FATEC Santana de Parnaíba

Curso: Análise e Desenvolvimento de Sistemas
Disciplina: Linguagem de Programação III
Prof. Pedro A.B. Oliveira

Aula 02 - Projeto Folha de Pagamento (POO e Herança)

1 Objetivo

Aplicar conceitos de Orientação a Objetos (Encapsulamento e Herança) para criar um sistema de cálculo salarial com dois tipos de colaboradores: Funcionário Comum e Gerente.

2 Criação do Projeto

1. Abra o NetBeans 8.2.
2. **Arquivo** > **Novo Projeto** > **Aplicação Java**.
3. Nome: `ProjetoFolhaPagamento`.
4. **Desmarque** a opção "Criar classe principal".
5. Clique em **Finalizar**.

3 Criando a Classe Pai (Funcionario)

Esta classe contém os dados comuns a todos.

1. Botão direito em **Pacotes de código fonte** > **Novo** > **Classe Java**.
2. Nome: `Funcionario`.
3. Pacote: `model`.
4. Digite o código abaixo:

```java
package model;

public class Funcionario {
    // Atributos Protected: Para que os Filhos (Gerente) possam ver
    protected String nome;
    protected double salarioBase;

    // Construtor
    public Funcionario(String nome, double salario) {
        this.nome = nome;
        // O 'this' diferencia o atributo do parametro
        this.salarioBase = salario;
    }

    // Regra: INSS de 11% (fixo para exemplo)
    public double calcularINSS() {
        return this.salarioBase * 0.11;
    }

    // Calculo do Liquido
    public double calcularLiquido() {
        return this.salarioBase - calcularINSS();
    }

    // Getters (Para a tela pegar os valores)
    public String getNome() { return nome; }
    public double getSalarioBase() { return salarioBase; }
}
```

Nota sobre o uso do 'this': A palavra reservada `this` é uma referência à **instância atual** da classe. No construtor acima, usamos `this.nome` para indicar que estamos falando do *atributo* da classe, e não do *parâmetro* recebido, que tem o mesmo nome.

4 Criando a Classe Filha (Gerente)

Aqui usamos **Herança**. O Gerente *é um* Funcionário, mas tem Bônus.

1. Novo > Classe Java. Nome: `Gerente`.
2. Pacote: `model`.

```java
package model;

// "extends" indica Heranca
public class Gerente extends Funcionario {

    private double bonus;

    public Gerente(String nome, double salario, double bonus) {
        super(nome, salario); // Chama o construtor do Pai (Funcionario)
        this.bonus = bonus;
    }

    // Polimorfismo: Sobrescrevemos o metodo calcularLiquido
    @Override
    public double calcularLiquido() {
        // Liquido do Gerente = (Salario + Bonus) - INSS
        // Usamos super.calcularINSS() para garantir o calculo base
        return (this.salarioBase + this.bonus) - super.calcularINSS();
    }
}
```

Entendendo a palavra reservada 'super'

O `super` funciona de forma similar ao `this`, mas referencia a **Classe Pai (Superclasse)**. Ele é fundamental na herança para três finalidades:

- **Acessar Construtores:** O comando `super(params)` deve ser a primeira linha do construtor filho para garantir que a parte "pai"do objeto seja construída corretamente.
- **Acessar Métodos Sobrescritos:** Se a classe filha mudou o comportamento de um método (Override), usamos `super.metodo()` para chamar a lógica original da classe pai.
- **Acessar Atributos:** Caso haja conflito de nomes de variáveis entre pai e filho, o `super.variavel` acessa o atributo da classe pai.

5 Criando a Interface (View)

1. Novo > **Form JFrame**. Nome: `TelaFuncionario` , Pacote: `View`
2. Adicione na tela:
   - Labels: "Nome", "Salário Base", "Bônus (se Gerente)".
   - Campos de Texto: `txtNome`, `txtSalario`, `txtBonus`.
   - Checkbox: "É Gerente?"(Nome da variável: `chkGerente`).
   - Botão: "Calcular"(`btnCalcular`).
   - Campos de Texto: `txtResultado` (comece com texto vazio).

6 Programando o Botão (Controller)

Importante: Importando os Pacotes Como a tela está no pacote `View` e as classes `Funcionario` e `Gerente` estão no pacote `model`, elas não são visíveis automaticamente. Vá até o topo do código da tela (logo após `package View;`) e adicione:

```
import model.Funcionario;
import model.Gerente;
```

Código do Botão: Dê duplo clique no botão Calcular e insira a lógica:

```
private void btnCalcularActionPerformed(java.awt.event.ActionEvent evt)
    {
    String nome = txtNome.getText();

    // Conversao de String para double usando a classe Wrapper
    double salario = Double.parseDouble(txtSalario.getText());
    double liquido = 0;

    if (chkGerente.isSelected()) {
        // Se for gerente, pega o bonus e cria um Gerente
        double bonus = Double.parseDouble(txtBonus.getText());
        Gerente g = new Gerente(nome, salario, bonus);
        liquido = g.calcularLiquido();
    } else {
        // Se nao, cria um Funcionario comum
        Funcionario f = new Funcionario(nome, salario);
        liquido = f.calcularLiquido();
    }
    txtSalarioLiquido.setText(String.valueOf(liquido));
}
```

Entendendo a Conversão: primitive vs Wrapper

Na linha `double salario = Double.parseDouble(...)`, note o uso de maiúsculas e minúsculas:

- `double` **(minúsculo):** É o tipo *primitivo* (ocupa 8 bytes). Ele armazena apenas o valor numérico bruto na memória para cálculos rápidos.
- `Double` **(Maiúsculo):** É a classe *Wrapper* (Empacotadora). Ela funciona como uma "caixa de ferramentas"para o tipo primitivo.

Como os campos da tela (`txtSalario.getText()`) retornam sempre um texto (`String`), o Java não consegue calcular diretamente. Por isso, usamos a ferramenta `parseDouble` da classe `Double` para converter esse texto em um número primitivo real.

7 Implementações adicionais

Agora é sua vez de evoluir o sistema. Utilize os conceitos aprendidos para resolver os dois cenários abaixo, comuns em sistemas de RH reais.

7.1 Desafio 1: O Colaborador Horista

Nem todos os funcionários recebem um salário fixo mensal. Existem os contratados por hora (Horistas).

- **Cenário:** O Horista ganha por produção. O salário bruto dele é calculado multiplicando o *Número de Horas Trabalhadas* pelo *Valor da Hora*.
- **Regra de Negócio:** O desconto do INSS (11%) deve ser aplicado sobre o valor total bruto calculado.

O que você deve fazer:

1. Crie a classe `Horista` dentro do pacote `model`, herdando de `Funcionario`.
2. Adicione os atributos específicos: `horasTrabalhadas` (int) e `valorHora` (double).
3. No construtor, utilize o `super`, mas pense: se ele não tem salário base fixo, que valor passar para o pai? (Dica: pode ser 0).
4. Utilize `@Override` no método `calcularLiquido()` para implementar a nova fórmula:

$$((horas \times valor) - 11\%)$$

5. Adapte a Tela (View) para incluir campos para digitar as horas e o valor, caso o usuário selecione a opção "Horista".

7.2 Tratamento de Erros (Try-Catch)

O que acontece se o usuário digitar letras no campo de Salário? O sistema trava.

- **Missão:** Utilize o bloco `try-catch` ao converter os números.
- Caso ocorra erro, exiba uma mensagem amigável usando:

```
1    JOptionPane.showMessageDialog(null, "Digite␣apenas␣n meros!")
       ;
```

7.3 Imposto de Renda

Além do INSS, implemente o desconto do IRRF simplificado. Crie o método `calcularIRRF()` na classe pai com a seguinte regra:

- Salários até R$ 2.000,00: **Isento** (retorna 0).
- De R$ 2.000,01 até R$ 5.000,00: Desconto de **10%**.
- Acima de R$ 5.000,00: Desconto de **20%**.

Atenção: O `calcularLiquido()` deve ser atualizado para:

$$SalarioBase + Bonus - INSS - IRRF$$

7.4 Desafio 2: Folha de Pagamento em Lote (JTable)

Um sistema de RH precisa calcular o salário de vários funcionários de uma vez, e não um por um.

O que você deve fazer:

1. Adicione um componente **Tabela (JTable)** no seu formulário.
2. Adicione um botão "Adicionar à Lista".
3. Crie uma estrutura de lista (ex: `ArrayList<Funcionario>`) para guardar os objetos criados na memória.
4. Ao clicar no botão, o sistema deve:
    - Criar o objeto (Gerente, Funcionario ou Horista).
    - Adicioná-lo na lista.
    - Atualizar a Tabela mostrando: Nome, Tipo (Cargo) e Salário Líquido Final.

Dica para a Tabela: Pesquise sobre a classe `DefaultTableModel` para conseguir adicionar linhas dinamicamente na sua JTable.