**Exercício de Reforço — Sistema de Funcionários e Cálculo de Bonificações**

Implemente um sistema em Java que modele funcionários de diferentes tipos, utilizando **herança, sobrescrita de métodos e polimorfismo**.

**1. Classe Base: Funcionario**

**Atributos protegidos:**

* String nome
* String matricula
* double salarioBase

**Métodos e Construtores:**

* Construtores (vazio e completo).
* Getters, setters e toString().
* Método public double calcularBonus() que retorna salarioBase \* 0.05 (5% do salário base).

**2. Classe Derivada: Gerente**

* Herda de Funcionario.
* **Atributo adicional:** int equipe (quantidade de funcionários que gerencia).
* **Sobrescreva** calcularBonus() para retornar 10% do salário base + 100 por funcionário da equipe.
* **Sobrescreva** toString() para incluir a quantidade de funcionários da equipe.

**3. Classe Derivada: Vendedor**

* Herda de Funcionario.
* **Atributo adicional:** double vendasMensais.
* **Sobrescreva** calcularBonus() para retornar 5% do salário base + 2% do valor das vendas mensais.
* **Sobrescreva** toString() para incluir o valor das vendas mensais.

**4. Classe Derivada: Estagiario**

* Herda de Funcionario.
* **Atributo adicional:** int horasEstagio.
* **Sobrescreva** calcularBonus() para retornar um valor fixo de 300 reais, **independente do salário**.
* **Sobrescreva** toString() para incluir as horas de estágio.

**5. Classe MainFuncionarios**

* Crie alguns objetos de Gerente, Vendedor e Estagiario.
* Crie um método estático exibir(Funcionario f) que mostre o objeto (toString()) de forma polimórfica.
* Armazene todos os funcionários em uma lista List<Funcionario>.
* Percorra a lista e calcule o **total de bonificações**.
* Exiba o resultado no console.

**Objetivos Didáticos**

* Reforçar o uso de **herança** para modelagem de classes.
* Exercitar **sobrescrita de métodos** para alterar comportamento conforme a subclasse.
* Compreender **polimorfismo** ao trabalhar com listas de objetos do tipo Funcionario que se comportam de formas diferentes.

Herança, anulação de método e polimorfismo

**Exercício de Reforço — Sistema de Veículos e Cálculo de Impostos**

Implemente um sistema em Java que modele veículos de diferentes tipos, utilizando **herança, sobrescrita de métodos e polimorfismo**.

**1. Classe Base: Veiculo**

* Atributos protegidos:
  + String placa
  + String modelo
  + double valorBase
* Construtores (vazio e completo).
* Getters, setters e toString().
* Método public double calcularImposto() que retorna valorBase \* 0.02 (2% do valor base).

**2. Classe Derivada: Carro**

* Herda de Veiculo.
* Atributo adicional: int portas.
* Sobrescreva o método calcularImposto() para cobrar **3% do valor base**.
* Sobrescreva toString() para exibir também a quantidade de portas.

**3. Classe Derivada: Moto**

* Herda de Veiculo.
* Atributo adicional: int cilindrada.
* Sobrescreva o método calcularImposto() para cobrar **1% do valor base se cilindrada <= 250**, ou **2% do valor base se cilindrada > 250**.
* Sobrescreva toString() para exibir também a cilindrada.

**4. Classe Derivada: Caminhao**

* Herda de Veiculo.
* Atributo adicional: int eixos.
* Sobrescreva o método calcularImposto() para cobrar **4% do valor base + R$ 500 por eixo**.
* Sobrescreva toString() para exibir também o número de eixos.

**5. Classe MainVeiculos**

* Crie alguns objetos de Carro, Moto e Caminhao.
* Crie um método estático exibir(Veiculo v) que mostre o objeto (toString()) de forma polimórfica.
* Armazene todos os veículos em uma lista List<Veiculo>.
* Percorra a lista e calcule o total de impostos de todos os veículos.
* Exiba o resultado no console.

**Objetivos Didáticos**

* Reforçar o uso de **herança** para modelagem de classes.
* Exercitar **sobrescrita de métodos** para alterar comportamento conforme a subclasse.
* Compreender **polimorfismo** ao trabalhar com listas de objetos do tipo Veiculo que se comportam de forma diferente.

Agregação e composição

**Exercício — Biblioteca Virtual**

Implemente um sistema simples de gerenciamento de uma biblioteca, utilizando **associação entre classes** e explorando métodos de manipulação de listas.

**1. Classe Book**

* **Atributos privados:**
  + title (String)
  + author (String)
  + year (int)
* **Métodos:**
  + Construtores (vazio e completo).
  + Getters e setters.
  + toString() para exibir informações do livro.

**2. Classe LoanItem**

* **Atributos privados:**
  + Book book
  + int days (quantidade de dias do empréstimo)
* **Métodos:**
  + Construtores, getters, setters e toString().

**3. Classe LibraryUser**

* **Atributos privados:**
  + String name
  + List<LoanItem> loans
* **Métodos obrigatórios:**
  + Construtores (inicializando a lista de empréstimos).
  + addLoan(Book book, int days) → adiciona um empréstimo à lista.
  + removeLoan(int id) → remove um empréstimo específico da lista.
  + clearLoans() → remove todos os empréstimos (zera a lista).
  + isEmpty() → retorna true se o usuário não tiver nenhum empréstimo.
  + totalDaysBorrowed() → retorna a soma dos dias de todos os empréstimos.
  + toString() → mostra o nome do usuário e todos os empréstimos atuais.

**4. Classe MainLibrary**

* Crie alguns objetos Book.
* Crie um objeto LibraryUser.
* Adicione empréstimos ao usuário com os livros criados.
* Mostre a lista de empréstimos com System.out.println().
* Remova um empréstimo específico com removeLoan().
* Mostre novamente a lista atualizada.
* Calcule o total de dias emprestados com totalDaysBorrowed().
* Esvazie todos os empréstimos com clearLoans().
* Verifique se a lista está vazia com isEmpty().

**Desafio extra**

1. Crie um método listBooks() em LibraryUser que imprima apenas os títulos dos livros emprestados.
2. Adicione uma regra: um usuário não pode ter mais do que **5 livros emprestados ao mesmo tempo**.

Encapsulamento

Exercício — Cafeteira Elétrica

Implemente uma classe em Java chamada Cafeteira que represente uma cafeteira elétrica simples. O objetivo é praticar encapsulamento e controle de atributos privados.

Requisitos da classe Cafeteira

1. A classe deve ter os seguintes atributos privados:
   * agua (int): quantidade de água disponível em ml (0 até 2000 ml).
   * poCafe (int): quantidade de pó de café em gramas (0 até 100 g).
   * ligada (boolean): indica se a cafeteira está ligada ou desligada.
2. Crie dois construtores:
   * Construtor vazio: inicia a cafeteira desligada, com 0 ml de água e 0 g de pó de café.
   * Construtor com parâmetros: recebe valores iniciais de água, pó e estado (ligada/desligada).
3. Crie os métodos de acesso (get/set) com as seguintes regras:
   * A água deve estar entre 0 e 2000 ml.
   * O pó de café deve estar entre 0 e 100 g.
4. Crie os seguintes métodos de instância:
   * ligar() e desligar().
   * adicionarAgua(int ml): aumenta a quantidade de água, respeitando o limite máximo.
   * adicionarCafe(int gramas): aumenta a quantidade de pó de café, respeitando o limite máximo.
   * prepararCafe(): só pode ser executado se a cafeteira estiver ligada, tiver pelo menos 100 ml de água e 10 g de pó. Ao preparar um café, deve reduzir a água em 100 ml e o pó em 10 g. Caso não haja recursos suficientes, exibir uma mensagem de erro.

Classe Main

* Crie objetos da classe Cafeteira e teste todos os métodos.
* Simule tentativas de preparar café sem água, sem pó ou com a cafeteira desligada.
* Mostre o estado da cafeteira (água, pó e se está ligada) após cada operação.

**Desafio extra:** implemente um método quantosCafesPossiveis() que retorne quantos cafés ainda podem ser preparados com a quantidade de água e pó disponíveis.

**Exercício — Cofrinho Digital**

Implemente uma classe em Java chamada **Cofrinho** que represente um cofrinho digital. O objetivo é praticar **encapsulamento** e a criação de **métodos que controlam o acesso e alteração de atributos privados**.

**Requisitos da classe Cofrinho**

1. A classe deve ter os seguintes **atributos privados**:
   * saldo (double): representa o valor armazenado no cofrinho.
   * fechado (boolean): indica se o cofrinho está fechado ou aberto.
2. Crie dois **construtores**:
   * Construtor vazio: inicializa o cofrinho com saldo 0 e aberto (fechado = false).
   * Construtor com parâmetros: recebe um valor inicial de saldo e se está fechado ou não.
3. Crie os **métodos de acesso (get/set)** com as seguintes regras:
   * O saldo não pode ser negativo.
   * O estado fechado pode ser alterado livremente por meio de abrir() e fechar().
4. Crie os seguintes **métodos de instância**:
   * depositar(double valor): adiciona valor ao saldo **somente se o cofrinho estiver aberto** e o valor for positivo.
   * retirarTudo(): retira todo o saldo do cofrinho (zerando o saldo), mas só funciona se o cofrinho estiver aberto. Deve exibir o valor retirado.
   * abrir(): abre o cofrinho.
   * fechar(): fecha o cofrinho.

**Classe Main**

* Crie objetos da classe Cofrinho e teste todos os métodos.
* Simule depósitos, retiradas e tentativas de uso quando o cofrinho estiver fechado.
* Mostre o saldo atual em cada operação utilizando System.out.println().

**Desafio extra:** implemente um método exibirSaldo() que mostre o saldo atual apenas se o cofrinho estiver aberto. Caso contrário, exiba a mensagem **"Cofrinho fechado. Saldo indisponível."**

Lista 1

**Exercício 1 — Lâmpada Inteligente (Domótica)**

**Objetivo**

Praticar estado do objeto, validações simples e métodos que alteram estado.

**Especificação**

Pacote: exe1

Classe Lampada:

* **Atributos (públicos inicialmente, como no seu exemplo)**  
  boolean ligada;  
  int brilho; // 0 a 100  
  String cor; // "branca", "amarela", etc.  
  String marca;
* **Construtores**
  + Vazio
  + Completo: (String marca, String cor, boolean ligada, int brilho)
* **Métodos**
  + void exibe() — imprime todos os dados.
  + void ligar() — liga se estiver desligada.
  + void desligar() — desliga e zera brilho.
  + void ajustarBrilho(int valor) — só funciona se ligada == true; mantém no intervalo **0–100**.
  + void mudarCor(String novaCor) — altera a cor apenas se ligada == true.

Classe Main:

* Crie uma Lampada com o construtor vazio, atribua valores e teste os métodos.
* Leia do teclado (via Scanner) os dados para uma segunda Lampada.
* Mostre o efeito de ligar/desligar e ajustar brilho.

**Regras de negócio**

* Se tentar ajustarBrilho desligada, não altera (ou apenas ignore a chamada).
* Brilho nunca sai de 0–100.

**Exercício 2 — Conta Bancária (Finanças pessoais)**

**Objetivo**

Praticar operações com números, pré‑condições e efeitos colaterais controlados.

**Especificação**

Pacote: exe2

Classe ContaBancaria:

* **Atributos**  
  String titular;  
  String numero;  
  double saldo;  
  boolean ativa;
* **Construtores**
  + Vazio
  + Completo: (String titular, String numero, double saldoInicial, boolean ativa)
* **Métodos**
  + void exibe() — imprime os dados (mas **não** o número completo? Pode mascarar os 4 primeiros dígitos se quiser).
  + void ativar() / void desativar() — ao desativar, o saldo precisa estar >= 0.
  + void depositar(double valor) — só se valor > 0 e ativa == true.
  + void sacar(double valor) — só se valor > 0, valor <= saldo e ativa == true.
  + void transferir(ContaBancaria destino, double valor) — usa sacar e depois destino.depositar.

Classe Main:

* Crie duas contas (uma por construtor vazio + set de atributos; outra pelo construtor completo).
* Peça via Scanner um valor a depositar, outro para saque e outro para transferir entre as contas.
* Exiba os saldos e estados antes e depois.

**Regras de negócio**

* Não permitir operações em conta inativa.
* Não permitir saldo negativo no saque/transferência.