Notas Práticas e Exercícios 4

Disciplina: Programação Orientada a Objetos

{*Anotações para uso pessoal}

Sumário

1 Exercícios da P1

1.1 Questão 1

```java
//método main
public static void main(String[] args) {
    //instanciando um obj Retangulo passando a base (3), altura (5)
    Retangulo ret = new Retangulo(3,5);
    Operacao op = new Operacao();
    op.aumentarRetangulo(ret, 1, 4);
    System.out.println("O retangulo tem base: "+ret.getBase()+" e altura:
       "+ ret.getAltura());
}
//método aumentarRetangulo dentro da classe Operacao.java
public void aumentarRetangulo(Retangulo ret, int b, int a){
        ret.setBase(ret.getBase() + b);
        ret.setAltura(ret.getAltura() + a);
        ret = null;
        ret  = new Retangulo();
        ret.setBase(5);
        ret.setAltura(8);
}
```

Neste exemplo, temos duas classes: `Retangulo` e `Operacao`. No método `main`, é instanciado um objeto `Retangulo` com base 3 e altura 5 e um objeto `Operacao`. Em seguida, chama-se o método `aumentarRetangulo` da classe `Operacao`, passando o objeto `Retangulo` e os valores 1 e 4.

O método `aumentarRetangulo` aumenta a base e a altura do objeto `Retangulo` passado por 1 e 4, respectivamente. Após isso, o objeto `Retangulo` tem base 4 (3+1) e altura 9 (5+4). O código também cria um novo objeto `Retangulo` com base 5 e altura 8 (linhas 14 a 16). No entanto, essa modificação não afeta o objeto `Retangulo` original no método `main`, pois a referência a este objeto é local para o método `aumentarRetangulo`.

Quando você chama

```
1 System.out.println("O retangulo tem base: " + ret.getBase() + " e altura: "
    + ret.getAltura())
```

no método `main`, ele imprimirá o valor atualizado da base e da altura do objeto `Retangulo` original, que são 4 e 9, respectivamente.

Portanto, a saída será:

```
O retangulo tem base: 4 e altura: 9
```

Isso ocorre porque o método `aumentarRetangulo` modificou o objeto `Retangulo` passado como argumento, mas a criação do novo objeto `Retangulo` com base 5 e altura 8 não afetou o objeto original instanciado e referenciado ainda na main, já que a referência ao objeto é local para o método `aumentarRetangulo`.

1.1.1 Entendendo passo-a-passo o que ocorre na memória

Quando o método `main` é executado, um objeto `Retangulo` é criado na memória com base 3 e altura 5. A referência a esse objeto é armazenada na variável `ret`. Em seguida, um objeto `Operacao` é criado, e o método `aumentarRetangulo` é chamado com a referência do objeto `Retangulo` e os valores 1 e 4.

```
1 Retangulo ret = new Retangulo(3,5);
2 Operacao op = new Operacao();
3 op.aumentarRetangulo(ret, 1, 4);
```

Dentro do método `aumentarRetangulo`, a base e a altura do objeto `Retangulo` passado como argumento são incrementadas pelos valores 1 e 4, respectivamente. Neste ponto, a base e a altura do objeto na memória são atualizadas para 4 e 9.

```
1 ret.setBase(ret.getBase() + b);
2 ret.setAltura(ret.getAltura() + a);
```

Em seguida, a referência local `ret` dentro do método `aumentarRetangulo` é atribuída a `null`, fazendo com que a referência local aponte para nenhum objeto. Isso não afeta o objeto `Retangulo` original na memória nem a referência `ret` a ele no método `main`.

```
1 ret = null;
```

Depois disso, um novo objeto `Retangulo` é criado na memória com base 5 e altura 8. A referência local `ret` dentro do método `aumentarRetangulo` agora aponta para este novo objeto.

```
1 ret  = new Retangulo();
2 ret.setBase(5);
3 ret.setAltura(8);
```

Quando o método `aumentarRetangulo` é encerrado, a referência local `ret` é descartada, e o novo objeto `Retangulo` criado fica sem referências apontando para ele. Eventualmente, esse objeto será coletado pelo coletor de lixo (garbage collector) do Java. No entanto, o objeto `Retangulo` original ainda tem a referência no método `main` e, portanto, não é afetado por essas modificações. Ao imprimir as informações do objeto original, a saída exibirá os valores atualizados de base e altura, que são 4 e 9, respectivamente.

1.2 Questão 3

```java
/* arquivo Carro.java */
public class Carro {
    private String modelo;
    private String placa;
    private double precoDiaria;
    private int chassi;
    private static int serial = 0;
    public Carro(String modelo, String placa, double precoDiaria) {
        modelo = modelo;
        placa = placa;
        precoDiaria = precoDiaria;
        chassi = ++serial;
    }
    public static void setSerial(int newSerial){
         serial = newSerial;
    }
    public int getSerial(){
        return serial;
    }
    public int getChassi(){
        return chassi;
    }
    public String getChassi(){
        return String.valueOf(chassi);
    }
    public static void reajustarDiaria(double valor){
        precoDiaria += valor;
    }
    /* considere já implementados os métodos get e set restante */
}
```

Arquivo com o método main.

```java
public static void main(String[] args) {
    Carro.setModelo("Gol");
    Carro.setPlaca("MLK-1109");
    Carro.setSerial(-1);
    Carro carro1 = new Carro();
    Carro carro2 = new Carro();
    Carro.setSerial(10);
    Carro carro3 = new Carro();
    carro3.reajustarDiaria(10);
    carro3.placa = "NNA-1010";
    System.out.println("O Carro de placa: " + carro1.getPlaca() + " possui
        chassi: " + carro1.getChassi());
    System.out.println("O Carro de placa: " + carro2.getPlaca() + " possui
        chassi: " + carro2.getChassi());
    System.out.println("O Carro de placa: " + carro3.getPlaca() + " possui
        chassi: " + carro3.getChassi());
}
```

Há alguns erros no código fornecido. Abaixo, os erros são listados linha a linha e são propostas soluções para cada um deles.

Erros no arquivo Carro.java

1. Linha 8-10: Erro na atribuição de valores aos atributos da classe. Deve-se utilizar a palavra-chave `this` para referenciar os atributos da classe:

```
1 this.modelo = modelo;
2 this.placa = placa;
3 this.precoDiaria = precoDiaria;
```

2. Linha 20: Erro devido à duplicidade do método `getChassi`. Remova a versão que retorna `String` ou `int`.

3. Linha 23: Erro ao tentar acessar a variável de instância `precoDiaria` de forma estática. O método `reajustarDiaria` deve ser um método de instância e não estático. Remova a palavra-chave `static`.

Erros no arquivo com o método main

1. Linha 2-3: Erros ao tentar chamar os métodos `setModelo` e `setPlaca` na classe `Carro` diretamente. Esses métodos devem ser chamados em um objeto da classe `Carro`, não diretamente na classe. Remova essas linhas.

2. Linha 5-6 e 8: Erro ao tentar criar um objeto `Carro` sem fornecer argumentos. O construtor da classe `Carro` requer 3 argumentos. Não existe construtor vazio. Adicione argumentos apropriados ou crie um construtor vazio.

```
1 Carro carro1 = new Carro("Gol", "MLK-1109", 100.0);
2 Carro carro2 = new Carro("Fusca", "ABC-1234", 80.0);
3 Carro carro3 = new Carro("BMW", "AWC-1234", 1000.0);
```

3. Linha 9: Erro ao tentar acessar o método `reajustarDiaria` de forma estática. Esse método deve ser chamado em um objeto da classe `Carro`. Corrija a chamada:

```
1 carro3.reajustarDiaria(10);
```

4. Linha 10: Erro ao tentar acessar diretamente o atributo `placa` de um objeto `Carro`. O atributo é privado. Use o método `setPlaca` para definir a placa do carro:

```
1 carro3.setPlaca("NNA-1010");
```

1.3 Questão 5

```
1     Retangulo [] rets = new Retangulo[100];
2     for (int i = 0; i < 100; i++){
3         rets[i] = new Retangulo();
4     }
5     rets = null;
```

Neste código, estamos trabalhando com um vetor de objetos do tipo `Retangulo`. Vou explicar passo a passo o que está acontecendo e como a memória alocada será liberada.

1. `Retangulo [] rets = new Retangulo[100];` - Aqui, estamos criando um vetor de referências de objetos `Retangulo`, chamada `rets`, com espaço para 100 elementos. A memória é alocada para armazenar 100 referências a objetos `Retangulo`, mas não para os objetos em si.

2. O loop `for (int i = 0; i < 100; i++)` está iterando de 0 a 99.

3. Dentro do loop, temos `rets[i] = new Retangulo();`. Aqui, estamos criando um novo objeto `Retangulo` e armazenando a referência desse objeto no vetor `rets` na posição `i`. A memória é alocada para cada objeto `Retangulo` criado.

4. Após a conclusão do loop, temos 100 objetos `Retangulo` criados e suas referências armazenadas no vetor `rets`.

5. `rets = null;` - Nesta etapa, estamos atribuindo `null` à variável `rets`. Isso faz com que o vetor `rets` não seja mais referenciada pela variável, o que significa `rets` será recolhido pelo coletor e as referências para os objetos `Retangulo` serão perdidas. Dessa forma, não há mais uma maneira de acessar os objetos `Retangulo` criados anteriormente, fazendo com que esses objetos também sejam recolhidos e liberados pelo coletor.

Após a execução deste código, a memória alocada para os objetos `Retangulo` e o vetor de referências `rets` será liberada eventualmente pelo *Garbage Collector* (Coletor de Lixo) do Java. O *Garbage Collector* é um mecanismo automático que identifica objetos não mais acessíveis (ou seja, sem referências) e os remove da memória, liberando recursos.

É importante observar que o momento exato em que o *Garbage Collector* atua não é previsível, pois depende da implementação e do estado do sistema. No entanto, ao atribuir `null` à variável `rets`, você garante que os recursos alocados serão liberados eventualmente.

Caso alguns objetos `Retangulo` tivessem outra referência além do vetor `rets`, esses objetos não seriam recolhidos. Por exemplo, no código abaixo os objetos referenciados nas posições 4 e 10 do vetor seriam mantidos em memória, pois ainda possuem uma referência (r1 e r2), enquanto os demais seriam recolhidos.

```
1     Retangulo [] rets = new Retangulo[100];
2     for (int i = 0; i < 100; i++){
3         rets[i] = new Retangulo();
4     }
5     Retangulo r1 = rets[4];
6     Retangulo r2 = rets[10];
7     rets = null;
```

2 Composição e Agregação

Para exemplificação adotaremos a classe `A` (por exemplo, conta) como a parte e a classe `B` como o todo (por exemplo, agência). Na composição, a classe `B` (**classe todo**) cria e gerencia um objeto da classe `A` (**classe parte**) como um atributo privado. O objeto da classe `A` é criado quando a instância da classe `B` é criada e tem seu ciclo de vida vinculado à instância da classe B. Ou seja, **não temos um método setter para o `objA` em `B`**. Isso significa que quando a instância da classe `B` é destruída, o objeto da classe `A` também é destruído.

A composição representa uma relação forte entre as classes, indicando que a classe `A` é uma parte integral da classe `B` e não pode existir independentemente dela.

Composição

```java
public class A {
    // Classe A com seus atributos e métodos
}

public class B {
    private A objA = new A(); // Composição com a classe A

    // Outros atributos e métodos da classe B
}
```

Listing 1: Composição entre A e B

Na agregação, a classe `B` tem um atributo privado do tipo A, mas o objeto da classe `A` não é criado dentro da classe B. Em vez disso, o objeto da classe `A` é fornecido externamente, **geralmente por meio de um método "setter"**.

A agregação representa uma relação mais fraca entre as classes, indicando que a classe `A` pode existir independentemente da classe B. A classe `B` apenas mantém uma referência ao objeto da classe `A` e não é responsável por seu ciclo de vida.

Agregação

```java
public class A {
    // Classe A com seus atributos e métodos
}

public class B {
    private A objA; // Agregação com a classe A

    public void setObjA(A objA) {
        this.objA = objA;
    }

    // Outros atributos e métodos da classe B
}
```

Listing 2: Agregação entre A e B

3 Herança com Composição e Agregação

Composição

Ao herdar da classe B, a classe C tem acesso ao objeto referenciado pelo atributo `objA` quando o atributo é declarado como protected. A classe C pode acessar e utilizar esse objeto da classe A, e é possível chamar os métodos públicos e protegidos do objeto `A` a partir da classe C.

No exemplo a seguir, a classe C acessa o objeto da classe `A` referenciado pelo atributo `objA`:

```java
public class A {
    // Classe A com seus atributos e métodos

    public void doSomething() {
```

```
        System.out.println("Doing something in class A");
    }
}

public class B {
    protected A objA new A();

    // Outros atributos e métodos da classe B
}

public class C extends B {
    // Classe C herda de B e tem acesso ao atributo 'b' do tipo A

    public void useA() {
        objA.doSomething();
    }
}

public class Main {
    public static void main(String[] args) {
        C objC = new C();
        objC.useA(); // "Doing something in class A" será printado
    }
}
```

Neste exemplo, a classe C tem acesso ao objeto da classe `A` referenciado pelo atributo `objA`, e pode chamar o método doSomething() do objeto A. O programa principal (main) cria uma instância da classe C e chama o método useA(), que por sua vez chama o método doSomething() do objeto A.

Caso o atributo `b` fosse declarado como private em B, C herdaria os comportamentos e atributos da classe B. No entanto, como o atributo `objA` é privado, ele não pode ser acessado diretamente pela classe C.

Agregação

Se a relação entre as classes `B` e `A` for uma agregação, o objeto da classe `A` não será criado dentro da classe B, mas será fornecido externamente e associado à classe `B` por meio de um método "setter". A classe C ainda herda a estrutura da classe `B` e tem acesso ao atributo `objA` do tipo A, desde que seja declarado como protected.

```
public class A {
    // Classe A com seus atributos e métodos

    public void doSomething() {
        System.out.println("Doing something in class A");
    }
}

public class B {
    protected A objA; // Agregação com a classe A

    public void setObjA(A objA) {
        this.objA = objA;
```

```
    }

    // Outros atributos e métodos da classe B
}

public class C extends B {
    // Classe C herda de B e tem acesso ao atributo 'b' do tipo A

    public void useA() {
        if (b != null) {
            b.doSomething();
        } else {
            System.out.println("A is not set in B");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new A();
        C c = new C();
        c.setObjA(a);
        c.useA(); // "Doing something in class A" will be printed
    }
}
```

Neste exemplo, a relação entre as classes `B` e `A` é de agregação, então a classe `B` não cria um objeto da classe `A` internamente. Em vez disso, o objeto `A` é criado no programa principal (main) e associado à classe `B` (e, por extensão, à classe C que herda de B) usando o método "setter"setObjA(). A classe C ainda pode acessar e utilizar o objeto da classe `A` referenciado pelo atributo `objA`.

Essa abordagem também funcionaria para a composição entre as classes `B` e A, como no exemplo anterior. A classe C herda a estrutura da classe `B` e tem acesso ao objeto da classe `A` referenciado pelo atributo `objA`, independentemente de a relação entre as classes `B` e `A` ser de agregação ou composição.

No exemplo a seguir, a classe `B` tem um atributo privado `objA` do tipo `A` e utiliza agregação. A classe C herda de `B` e, para acessar o objeto referenciado por `objA`, usamos um método "getter" na classe B. O mesmo ocorreria em uma composição quando for usado private para os atributos da classe pai.

```
public class A {
    // Classe A com seus atributos e métodos

    public void doSomething() {
        System.out.println("Doing something in class A");
    }
}

public class B {
    private A objA; // Agregação com a classe A (atributo privado)

    public void setObjA(A objA) {
        this.objA = objA;
    }

```

```
16      // Método "getter" para acessar o atributo privado 'b'
17      public A getB() {
18          return b;
19      }
20
21      // Outros atributos e métodos da classe B
22  }
23
24  public class C extends B {
25      // Classe C herda de B
26
27      public void useA() {
28          A a = getB();
29          if (a != null) {
30              a.doSomething();
31          } else {
32              System.out.println("A is not set in B");
33          }
34      }
35  }
36
37  public class Main {
38      public static void main(String[] args) {
39          A a = new A();
40          C c = new C();
41          c.setObjA(a);
42          c.useA(); // "Doing something in class A" will be printed
43      }
44  }
```

Neste exemplo, a classe `B` utiliza agregação com a classe `A` e mantém um atributo privado b do tipo A. A classe C herda de B, mas não pode acessar o atributo privado b diretamente. Para contornar isso, a classe `B` fornece um método "getter"getB() que retorna o objeto da classe `A` referenciado por b. A classe C utiliza o método "getter"para acessar o objeto da classe `A` e chamar o método doSomething() nele.

4 Abstract e Final

Em Java, os modificadores *abstract* e *final* são usados para impor restrições em classes e métodos.

- **Abstract:** A palavra-chave *abstract* é usada para indicar que uma classe ou método não pode ser instanciado ou implementado diretamente. Esses elementos servem como base para outras classes ou métodos e devem ser estendidos ou sobrescritos para serem utilizados.
  - *Classe Abstract:* Uma classe abstrata não pode ser instanciada diretamente. Ela é projetada para ser herdada por outras classes. Uma classe abstrata pode conter métodos abstratos e não abstratos. As classes filhas que herdam uma classe abstrata devem fornecer implementações para todos os métodos abstratos da classe base, a menos que a classe filha também seja declarada como abstrata.

```
1  // Classe abstrata Animal
2  public abstract class Animal {...}
```

- *Método Abstract:* Um método abstrato é declarado sem implementação na classe base. A implementação desse método deve ser fornecida pelas classes filhas que herdam a classe base. Métodos abstratos só podem existir em classes abstratas.

```
// Método abstrato: as subclasses de Animal devem fornecer uma
    implementação para este método
public abstract void fazerSom();
```

- **Final:** A palavra-chave *final* é usada para indicar que um elemento não pode ser modificado. Quando aplicada a uma classe ou método, impede a herança ou a sobrescrita, respectivamente.

  - *Classe Final:* Uma classe final não pode ser estendida, ou seja, não é possível criar classes filhas a partir dela. Isso é útil quando se deseja criar uma classe imutável ou quando a classe já possui todas as funcionalidades necessárias e não deve ser modificada por herança.

```
// Classe final Veiculo
public final class Veiculo { ...}
```

  - *Método Final:* Um método final não pode ser sobrescrito pelas classes filhas. Isso é útil quando se deseja garantir que a implementação de um método específico não seja alterada por subclasses. Dessa forma, é possível manter um comportamento consistente em toda a hierarquia de classes.

```
// Método final: este método não pode ser sobrescrito pelas
    subclasses de Animal
public final void dormir() {
        System.out.println("O animal está dormindo.");
    }
```

5 Exercícios

1. Classe Animal: Crie uma classe chamada `Animal` com os métodos abstratos `fazerSom()` e `seMovimentar()`. Crie subclasses como `Mamifero`, `Ave`, `Reptil`, etc., e implemente os métodos de acordo com o comportamento específico de cada classe. Use polimorfismo para chamar esses métodos em um array de objetos `Animal`.

2. Calculadora de formas geométricas: Crie uma classe chamada `FormaGeometrica` com métodos abstratos para calcular a área e o perímetro. Crie subclasses como `Circulo`, `Retangulo`, `Triangulo`, etc., e implemente os métodos específicos para cada forma. Utilize polimorfismo para calcular e exibir a área e o perímetro de diferentes formas geométricas.

3. Sistema de pagamento: Crie uma classe chamada `Funcionario` com atributos como nome e salário. Crie métodos abstratos para calcular o salário de diferentes tipos de funcionários, como `Horista`, `Assalariado` e `Comissionado`. Implemente as subclasses e use polimorfismo para calcular o salário de cada funcionário em um array de objetos `Funcionario`.

4. - **Passo 1:** Crie uma classe chamada `ContaBancaria` com os seguintes atributos e métodos:
     - `numeroDaConta`: int
     - `saldo`: double
     - `depositar(double valor)`: Adiciona o valor ao saldo.
     - `sacar(double valor)`: Subtrai o valor do saldo, se possível.

- **Passo 2:** Crie uma classe chamada `ContaPoupanca` que estende `ContaBancaria`:
  - Adicione um atributo `taxaDeJuros`: double
  - Crie um construtor que recebe `numeroDaConta`, `saldo` e `taxaDeJuros` como parâmetros e use a palavra-chave `super` para inicializar os atributos da classe `ContaBancaria`.
  - Adicione um método `aplicarRendimento()`: Multiplica o saldo pelo `taxaDeJuros` e adiciona o resultado ao saldo atual.
- **Passo 3:** Crie uma classe chamada `ContaCorrente` que estende `ContaBancaria`:
  - Adicione um atributo `limite`: double
  - Crie um construtor que recebe `numeroDaConta`, `saldo` e `limite` como parâmetros e use a palavra-chave `super` para inicializar os atributos da classe `ContaBancaria`.
  - Sobrescreva o método `sacar(double valor)`: Subtrai o valor do saldo, se possível, considerando o limite disponível. Isto é, caso o saldo não seja suficiente, use o valor do limite disponível (se for suficiente) e deixe o saldo negativo.
- **Passo 4:** Crie uma classe `Principal` com o método `main`:
  - Instancie um objeto `ContaPoupanca` e um objeto `ContaCorrente`.
  - Utilize casting implícito para armazenar os objetos `ContaPoupanca` e `ContaCorrente` em um array de `ContaBancaria`.
  - Percorra o array e chame o método `sacar()` e `depositar()` de cada conta, utilizando casting quando necessário para acessar os métodos específicos de cada tipo de conta.

Exercício: Implementar, Identificar e Corrigir Erros em um Sistema de Gestão de Funcionários

Neste exercício, você receberá um trecho de código que implementa um sistema de gestão de funcionários usando herança, polimorfismo e casting. Sua tarefa é implementar o código, identificar e corrigir os erros no código para que ele funcione corretamente.

```java
// Arquivo Funcionario.java
public abstract class Funcionario {
    private String nome;
    private double salario;

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

    public final String getNome(){
        return nome;
    }
    public abstract double calcularBonus();
}

// Arquivo Gerente.java
public class Gerente extends Funcionario {
    private double bonus;
    private String area;

    public Gerente(String nome, double salario, double bonus, String area) {
        super(nome, salario);
        this.bonus = bonus;
```

```
    }

    public double calcularBonus() {
        return salario + bonus;
    }

    public String getNome(){
        System.out.println("Nome: " + super.getNome());
        return null;
    }

    public String getArea(){
        return this.area;
    }
}

// Arquivo Vendedor.java
public class Vendedor extends Funcionario {
    private double comissao;

    public Vendedor(String nome, double salario, double comissao) {
        super.Funcionario(nome, salario);
        this.comissao = comissao;

    }

    public double calcularBonus() {
        return salario + comissao;
    }


}

// Arquivo Principal.java
public class Principal {
    public static void main(String[] args) {
        Funcionario[] funcionarios = new Funcionario[3];
        funcionarios[0] = new Gerente("Maria", 5000, 1000);
        funcionarios[1] = new Vendedor("João", 3000, 500);
        funcionarios[2] = new Vendedor("Lucas", 2500, 400);
        funcionarios[3] = new Funcionario("Lucas", 2500, 400);

        for (Funcionario funcionario : funcionarios) {
            System.out.println("Nome: " + funcionario.getNome());
            System.out.println("Area: " + funcionario.getArea());
            System.out.println("Bonus: " + funcionario.calcularBonus());
        }
    }
}
```