Sim, com a adição do arquivo teórico ("PARADIGMAS DE PROGRAMAÇÃO - UCDB.pdf"), podemos refazer a lista de exercícios, fornecendo respostas mais precisas e diretamente apoiadas pelas definições formais dos pilares da Orientação a Objetos.

**Resolução dos Exercícios de Múltipla Escolha (1 a 4)**

**Exercício 1 - O que é encapsulamento em Orientação a Objetos?**

O encapsulamento visa **proteger atributos** e **expõe apenas métodos** para acessar esses dados. Ele envolve o uso de escopos como privado, protegido e público.

**Resposta Correta:** c) **A proteção de dados internos e a exposição controlada por métodos públicos**. *(Isto é exemplificado na prática quando atributos como idade e nome são declarados como private em classes como Pessoa e Bebe, e acessados ou modificados através de métodos públicos como getIdade() e setIdade()).*

**Exercício 2 - A herança é usada para:**

A herança permite **criar nova classe partindo de outra pré-existente**, sendo utilizada para **reuso** e **especialização**.

**Resposta Correta:** a) **Reutilizar código e criar classes especializadas a partir de classes base**. *(Isso é demonstrado quando Advogado, Reu, Juiz e Promotor estendem a classe Pessoa, reutilizando seus atributos e métodos).*

**Exercício 3 - O polimorfismo permite:**

O polimorfismo permite a **sobrescrita** e **sobrecarga de métodos**, ou seja, a **reescrita de métodos de formas diferentes**.

**Resposta Correta:** b) **Que objetos de classes diferentes respondam de maneiras distintas ao mesmo método**. *(Isso é visto na prática, onde classes como Juiz e Promotor sobrescrevem (@Override) os métodos defender() e atacar() herdados de Advogado).*

**Exercício 4 - Em UML, qual símbolo representa um atributo privado?**

O diagrama de classes UML utiliza símbolos específicos para escopos:

* + representa PÚBLICO
* - representa PRIVADO
* # representa PROTEGIDO

**Resposta Correta:** c) **-**.

**Resolução dos Exercícios de Programação (5 a 8)**

As soluções de código a seguir utilizam as classes existentes (Pessoa) e as práticas de programação Java demonstradas nas fontes (uso de *getters* e *setters*, herança e sobrescrita (@Override)).

**Exercício 5 - Crie uma classe Pessoa com atributos nome e idade. Implemente um método apresentar() que exiba: Olá, meu nome é e tenho anos.**

A classe Pessoa já existe e contém os atributos nome e idade privados, com métodos *getters* e *setters*, seguindo o princípio do encapsulamento. Adicionamos o método apresentar():

/\* Baseado em Pessoa.txt \*/

public class Pessoa {

private int idade;

private double peso, altura;

private String nome;

/\* Getters e Setters existentes (omitidos para brevidade) \*/

public int getIdade() { return idade; }

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

public String getNome() { return nome; }

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

// Método solicitado para o exercício 5

public void apresentar() {

System.out.print("Olá, meu nome é ");

System.out.print(getNome()); // Uso do getter, conforme a prática vista em AulaParte2.txt

System.out.print(" e tenho ");

System.out.print(getIdade());

System.out.println(" anos.");

}

}

**Exercício 6 - Crie uma classe Aluno que herde de Pessoa e adicione o atributo curso. Sobrescreva o método apresentar() para mostrar também o curso.**

A classe Aluno deve usar a herança, estendendo Pessoa, seguindo o padrão visto em Reu extends Pessoa e Advogado extends Pessoa. O método apresentar() é sobrescrito (@Override), uma forma de polimorfismo.

/\* Baseado no conceito de Herança e estrutura de Reu.txt \*/

public class Aluno extends Pessoa {

private String curso;

public String getCurso() {

return curso;

}

public void setCurso(String curso) {

this.curso = curso;

}

// Sobrescrita do método apresentar() (Polimorfismo)

@Override

public void apresentar() {

// Uso de getters para acessar atributos herdados (nome e idade)

System.out.print("Olá, meu nome é ");

System.out.print(getNome());

System.out.print(", tenho ");

System.out.print(getIdade());

System.out.print(" anos e faço ");

System.out.print(getCurso());

System.out.println(".");

}

}

**Exercício 7 - Crie uma classe Animal com um método falar(). Crie as subclasses Cachorro e Gato, sobrescrevendo o método para mostrar sons diferentes. Demonstre o polimorfismo no método main().**

Este exercício demonstra o polimorfismo por sobrescrita. A sobrescrita de método é similar ao que é visto nas classes Juiz e Promotor, que modificam o comportamento herdado.

**Classes:**

// Classe Base

public class Animal {

public void falar() {

System.out.println("Som de animal");

}

}

// Subclasse (sobrescreve o método)

public class Cachorro extends Animal {

@Override

public void falar() {

System.out.println("Au Au!");

}

}

// Subclasse (sobrescreve o método)

public class Gato extends Animal {

@Override

public void falar() {

System.out.println("Miau!");

}

}

**Demonstração do Polimorfismo (main):**

/\* Baseado na estrutura de Main e criação de objetos em AulaParte2.txt \*/

public class MainPolimorfismo {

public static void main(String[] args) {

// Objetos criados como tipo base (Animal)

Animal meuCachorro = new Cachorro();

Animal meuGato = new Gato();

// Chamada polimórfica: o comportamento exato (o som) depende do tipo real do objeto.

meuCachorro.falar(); // Executa o método de Cachorro

meuGato.falar(); // Executa o método de Gato

}

}

**Exercício 8 - Crie um programa em Java que use Selection Sort para ordenar um vetor de números reais.**

As fontes fornecidas contêm a estrutura básica para a manipulação de vetores e a leitura de números reais usando Scanner e nextDouble().

No entanto, **as fontes (incluindo o novo arquivo teórico) não contêm o código ou a lógica de implementação do algoritmo de ordenação Selection Sort**.

Abaixo está o esqueleto do programa, baseado nas práticas de I/O e manipulação de vetores observadas.

/\* Baseado em AulaParte1.txt, AulaParte2.txt e AulaParte3.txt \*/

import java.util.Scanner;

public class SelectionSortProgram {

private static final int TAMANHO = 100;

public static void main(String[] args) {

Scanner scanf = new Scanner(System.in);

double vetor[] = new double[TAMANHO];

// 1. Ler 100 números reais (Uso de nextDouble())

System.out.println("Digite 100 números reais:");

for (int i = 0; i < TAMANHO; i++) {

System.out.print("Número " + (i + 1) + ": ");

vetor[i] = scanf.nextDouble();

}

// 2. Exibir o vetor antes da ordenação

System.out.println("\nVetor antes da ordenação:");

exibirVetor(vetor);

// 3. Realizar a ordenação

// \*\*\* A LÓGICA DO ALGORITMO SELECTION SORT NÃO ESTÁ PRESENTE NAS AULAS FORNECIDAS. \*\*\*

// Para que o programa funcionasse, a função selectionSort(vetor) deveria ser implementada aqui.

// 4. Exibir o vetor depois da ordenação

System.out.println("\nVetor depois da ordenação (sem a ordenação aplicada):");

exibirVetor(vetor);

}

// Método auxiliar para exibir o vetor

public static void exibirVetor(double vet[]) {

// Iteração sobre o vetor similar a vetorIteracao e main

for(int i = 0; i < vet.length; i++) {

System.out.print(vet[i] + " ");

}

System.out.println();

}

}