s\nIdade= %d\n",pZ.getNome(),pZ.getIdade());
    }
}

```

4. MODIFICADORES DE ACESSO PUBLIC E PRIVATE

Os atributos são os elementos que compõem a coleção de dados privados de uma classe. Eles (os atributos) contêm as informações do objeto e podem ser declaradas com modificadores de acesso PUBLIC e PRIVATE. Isto implica no tipo de acesso (ou visibilidade) que será possível realizar com estas informações.

a) Exemplo de membros PUBLIC:

Caso os atributos sejam declaradas PUBLIC dentro de uma determinada classe, é possível acessá-las externamente (utilizando um outro programa, é possível acessar diretamente os atributos utilizadas dentro da classe). Isto representa um grave perigo de violação das informações internas, segundo as regras de encapsulamento e ocultação da informação em POO – Programação Orientada a Objetos.

```

public class Pessoa {
    // atributos do tipo PUBLIC
    public String nome; // NÃO atende as boas regras da POO!!!
    public int idade;
}

```

```
// metodos
public Pessoa()
{
    nome="";    idade=0;
}

public Pessoa(String nome, int idade) {
    this.nome=nome;
    this.idade = idade;
}

public void setNome( String nome) {
    this.nome = nome;  }
```

.....// continuação da implementação da classe Pessoa....

A seguir, temos a classe testaPessoa1, cujo propósito é testar o acesso externo aos atributos na classe Pessoa:

```
import java.util.Scanner;
public class testaPessoa1
{
    public static void main (String args[])
    {
        pessoa pX = new pessoa();
        pX.nome="Luciana Vieira";
        pX.idade = 16;
        System.out.printf("nome X= %s \nIdade %d\n",
            pX.getNome(),pX.getIdade());
    }
}
```

O resultado obtido ao executarmos o programa testaPessoa1 é:

```
nome X= Luciana Vieira
Idade 16
```

Observação: notamos que é possível acessar externamente os atributos da classe Pessoa. Basta declararmos que as suas variáveis de instância são PUBLIC. Isto representa um alto risco para a integridade dos dados!!! Não utilize PUBLIC nos atributos. Isto não condiz com as regras da POO.

b) Exemplo de membros **PRIVATE**:

Utilizaremos o mesmo exemplo do item A (classe Pessoa). Ao invés de declararmos que os atributos são **PUBLIC**, iremos declará-los como **PRIVATE**. Ou seja, agora os atributos são visíveis apenas dentro da própria classe. Somente os métodos internos à classe Pessoa terão acesso direto as informações (atributos).

```
public class Pessoa {  
    // atributos PRIVATE  
    private String nome; // CORRETO!!!  
    private int idade;  
  
    // metodos  
    public Pessoa()  
    {  
        nome=""; idade = 0;  
    }  
  
    public Pessoa(String nome, int idade) {  
        this.nome=nome;  
        this.idade = idade;  
    }  
  
    public void setNome( String nome) {  
        this.nome = nome;  }  
}
```

.....// continuação da implementação da classe Pessoa....

A seguir, temos a classe testaPessoa1, cujo propósito é testar o acesso externo as variáveis de instância na classe Pessoa:

```
import java.util.Scanner;  
public class testaPessoa1  
{  
    public static void main (String args[])  
    {  
        pessoa pX = new pessoa();  
        pX.nome="Luciana Vieira";  
        pX.idade = 16;  
        System.out.printf("nome X= %s\nIdade %d\n",  
            pX.getNome(),pX.getIdade());  
    }  
}
```

Ao compilarmos o programa que contém a classe testaPessoa1, são listados 2 erros:

- *nome* has private access in pessoa
- *idade* has private access in pessoa

Observação: Quando declaramos que os atributos de uma classe são *private*, notamos que NÃO é possível acessar externamente suas informações internas. Ao declararmos que as informações internas são *private*, estaremos obedecendo as regras de encapsulamento e ocultação da informação da POO (programação orientada a objetos).

→ Como seria o programa testaPessoa1 correto? O objetivo é modificar os atributos do objeto pX. Como fazer tais alterações? A **resposta é: utilizando os métodos da classe Pessoa!**

```
public class testaPessoa1
{
    public static void main (String args[])
    {
        pessoa pX = new pessoa();
        pX.setNome("Luciana Vieira");
        pX.setIdade(16);
        System.out.printf("nome X= %s\n Idade %d\n",
            pX.getNome(),pX.getIdade());
    }
}
```

5. HERANÇA

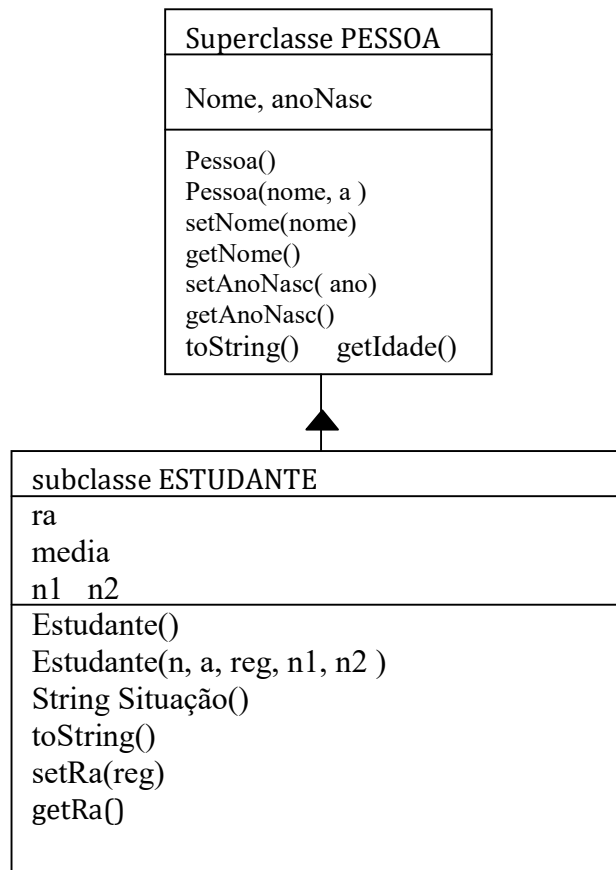
Permite que uma nova classe seja descrita a partir de outra classe já existente, herdando suas propriedades. Dizemos que a subclasse herda os métodos e os atributos de sua superclasse.

A subclasse deve ser capaz também de adicionar novos métodos e atributos aos originais.

Na subclasse é possível redefinir os métodos herdados, isto é, refazer a implementação do método, sem que a superclasse precise ser modificada.

Embora os nomes dos métodos possam ser duplicados na herança, os nomes dos atributos NÃO podem.

Assim, uma subclasse pode ser vista como uma especialização de sua superclasse. A especialização envolve a adição ou modificação de características mas nunca sua subtração.



Dizemos que um Estudante “é-um” Aluno.
O relacionamento “é-um” representa herança.

```

public class Pessoa {
    private String nome;
    private int anoNasc;

    public Pessoa(){
        this.nome="";
        this.anoNasc=0;
    }
    public Pessoa(String n,int a){
        this.nome=n;
        this.anoNasc=a;
    }
    public String toString(){
        return ("\nNome: "+this.nome+
            " Ano de Nascimento: "+this.anoNasc+"\n");
    }
    public String getNome() {
        return nome;
    }
}
  
```

```

    public void setNome(String nome) {
        this.nome = nome;
    }
    public int getAnoNasc() {
        return anoNasc;
    }
    public void setAnoNasc(int ano) {
        this.anoNasc = ano;
    }
    public int getIdade(){
        Calendar datahoje = Calendar.getInstance();
        int anohoje = datahoje.get(Calendar.YEAR);
        int idade = anohoje - this.anoNasc;
        return idade;
    }
}

public class Estudante extends Pessoa{
    private String ra;
    private double n1,n2;
    private double media;

    public Estudante(){
        super();
        this.ra="";
        this.n1=0.0;
        this.n2=0.0;
        this.media=0.0;
    }
    public Estudante(String n,int a, String reg, double n1, double n2){
        super(n,a);
        this.ra=reg;
        this.n1=n1;
        this.n2=n2;
        this.media=(this.n1+this.n2)/2;
    }
    // colocar os GET's e SET's para cada atributo
    public String Situacao(){
        this.media=(this.n1+this.n2)/2;
        if (this.media >=6)
            return "aprovado";
        else return "reprovado";
    }
    public String toString(){
        String s = super.toString()+
        String.format("Registro Academico=%s Nota1=%5.2f
                        Nota2=%5.2f Media=%5.2fn",
                        this.ra,this.n1,this.n2,this.media)+this.Situacao()+"\n";

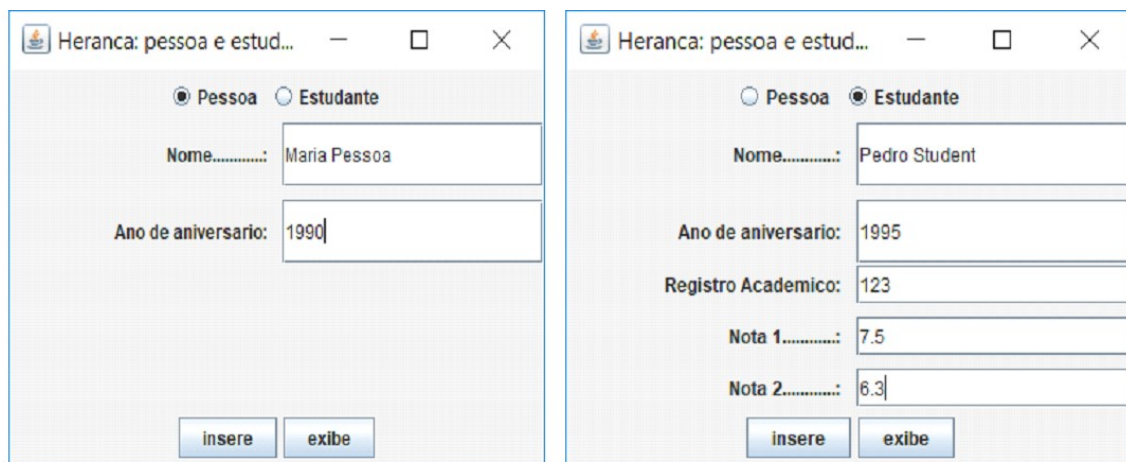
        return s;
    }
}

```


Membros Public, Private e Protected

- Os membros *public* de uma classe são visíveis internamente e externamente à classe. Isto significa que as subClasses também tem acesso os atributos de sua superClasse. Isto significa que NÃO devemos utilizar este tipo de modificador de acesso para atender as regras de ocultação da informação;
- Os membros *private* de uma classe só são acessíveis por dentro da própria classe. Suponha que as variáveis de instância de uma superClasse foram declaradas como “*private*”. Isto significa que as subClasses que herdam desta superClasse, não conseguem enxergar/acessar os atributos herdados da superClasse. E como resolver isto? As subClasses devem utilizar os métodos SET e GET da superClasse para acessar/alterar os atributos herdados. Deve-se priorizar a utilização de membros *private* com o objetivo de incentivar uma engenharia de software adequada.
- O acesso *protected* oferece um nível intermediário entre acesso *public* e *private*. Isto significa que as subClasses podem acessar diretamente os atributos da superClasse, da qual herdaram. Podem surgir vários problemas de manutenção do software ao utilizar este modificador de acesso (*protected*). Iremos evitar o seu uso.

Exercício: FAZER A TELA PRINCIPAL PARA criar um ARRAYLIST de Pessoa e Estudante



```
public class TelaGrafica extends JFrame implements ActionListener,
ItemListener{
    private JRadioButton rdPessoa;
    private JRadioButton rdEstudante;
    private ButtonGroup grupo;

    private LinkedList <Pessoa> lista = new LinkedList<Pessoa>();

    public TelaGrafica(){
        super("Heranca: pessoa e estudante - LIST");
        setLayout(new BorderLayout());
        grupo = new ButtonGroup();
```

```
rdPessoa = new JRadioButton("Pessoa",false);
rdEstudante = new JRadioButton("Estudante",false);
grupo.add(rdPessoa);
grupo.add(rdEstudante);
JPanel norte = new JPanel(new FlowLayout());
norte.add(rdPessoa);
norte.add(rdEstudante);
add(norte, BorderLayout.NORTH);
```

```
rdPessoa.addItemListener(this);
rdEstudante.addItemListener(this);
```

```
lnome = new JLabel("Nome.....: ", SwingConstants.RIGHT);
lano = new JLabel("Ano de aniversario: ", SwingConstants.RIGHT);
lra = new JLabel("Registro Academico: ", SwingConstants.RIGHT);
lnota1 = new JLabel("Nota 1.....: ", SwingConstants.RIGHT);
lnota2 = new JLabel("Nota 2.....: ", SwingConstants.RIGHT);
```

```
tnome = new JTextField(40);
tano = new JTextField(4);
painelpessoa = new JPanel(new GridLayout(2,2,8,8));
painelpessoa.add(lnome);
painelpessoa.add(tnome);
painelpessoa.add(lano);
painelpessoa.add(tano);
painelpessoa.setVisible(false);
//
painelestuda = new JPanel(new GridLayout(3,2,8,8));
tra = new JTextField(6);
tn1 = new JTextField(5);
tn2 = new JTextField(5);
painelestuda.add(lra);
painelestuda.add(tra);
painelestuda.add(lnota1);
painelestuda.add(tn1);
painelestuda.add(lnota2);
painelestuda.add(tn2);
painelestuda.setVisible(false);
```

```
JPanel centro = new JPanel(new GridLayout(2,1));
centro.add(painelpessoa);
centro.add(painelestuda);
add(centro, BorderLayout.CENTER);
```

```
bInsere = new JButton("insere");
bExibe = new JButton("exibe");
btInsere.addActionListener(this);
btExibe.addActionListener(this);
```

```
JPanel sul = new JPanel(new FlowLayout());
```

```

        sul.add(bInsere);
        sul.add(bExibe);
        add(sul, BorderLayout.SOUTH);
    } // fecha o construtor

```

```

public void itemStateChanged(ItemEvent evento){ // eventos dos radioButtons
    if (evento.getSource()==rdPessoa){
        painelpessoa.setVisible(true);
        painelestuda.setVisible(false);
    }
    if (evento.getSource()==rdEstudante){
        painelpessoa.setVisible(true);
        painelestuda.setVisible(true);
    }
} // fecha itemStateChanged

```

```

public void actionPerformed(ActionEvent e){ // eventos dos botões Insere e Exibe
    if (e.getSource() == btInsere) {
        double nota1, nota2, media;
        int ano;
        String data;
        Estudante a;
        Pessoa p;
        String nome = tnome.getText();
        String an = tano.getText();
        try{
            ano=Integer.parseInt(an);
        }catch(NumberFormatException erro){
            ano=1990;
        }
        if (rdEstudante.isSelected()){
            String ra = tra.getText();

            String n1 = tn1.getText();
            String n2 = tn2.getText();
            try{
                nota1 = Double.parseDouble(n1);
                nota2 = Double.parseDouble(n2);
            }catch (NumberFormatException erro){
                nota1=0; nota2=0;
            }
            a = new Estudante(nome,ano,ra,nota1,nota2);
            lista.add(a);
        }
        else {
            p = new Pessoa(nome,ano);
            lista.add(p);
        }

        tnome.setText("");tano.setText("");
        tra.setText("");tn1.setText(""); tn2.setText("");
    }
}

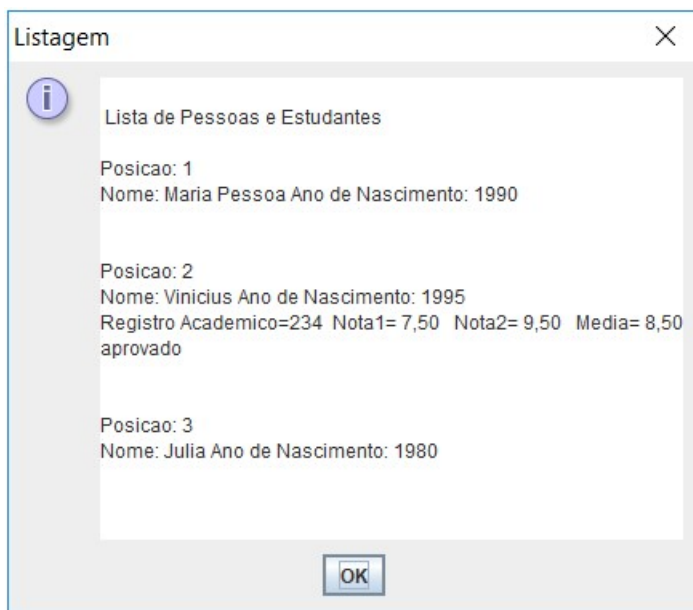
```

```

        JOptionPane.showMessageDialog(null,
                                     "Incluido na lista com sucesso");
    } else if (e.getSource() == btExibe) {
        String saida="\n Lista de Pessoas e Estudantes\n\n";
        int i=1;
        for (Pessoa p:lista){
            saida += "Posicao: "+i+p.toString()+"\n\n";
            i++;
        }
        JTextArea area = new JTextArea(saida,11,10);
        JOptionPane.showMessageDialog(null,area,"Listagem",
                                    JOptionPane.INFORMATION_MESSAGE);
    } // fecha IF do botão Exibe
} // fecha actionPerformed

public static void main(String[] args) {
    TelaGrafica x = new TelaGrafica();
    x.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    x.setSize(400,300);
    x.setLocationRelativeTo(null);
    x.pack();
    x.setVisible(true);
}

```

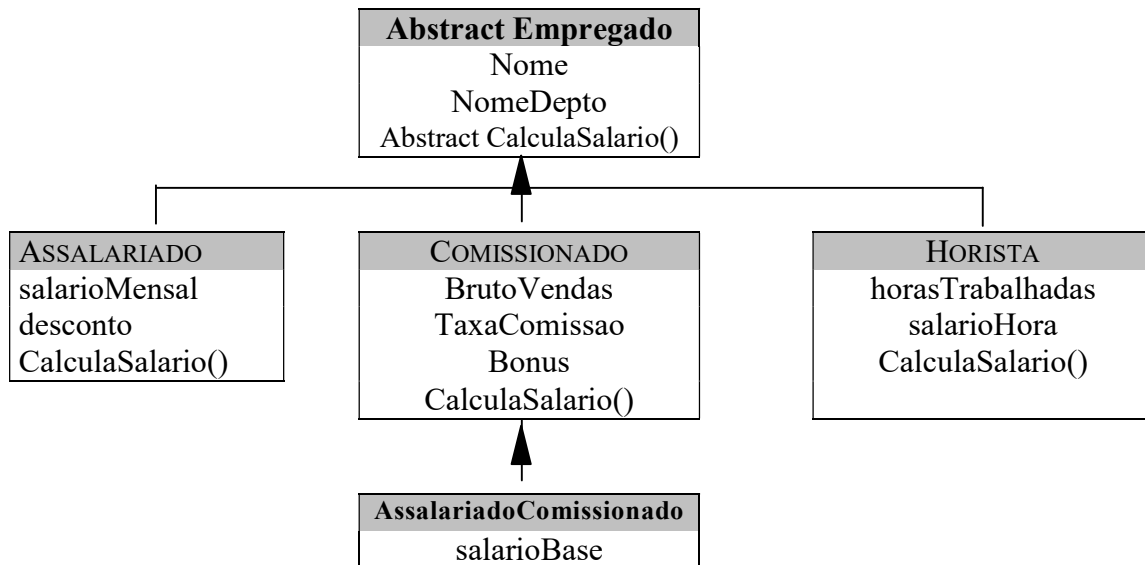


6. CLASSES ABSTRATAS

São classes que nunca poderão instanciar objetos, pois são classes incompletas. As subclasses (herdadas da classe abstrata) devem declarar as partes ausentes. O propósito de uma classe abstrata é fornecer uma superclasse apropriada, a partir da qual outras classes podem herdar e assim compartilhar um projeto comum.

No exemplo abaixo, a superClasse Empregado será declarada Abstrata, ou seja, será uma classe definida com o propósito de especificar o que é comum entre as

subClasses Assalariado, Comissionado, Horista e AssalariadoComissionado. Fazer um método `CalculaSalario()` para cada subclasse, contendo os cálculos necessários de acordo com seus atributos. O Método `CalculaSalario()` da superclasse também é abstrato.



```
ArrayList <Empregado> lista = new ArrayList <Empregado>();
```

- A classe Abstrata contém um ou mais métodos abstratos
- Métodos abstratos não fornecem implementações
- cada subClasse concreta de uma superClasse Abstrata deve, obrigatoriamente, re-escrever os métodos que foram declarados abstratos na superClasse. A implementação (código) do método deve ser realizada na subClasse.
- os construtores e métodos static não podem ser declarados abstract
- as subClasses não podem sobrescrever métodos static

—Neste programa, é preciso calcular o salário de todos os EMPREGADOS da empresa. Para isto, através de um comando FOR, todos os objetos do ArrayList de Empregados executam o método **calcula salário**. Ou seja, o método `calculasalario` é executado para cada objeto armazenado na lista (veja código abaixo).

—**Observe** que não é feita uma verificação para descobrir qual tipo de objeto está armazenado em cada posição do ArrayList. O método **calculasalário** possui a mesma assinatura em todas as classes (neste exemplo, este método não possui argumentos).

```

for (i=0;i<lista.size();i++){
    saida += lista.get(i).toString()+
        " Salario:"+lista.get(i).calculaSalario()+"\n\n";
}
  
```

A seguir tem-se as implementações das classes acima especificadas.

a) superClasse ABSTRATA Empregado

```
public abstract class Empregado {  
    private String nome;  
    private String nomeDepto;  
  
    public Empregado() // constructor SEM parâmetros  
    {  
        nome="";  
        nomeDepto="";  
    }  
  
    public Empregado(String n, String nd)  
    // construtor COM parâmetros  
    {  
        this.nome = n;  
        this.nomeDepto=nd;  
    }  
  
    // metodo abstrato a ser reescrito e implementado nas subClasses  
    public abstract double calculaSalario(); // nenhum implementacao aqui  
  
    public void setNomeDepto( String nd) {  
        this.nomeDepto = nd;    }  
  
    public void setNome( String nome) {  
        this.nome = nome;  
    }  
  
    public String getNomeDepto() {  
        return (this.nomeDepto);  
    }  
  
    public String getNome() {  
        return (this.nome);  
    }  
  
    public String toString() {  
        return String.format("\nNome=%s Depto=%s",  
                               this.getNome(),this.getNomeDepto());  
    }  
} // fecha classe empregado
```

b) subClasse Assalariado (herdada de Empregado)

```
public class Assalariado extends Empregado  
{  
    private double salarioMensal;  
    private double descontos;  
  
    public Assalariado()
```

```

    { super();
      this.salarioMensal=0;
      this.descontos=0;
    }

    public Assalariado(String nome, String nd, double salario, double descontos)
    {
        super(nome,nd);
        this.salarioMensal = salario;
        this.descontos = descontos;
    }

    // metodo OBRIGATORIO devido a herança da superClasse abstrata Empregado
    public double calculaSalario() {
        return (this.salarioMensal-this.descontos);
    }

    public void setSalarioMensal(double salario) {
        this.salarioMensal = salario;
    }

    public void setDescontos( double descontos) {
        this.descontos = descontos;
    }

    public double getSalarioMensal() {
        return (this.salarioMensal);
    }

    public double getDescontos() {
        return (this.descontos);
    }

    public String toString() {
        String temp = "\nAssalariado "+super.toString()+ String.format(
            "Salario Mensal=$%,.2f Desc=$%,.2f Salario Final=$%,.2f",
            this.getSalarioMensal(),this.getDescontos(), this.calculaSalario());
        return (temp);
    }
} // fecha classe assalariado

```

c) subClasse Horista (herdada de Empregado)

```

public class Horista extends Empregado
{
    private int horasTrabalhadas;
    private double salarioHora;

    public Horista() // constructor SEM parâmetros
    { super();
      this.salarioHora=0;   this.horasTrabalhadas=0;
    }
}

```

```

    }

    public Horista(String nome, String nd, int horasTrabalhadas, double salarioHora) {
        super(nome,nd);
        this.horasTrabalhadas = horasTrabalhadas;
        this.salarioHora = salarioHora;
    }

    // metodo OBRIGATORIO devido a heranca da superClasse abstrata Empregado
    public double calculaSalario() {
        if (this.horasTrabalhadas <= 40)
            return(horasTrabalhadas * salarioHora);
        else {
            int hsExtras = horasTrabalhadas - 40;
            return((salarioHora * 40) + (hsExtras * (salarioHora * 1.5)));
        }
    }

    public void setSalarioHora(double salario) {
        this.salarioHora = salario;
    }

    public double getSalarioHora() {
        return (this.salarioHora);
    }

    public int getHorasTrabalhadas() {
        return(this.horasTrabalhadas);
    }

    public String toString() {
        String temp = "\nHorista "+super.toString()+ String.format(
            "Hs Trabalhadas=%d Salario Hora=$%,.2f Salario final=$%,.2f",
            this.getHorasTrabalhadas(),this.getSalarioHora(),this.calculaSalario());
        return (temp);
    }
} // fecha classe Horista

```

d) subClasse Comissionado (herdada de Empregado)

public class Comissionado extends Empregado

```

{
    private double brutoVendas;
    private double taxaComissao;
    private double bonus;

    public Comissionado()
    { super();
      this.brutoVendas=0;
      this.taxaComissao=0;
    }
}

```



```

        this.bonus=0;
    }

    public Comissionado(String nome, String nd, double brutoVendas,
                        double taxaComissao,double bonus)
    {
        super(nome,nd);
        this.brutoVendas = brutoVendas;
        this.taxaComissao = taxaComissao;
        this.bonus = bonus;
    }

    // metodo OBRIGATORIO devido a heranca da superClasse abstrata Empregado
    public double calculaSalario() {
        return (brutoVendas * taxaComissao) + this.bonus;
    }

    public void setBrutoVendas(double bruto) {
        brutoVendas = bruto;
    }

    public void setTaxaComissao( double taxa) {
        this.taxaComissao = taxa;
    }

    public void setBonus (double bonus) {
        this.bonus = bonus;
    }

    public double getBrutoVendas() {
        return (this.brutoVendas);
    }

    public double getTaxa() {
        return(this.taxaComissao);
    }

    public double getBonus() {
        return (this.bonus);
    }

    public String toString() {
        String temp = "\nComissionado "+super.toString()+String.format(
        "Bruto Vendas=%%,.2f Taxa Comissao=%%,.2f Bonus=%%,.2f Salario Final=%%,2f",
        this.getBrutoVendas(),this.getTaxa(),this.getBonus(),this.calculaSalario());
        return (temp);
    }
}
} // fecha classe Comissionado

```

d) subClasse AssalariadoComissionado (herdada de Comissionado)

Segundo a hierarquia de classes apresentada, a classe AssalariadoComissionado herda da classe Comissionado, e a classe Comissionado herda da classe Empregado. Desta forma, a subClasse Comissionado é considerada superClasse para a classe AssalariadoComissionado.

```
public class AssalariadoComissionado extends Comissionado
{
    private double salarioBase;

    public AssalariadoComissionado()
    {
        super(); // chama constructor de Comissionado
        this.salarioBase=0;
    }

    public AssalariadoComissionado(String nome,String nd, double brutoVendas,
                                   double taxaComissao,double bonus, double salarioBase)
    {
        super(nome,nd,brutoVendas,taxaComissao,bonus); // constructor Comissionado
        this.salarioBase = salarioBase;
    }

    // metodo OBRIGATORIO devido a herança
    public double calculaSalario() {
        return (salarioBase+ super.calculaSalario());
    }

    public void setSalarioBase(double salario) {
        this.salarioBase = salario;
    }

    public double getSalarioBase() {
        return (this.salarioBase);
    }

    public String toString() {
        String temp = "\nAssalariado Comissionado "+super.toString()+
            String.format("Salario Base=$%,.2f Sal.Final=$%,2f",
                this.getSalarioBase(), this.calculaSalario());
        return (temp);
    }
}
// fecha classe assalariadoComissionado
```

Fazer a tela gráfica para funcionamento das classes declaradas.

TELA GRÁFICA

- Painel Norte (contêm radioButtons para escolha do tipo de Empregado)
- Painel CENTRO(contém vários painéis. Muda de acordo com o radioButton escolhido previamente)

- Painel Sul. Contém 6 botões. Para inserir um objeto Empregado (ou Horista, ou Comissionado, ou Comissionado/Assalariado ou Assalariado) no arraylist e Também para exibir os objetos armazenados.

Você percebeu que existe um botão para cada objeto da hierarquia de classes apresentada. Por exemplo, “EXIBE APENAS HORISTAS”. Para exibir somente os horistas armazenados no ArrayList **Lista**, você deve utilizar o comando INSTANCEOF. Veja o exemplo a seguir.

// BOTÃO DE LISTAGEM

```
public void actionPerformed(ActionEvent evento) {
    String saida="\nListagem\n\n";
    int i;
    if (evento.getSource()==exibeTodos) {
        for (i=0;i<lista.size();i++) {
            saida += lista.get(i).toString()+
                " Salario:"+lista.get(i).calculaSalario()+"\n\n";
        }
    }
    else if(evento.getSource()==exiAssalariado) {
```

```

        for (i=0;i<lista.size();i++){
            if (lista.get(i) instanceof Assalariado)
                saida += lista.get(i).toString()+
                " Salario:"+lista.get(i).calculaSalario()+"\n\n";
        }
    }

    else if (evento.getSource()==exiComissionado){
        for (i=0;i<lista.size();i++){
            if (lista.get(i) instanceof Comissionado)
                saida += lista.get(i).toString()+
                " Salario:"+lista.get(i).calculaSalario()+"\n\n";
        }
    }

    else if (evento.getSource()==exiHorista){
        for (i=0;i<lista.size();i++){
            if (lista.get(i) instanceof Horista)
                saida += lista.get(i).toString()+
                " Salario:"+lista.get(i).calculaSalario()+"\n\n";
        }
    }

    JTextArea area = new JTextArea(saida,11,10);
    JOptionPane.showMessageDialog(null, area);
}
}

```

7. Polimorfismo

- Chamamos de polimorfismo a criação de uma família de funções que compartilham do mesmo nome, mas cada uma tem código independente.
- Permite que um método aceite como parâmetro objetos de mais uma classe. Para isto é necessário construir métodos capazes de receber diversos tipos de objetos como parâmetros. No entanto, é necessário que o protocolo de suas classes tenham um mínimo de afinidade (herança).

No exercício anterior, temos a superclasse abstrata EMPREGADO.

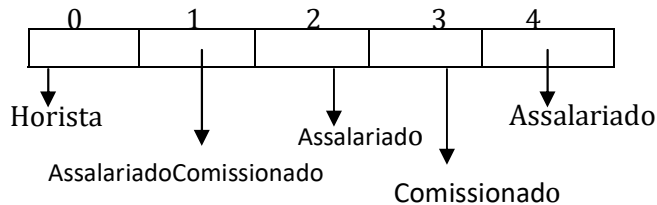
Na hierarquia de classes apresentadas, tem-se a derivação das sub classes concretas Horista, Comissionado, Comissionado/Assalariado e Assalariado.

Veja o exemplo da **função calculaSalario()** nas classes citadas: cada uma tem uma implementação diferente. Ou seja, possuem a mesma assinatura (tem o mesmo nome) mas a implementação é diferente em cada classe da hierarquia apresentada.

ArrayList <Empregado> **Lista** = new ArrayList <Empregado> ();

Array Polimórfico: considere a hierarquia de classes anteriormente criadas: Empregado, Comissionado, Horista, Assalariado e AssalariadoComissionado.

Utilizaremos um vetor (*ArrayList*) para armazenar objetos do tipo Empregado. Se cada posição do *array* *permite* um objeto do tipo Empregado, significa que é permitido armazenar objetos do tipo Horista, Assalariado, AssalariadoComissionado e Comissionado, pois estes objetos são derivados da classe Empregado (relacionamento "é-um"). Veja ilustração abaixo..



No projeto apresentado, como fica o botão INSERE (evento dentro da função ?

```

public void actionPerformed(ActionEvent evento){ //eventos dos botões
    String nome, nomedp;
    double salarioHora;
    int horasTrabalhadas;
    if (evento.getSource() == btnInsere){ // evento botão INSERE
        nome = txtNome.getText();
        nomedp = txtNomeDepto.getText();
        if (rdHorista.isSelected()){ // se o radioButton for HORISTA
            try{
                salarioHora = Double.parseDouble(txtSalarioHora.getText());
                horasTrabalhadas = Integer.parseInt(txtHorasTrabalhadas.getText());
            }
            catch(NumberFormatException erro){
                salarioHora=0.0;
                horasTrabalhadas = 0;
            }
            Horista h = new Horista(nome,nomedp,horasTrabalhadas,salarioHora);
            Lista.add(h);
            JOptionPane.showMessageDialog(null, "Horista cadastrado com sucesso");
            txtNome.setText("");txtNomeDepto.setText("");
            txtSalarioHora.setText("");txtHorasTrabalhadas.setText("");
            // fecha o IF para inserir um Horista

            if (rdComissionado.isSelected()){
                //inserir um objeto Comissionado na Lista
            }
            if (rdAssalariado.isSelected()){
                //inserir um objeto Assalariado na Lista
            }
        }
        // fecha o botão btnInsere

        if (evento.getSource() == exibeTodos){
        }
        if (evento.getSource() == exiAssalariado){
        }
    }
}

```

}

Resumo das atribuições permitidas entre superClasses e subClasses:

superClasse \Leftarrow superClasse

```
comissionado k = new comissionado("1234", "Jose", 3,3,1983,45000.00,0.03,0.0);
```

```
comissionado m = k;
```

- esta operação é simples e direta

subClasse \Leftarrow subClasse

```
assalariadoComissionado h = new assalariadoComissionado ("1111", "Pedrão", 1,1,1981,60000.00,0.05,0.0,350.00);
```

```
assalariadoComissionado r = h;
```

- esta operação é simples e direta

// superClasse \Leftarrow subClasse

```
comissionado x = new assalariadoComissionado ("3333", "Poliana", 3,3,1983,12500.00,0.05,0.0,650.00);
```

- superClasse \Leftarrow subClasse. Este tipo de atribuição é possível, mas apenas pode ser utilizado para referenciar os membros/métodos da superClasse. Se desejamos invocar um método da subClasse por meio do objeto **x**, o compilador informará erros.

```
x.setSalarioBase(double novoSalario); // erro pois SalarioBase é somente da
// subclasse assalariadoComissionado!!!
```

```
x.getSalarioBase() // erro!!!! getSalarioBase e setSalarioBase são da subClasse
```

```
x.toString(); // correto, executará o método toString() da subClasse
```

// subClasse \Leftarrow superClasse

```
assalariadoComissionado z = (assalariadoComissionado) vetor[i];
```

- No exemplo acima, vetor é um array de objetos do tipo empregado (superClasse). Para realizar esta atribuição é necessário fazer uma coerção explícita para o tipo da subClasse. O correto é verificar primeiramente se o objeto armazenado no vetor[i] é mesmo um tipo assalariadoComissionado, antes de realizar a coerção.

```
for (i=0; i < 5; i++)
```

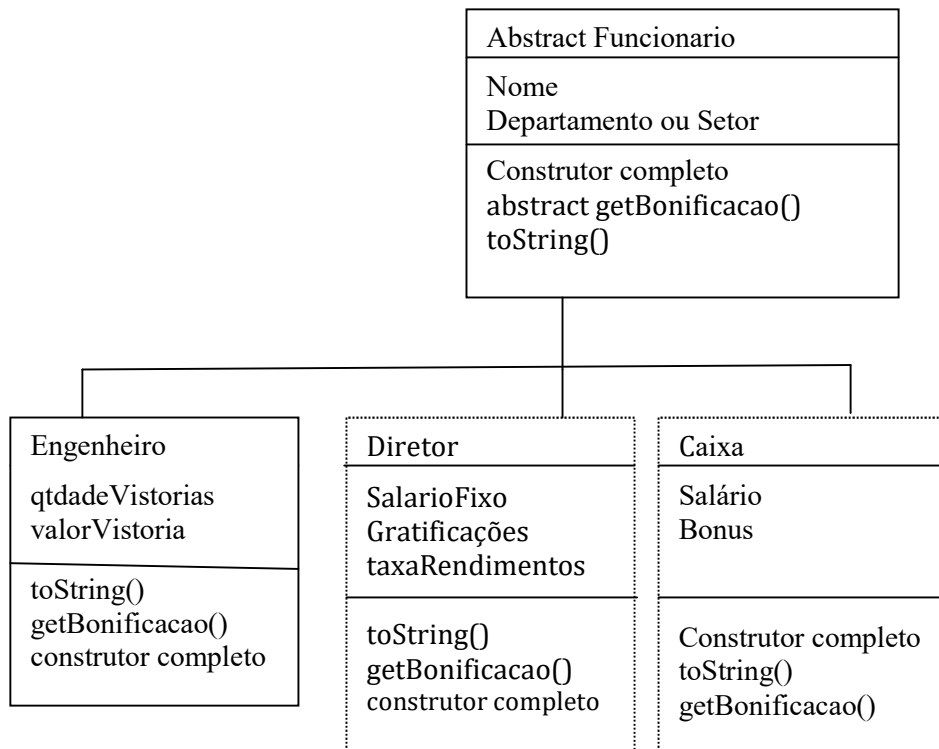
```
if (vetor[i] instanceof assalariadoComissionado) {
```

```
    assalariadoComissionado z = (assalariadoComissionado) vetor[i];
```

```
    z.setSalarioBase(6500.00); // correto!!
```

```
}
```

EXERCÍCIO: Implemente as classes (abstrata e concretas) :



Crie um ArrayList de Funcionarios.

Repare que o método **getBonificação()** na classe **Funcionário** é ABSTRATA (não tem código).

Implemente a tela abaixo contendo os eventos dos botões **Adiciona** e **Exibe Bonificação**.

Quando for engenheiro, exibe **Qtde Vistorias** e **Valor da Vistoria**.

Quando for Diretor, exibe **salário fixo**, **taxa de rendimentos** e **gratificações**.

Quando for Caixa, exibe o **salário** e o **Bônus**.

The screenshot shows a Java Swing window titled "Form1". It features a "Tipo" section with three radio buttons: "Engenheiro" (selected), "Caixa", and "Diretor". To the right of the radio buttons is a text field labeled "Nome". Below the radio buttons, there are three groups of input fields. The first group, for "Engenheiro", contains "Qtde Vistorias" and "Valor da Vistoria". The second group, for "Caixa", contains "Salario" and "Bonus". The third group, for "Diretor", contains "Salario Fixo", "Taxa de Rendimentos", and "Gratificações". At the bottom of the window, there are two buttons: "Exibe Bonificação de Todos" and "Adiciona".

8. Interface

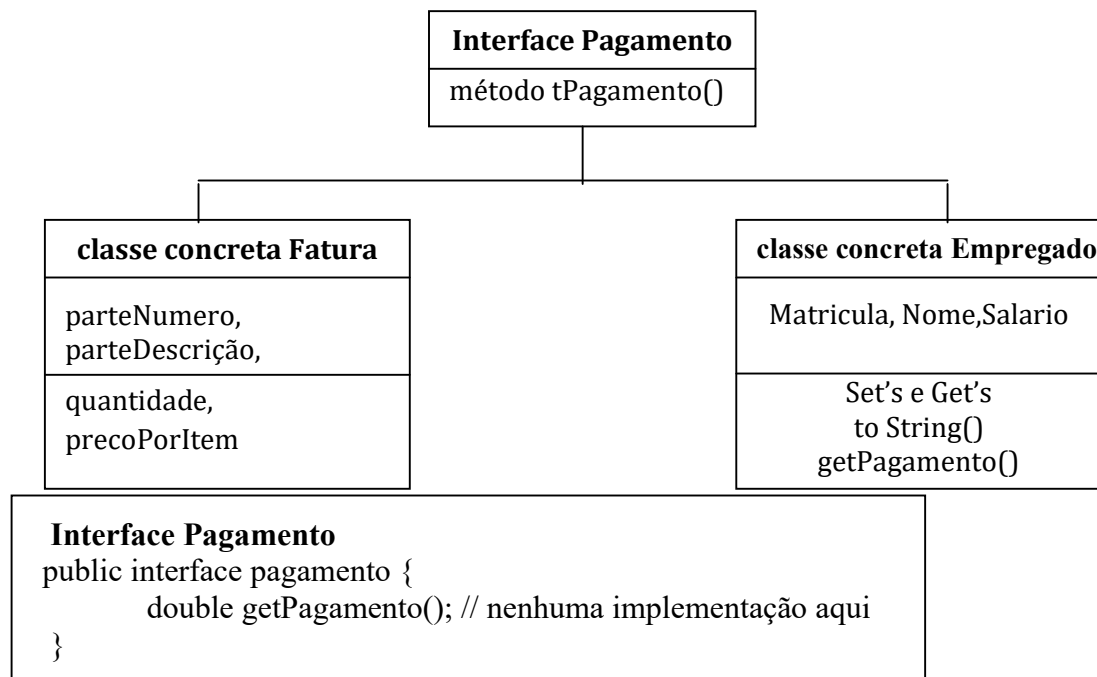
Interface é semelhante a herança, porém a interface força uma relação entre duas ou mais classes que não possuem características em comum.

Uma interface especifica quais operações (métodos) os usuários poderão invocar, mas não especifica como estas operações serão realizadas (na interface, os métodos não contêm implementação).

Uma declaração de interface inicia-se com a palavra-chave *interface* e contém somente constantes e métodos *abstract*. Diferentemente das classes, todos os membros da interface devem ser *public* e as interfaces não podem especificar nenhum detalhe de implementação como declarações de método concretos e variáveis de instância.

Portanto, todos os métodos declarados em uma interface são métodos *public abstract* (sem implementação) e todos os campos são *public, static* e *final* (ou seja, constantes).

Na hierarquia abaixo, modificaremos a classe *Empregado*. No capítulo anterior ela foi declarada como uma classe abstrata, agora ela será concreta. Como ela implementa a interface *Pagamento*, obrigatoriamente deverá implementar o método *getPagamento()*.



classe concreta Fatura

```
public class Fatura implements Pagamento {
```

```
    private String parteNumero;
    private String parteDescricao;
    private int quantidade;
    private double precoPorItem;
```

```
    public Fatura(String parteN, String parteD, int cont, double preco)
    {
        this.parteNumero=parteN;
        this.parteDescricao=parteD;
        this.quantidade = cont;
    }
}
```



```

        this.precoPorItem=preco;
    }
    public void setParteNumero(String parte) { parteNumero=parte; }
    public String getParteNumero() { return parteNumero; }
    public void setParteDescricao(String parte) { parteDescricao=parte; }
    public String getParteDescricao() { return parteDescricao; }
    public void setQuantidade(int q) { quantidade = q; }
    public int getQuantidade() { return(quantidade); }
    public void setPreco(double p) { precoPorItem = p; }
    public double getPreco() { return(precoPorItem); }

    public String toString() {
        return String.format("%s %s: %s (%s) %s: %d %s: $%,.2f",
            "Fatura ", "Numero", getParteNumero(), getParteDescricao(),
            "quantidade", getQuantidade(), "Preco por Item", getPreco());
    }
    // metodo obrigatorio para executar o contrato com a interface pagamento
    public double getPagamento() {
        return getQuantidade() * getPreco();
    }
}

```

classe concreta Empregado

public class Empregado implements Pagamento

```

{
    private String matricula;
    private String nome;
    private double salario;

    public Empregado(String matricula, String nome, double salario)
    {
        this.nome = nome;
        this.matricula = matricula;
        this.salario = salario;
    }

    public String getNome() { return nome; }
    public String getMatricula() { return matricula; }
    public double getSalario() { return salario; }
    public void setSalario(double sal) { salario = sal < 0.0 ? 0.0: sal; }

    // metodo OBRIGATORIO pois implementa a interface Pagamento
    public double getPagamento() { return getSalario(); }

    public String toString() {
        return String.format("\n\nMatricula=%s Nome=%s Salario=$%,.2f",
            getMatricula(), getNome(), getSalario());
    }
} // fecha classe empregado

```

Semelhante ao programa apresentado no capítulo sobre Polimorfismo, iremos criar um array para armazenar objetos do tipo Pagamento. Devido a hierarquia construída, podemos armazenar no vetor objetos do tipo Fatura e Empregado, que são derivados da interface Pagamento.

