Chamar atenção para:

- Private e o acesso a variáveis nas classes filhas.
- Sobrescrita é só em HERANÇA e sobrecarga na mesma classe.

1. (1,5pt) Identifique e explique o(s) erro(s) nas classes abaixo.

a) Esses dois métodos construtores não estão fazendo corretamente o polimorfismo de sobrecarga, pois a assinatura dos dois métodos é a mesma.

```
public class Ponto2D {
    private double x,y;
    Ponto2D(double x,double y)
    {
        this.x = x;
        this.y = y;
    }
    Ponto2D(double coord1,double coord2)
    {
        this.x = coord1;
        this.y = coord2;
    }
}
```

b) **A classe deveria ser abstract por ter um método abstrato ou remover o abstrato do método. O método abstrato não deveria estar implementado. O método calcular da classe RaizQuadrada não está implementando o polimorfismo de sobrescrita de maneira correta, pois a assinatura está diferente do método na classe-mãe.**

```
public class OperandoUnario {
    private double operando;

    public OperandoUnario(double operando){
        this.operando = operando;
    }

    public abstract double calcular(){
        return 0;
    }
}

public class RaizQuadrada extends OperandoUnario{
    public RaizQuadrada(double operando){
        super(operando);
    }

    @Override
    public double calcular(double operando){
       return Math.sqrt(operando);
    }
}
```

2. (2 pts) Veja o código abaixo. Suponha que o método "sacar" da classe Conta vai ser rescrito de forma a lançar uma exceção criada por você, cuja classe é SaldoNegativoException. O limite é o valor mínimo que a conta pode ficar negativa. A exceção é lançada sempre que, ao tentar realizar um saque, o saldo da conta for inferior ao limite estabelecido.

```
public class Main {
    public static void main(String[] args) {
        Conta contaBancaria = new Conta();
        contaBancaria.depositar(100);
        contaBancaria.setLimite(-100);
        contaBancaria.sacar(1000);
    }
}
```

a) (0,5 pt) Implemente a classe SaldoNegativoException.
Fazer uma classe que estenda da classe Exception, RuntimeException ou IllegalArgumentException.

b) (0,75 pt) Implemente o método sacar que lança a exceção.
O método sacar deve receber um argumento double ou int, verificar se o valor a ser sacado é menor ou igual que saldo + limite e lançar uma exceção caso o valor seja maior e atualizar o valor do saldo.
if (this.saldo - saque < limite

c) (0,75 pt) Rescreva o código acima com o devido tratamento da exceção.
É necessário colocar um bloco try/catch ao redor de sacar. O catch deve esperar a exceção criada (SaldoNegativoException) e informar uma mensagem de erro para o usuário.

3. (1 pt) Sobre a orientação a objeto é correto afirmar:

a) Em uma aplicação que utiliza herança múltipla, uma superclasse deve herdar atributos e métodos de diversas subclasses. Todas as linguagens de programação orientadas a objeto permitem herança múltipla.

b) O polimorfismo associado à herança trabalha com a redeclaração de métodos previamente herdados por uma classe. Esses métodos, embora semelhantes, podem diferir de alguma forma da implementação utilizada na superclasse, podendo ser necessário, portanto, reimplementá-los na subclasse.

c) A visibilidade protegida significa que somente os objetos da classe detentora do atributo ou método poderão enxergá-lo ou utilizá-lo.

d) Em uma relação de herança é possível criar classes gerais, com características compartilhadas por muitas classes. As classes herdadas não podem possuir diferenças.

4. (1,5 pt) Identifique se a frase está correta ou incorreta. Se incorreta, identifique o que está errado.

a) (0,5 pt) A herança permite que os membros de uma classe, chamada de classe-mãe, possam ser reaproveitados na definição de outra classe, chamada de classe-filha. Esta classe-filha tem acesso aos membros públicos e protegidos da classe-mãe. O polimorfismo, associado à herança, permite que métodos abstratos definidos em uma classe abstrata sejam implementados nas classes-filhas, podendo estes métodos, nas classes-filhas, apresentar comportamentos distintos. **Correto**

b) (0,5 pt) Atributos e métodos podem ser reaproveitados através da herança, quando uma subclasse herda as características de uma superclasse. Uma subclasse pode ter acesso aos membros de uma superclasse, **independente do modificador atribuído**. O polimorfismo é um recurso que permite a uma subclasse **reimplementar** os métodos herdados de uma superclasse, sendo este método abstrato ou não **(quando é abstrato não está implementado em primeiro lugar).**

c) (0,5 pt) O conceito de herança estabelece que uma classe possa aproveitar a implementação, definições dos atributos e métodos de uma classe-mãe. A classe-filha pode ter acesso aos métodos e atributos públicos e protegidos da classe-mãe. **O polimorfismo é aplicado ao caso em que existe a necessidade de implementar métodos sobrecarregados, nos quais a classe-filha necessita implementar dois métodos com o mesmo nome e parâmetros diferentes. → Misturou os tipos de polimorfismo.**

5. (4 pts) Você precisa desenvolver um sistema para gerenciar os clientes de pessoa física e jurídica da empresa XYZ. O sistema possui os seguintes requisitos:

i) Um inventário que possui o nome do proprietário do inventário e uma lista de clientes associada a esse inventário. É possível adicionar e remover um cliente no inventário. É possível imprimir a lista de clientes ordenada por CPF/CNPJ. Os clientes que são pessoas físicas devem aparecer na frente dos clientes que são pessoas jurídicas.

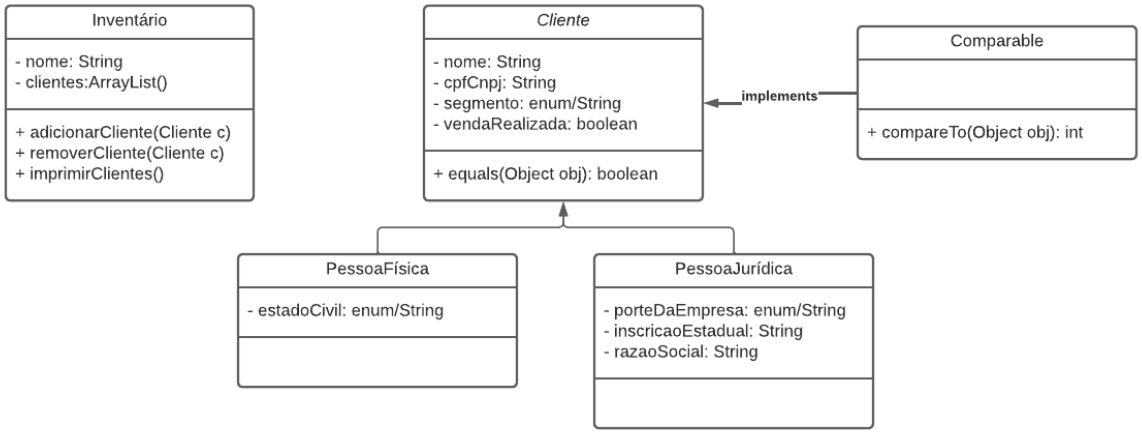
ii) Existem duas categorias de clientes: pessoa física (CPF) ou pessoa jurídica (CNPJ). Todos os clientes registrados devem necessariamente pertencer a uma dessas categorias, não

podendo não pertencer a categoria nenhuma. Nenhuma subcategoria pode ser herdada dessas.

iii) Cada cliente possui o CPF/CNPJ, segmento em que trabalha, nome e se realizou a venda ou não. Se o cliente for uma pessoa jurídica, é preciso registrar o porte da empresa (micro-empresa, pequena, média ou grande), sua inscrição estadual e razão social. Se o cliente for uma pessoa física, é preciso registrar seu estado civil.

Questões:

a) (0,5 pt) Desenhe o diagrama UML representando os requisitos descritos acima. Represente as classes e os relacionamentos de herança, não se preocupe com os relacionamentos de listas e mapas.



b) (1,25 pt) Implemente a classe Inventário com os atributos e métodos correspondentes. Escolha modificadores de acesso adequados para a classe e seus atributos. Não precisa implementar os getters e setters correspondentes, mas indique se é necessário ou não a sua existência. Implemente o construtor correspondente.

Verificação: atributos (0,1 pt), getters/setters (explicação) (0,1 pt) , construtor (0,15 pt), adicionar (0,3 pt), remover (0,3 pt) e imprimir (0,3 pt)

```
import java.util.ArrayList;
import java.util.Collections;

public class Inventario {
   private String nome;
   private ArrayList<Cliente> clientes;

   public Inventario(String nome){
        setNome(nome);
        clientes = new ArrayList<Cliente>();
   }
```

```
    public void adicionarCliente(Cliente cliente){
        clientes.add(cliente);
    }

    public void removerCliente(Cliente cliente){
        clientes.remove(cliente);
    }

    public void imprimirClientes(){
        Collections.sort(clientes);
        for (Cliente cliente: clientes) {
            System.out.println(cliente);
        }
    }
    …
    // Setters e getters
}
}
```

c) (1,25 pt) Implemente a classe Cliente com os atributos e métodos correspondentes. Escolha modificadores de acesso adequados para a classe e seus atributos. Não precisa implementar o construtor, nem os getters e setters correspondentes, mas indique se é necessário ou não a sua existência. Essa classe deve implementar a interface Comparable para sua ordenação. Essa classe deve implementar o método equals para facilitar a remoção de um item.

Verificação: modificador abstract (0,1 pt), implementa interface (0,15 pt), atributos (0,1 pt), getters/setters (explicação) (0,1 pt) , compareTo (0,4 pt), equals (0,4 pt)

```
public abstract class Cliente implements Comparable {
    private String nome;
    private String cpfCnpj;
    private Segmentos segmento;
    private boolean vendaRealizada;

    @Override
    public int compareTo(Object obj){
        Cliente c = (Cliente) obj;
        // Se o tamanho da String é menor, é um CPF. Deve vir
primeiro na ordenação
        if(this.cpfCnpj.length() < c.getcpfCnpj().length()){
            return -1;
        } else if (this.cpfCnpj.length() > c.getcpfCnpj().length()) {
            return 1;
        } else { // Se eles são do mesmo tamanho, basta utilizar a
comparação padrão de Strings
            return this.cpfCnpj.compareTo(c.getcpfCnpj());
        }
    }

    @Override
    public boolean equals(Object obj){
```

```
        Cliente cliente = (Cliente) obj;
        // Mesma lógica do anterior. Para verificar se dois clientes
são iguais,
        // comparamos apenar o atributo cpfCnpj como String.
        return this.cpfCnpj.equals(cliente.getcpfCnpj());
    }
    …
    // toString, setters, getters e construtor
}
```

d) (1 pt) Implemente as classes PessoaFisica e PessoaJuridica com os atributos e métodos correspondentes. Escolha modificadores de acesso adequados para a classe e seus atributos. Não precisa implementar o construtor, nem os getters e setters correspondentes, mas indique se é necessário ou não a sua existência.

Verificação para cada classe: modificador final (0,15 pt), extends classe-mãe (0,15 pt), atributos (0,1 pt), getters/setters (explicação) (0,1 pt)

```
public final class PessoaFisica extends Cliente{
    private EstadoCivil estadoCivil;
    …
    // toString, setters, getters e construtor
}
```

```
public final class PessoaJuridica extends Cliente{
    private PorteDaEmpresa porteDaEmpresa;
    private String inscricaoEstadual;
    private String razaoSocial;
    …
    // toString, setters, getters e construtor
}
```

Programa principal como exemplo:

```
public class Main {
    public static void main(String[] args) {

        Inventario inventario = new Inventario("Laura");

        // Cria dois clientes de exemplo.
        PessoaJuridica pj = new
PessoaJuridica("11111111111111111111", Segmentos.Esportes,
false,
                PorteDaEmpresa.PEQUENA, "123123", "Empresa
LTDA");
        PessoaFisica pf = new PessoaFisica("999999999999",
```

```
Segmentos.Marketing, true, EstadoCivil.Casado);
        // Mesmo CPF, outros atributos diferentes
        PessoaFisica pf2 = new PessoaFisica("999999999999",
Segmentos.Jornalismo, false, EstadoCivil.Solteiro);

        inventario.adicionarCliente(pj);
        inventario.adicionarCliente(pf);

        // Imprime primeiro PF e depois PJ
        inventario.imprimirClientes();

        System.out.println("Pós-remoção:");
        inventario.removerCliente(pf2);
        inventario.imprimirClientes();
    }
}
```