Antes de começar a resolver as questões, leia cuidadosamente as instruções a seguir:

1 - Os exercícios devem ser realizados INDIVIDUALMENTE.

2 - As resoluções dos exercícios devem estar em ARQUIVOS DIFERENTES e uma pasta .zip com todos os arquivos deverá ser entregue até às 23:59 do dia 10/03/2025 pelo Google Classroom.

3 - Nas descrições das questões, todos os atributos, métodos, classes e/ou interfaces em negrito devem ser explicitamente criados com o MESMO NOME nas soluções. Do contrário, você é livre para escolher qual a assinatura dos mesmos.

4 - Os tipos de entrada e saída explicitados devem ser respeitados.

5 - É permitido criar atributos, métodos, classes e/ou qualquer estrutura auxiliar que você considerar necessária para a resolução do problema. Porém, tudo que for pedido deverá obrigatoriamente ser implementado.

6 - Os métodos que forem sobrescritos devem ter a anotação **@Override**.

7 - A correção da lista será feita através da análise individual de cada código, o principal aspecto não se baseia no output correto, mas em uma arquitetura de solução condizente com os princípios da programação orientada a objeto. Logo, utilizem getters, setters, modificações de visibilidade e todos os demais conceitos estudados em sala de aula na medida que julgarem necessário para resolução do problema.

8 - Qualquer dúvida ou inconsistência em relação às questões, deve-se informar imediatamente aos monitores.

9 - Boa sorte!

Questão 1

Fila de Cinema

Você deve criar uma fila que faz com que as pessoas mais velhas fiquem sempre na frente de alguém mais jovem, e que existem 2 tipos de ingresso, um para os adultos e um para as crianças. Use o conceito de polimorfismo para diferenciar o tipo de ingresso que a pessoa terá e, usando os conceitos de generics, faça com que essa fila só aceite objetos que herdam do tipo Pessoa que você deve criar.

Instruções:

- Crie um **enum Ticket** que tenha os dois tipos de ingresso disponíveis: ADULTO e

CRIANÇA;

- Crie uma classe abstrata chamada Pessoa que implementa a interface **Comparable**,

essa classe deve ter um único método abstrato chamado **getTicketType()**, além desse método, a classe deve conter os outros atributos e métodos:

> Atributos:

● **idade(int)**

● **nome (String)**

Obs: use o construtor de Pessoa para inicializar ambos os atributos

> Métodos: ● **int compareTo(Person person)** – o método que deve ser implementado quando você cria uma classe que implementa a interface Comparable (use a idade para implementar esse método)

● **String getName()**

● **int getAge()**

● **toString()** - com o seguinte retorno: name+“: “+age+”[“+getTicketType()+”]”;

• Crie duas classes que estendem a classe Pessoa:

● **Adulto**

● **Criança**

Obs: cada classe deve retornar o tipo de ingresso correspondente com a o seu tipo quando implementar a classe Pessoa.

• Crie a classe **Queue** que deve aceitar apenas objetos que implementam a classe Pessoa

(use os conceitos de generics) e adicione os seguintes atributos e métodos na fila:

> Atributos:

● **pessoas (ArrayList)**

Obs: use o construtor da classe para dizer a capacidade inicial da fila e se nenhum valor for informado, use 10 como um valor padrão.

> Métodos:

● **push(T pessoa)** – lembre-se de fazer com que as pessoas mais velhas sempre

sejam inseridas antes das mais novas

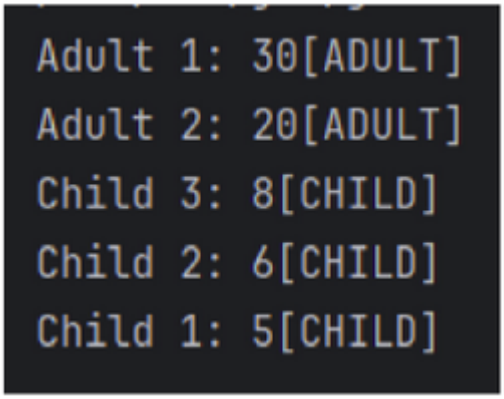
● **T pop()**

● **boolean isEmpty()**

Obs: a intenção não é testar a melhor implementação quando fizer o push na fila, inclusive, recomendo que faça da seguinte forma: sempre insira no array e dê um sort pela ordem reversa, a classe Collections do java pode simplificar o processo!

• Implemente a função main de uma maneira semelhante como a imagem abaixo:

E a saída deve ser semelhante a saída abaixo:



A saída acima foi obtida por causa do método toString da classe Pessoa, por isso, se você implementou de maneira diferente, mas que seja semelhante, não tem problema!

Questão 2

Parabéns, você foi contratado para desenvolver um sistema de gerenciamento de produtos para uma cafeteria. O sistema deve utilizar herança para representar os diferentes tipos de produtos vendidos (por exemplo, bebidas e comidas) e deve implementar tratamento de exceções para lidar com operações inválidas durante a realização de vendas.

Requisitos do Sistema

Classe Base: **Produto**

- Atributos:

  id (int): Identificador único do produto.

  nome (String): Nome do produto.

  preco (double): Preço unitário do produto.

  estoque (int): quantidade do produto no estoque
- Construtor:

  Deve inicializar todos atributos.
- Métodos:

  Getters e setters para cada atributo.

  toString(): Retorna uma representação textual do produto.

Subclasse: **Bebida (herda de Produto)**

- Atributos:

  volume (double): Volume da bebida em mililitros.

- Construtor:

Inicializa os atributos herdados e o atributo específico volume.

- Métodos:

Getters e setters.

toString(): Deve sobrescrever o método para incluir a informação do volume.

Subclasse: **Comida (herda de Produto)**

- Atributos:

ingredientes (List): Lista contendo os ingredientes da comida.

- Construtor:

Inicializa os atributos herdados e a lista de ingredientes.

- Métodos:

Getters e setters.

toString(): Deve sobrescrever o método para incluir os ingredientes.

Exceções Customizadas

ProdutoNaoEncontradoException: Deve estender a classe Exception. Lançada quando uma busca por um produto (por nome) na cafeteria não encontra o produto desejado.

QuantidadeInvalidaException: Deve estender a classe Exception. Lançada quando o valor da quantidade informada para a venda for menor ou igual a zero.

Classe: **Cafeteria**

- Atributos:

produtos (List): Lista contendo todos os produtos disponíveis na cafeteria.

- Método

adicionarProduto(Produto produto): Adiciona um novo produto à lista.

buscarProdutoPorNome(String nome): Retorna o produto cujo nome corresponde ao parâmetro. Se o produto não for encontrado, lança uma ProdutoNaoEncontradoException.

realizarVenda(String nomeProduto, int quantidade): Simula a venda de um produto. Este método deve: Verificar se a quantidade informada é válida (maior que zero); caso contrário, lançar uma QuantidadeInvalidaException.

Buscar o produto pelo nome; se não for encontrado, lançar ou propagar a ProdutoNaoEncontradoException. Se a venda for válida, exibir uma mensagem informando o produto vendido, a quantidade e o valor total da venda.

Classe: **Main**

Função:

Crie um cenário de teste onde:

- São instanciados objetos de Bebida e Comida e estes são adicionados à cafeteria.
- São realizados testes de venda, incluindo:
  - Uma venda válida.
  - Uma tentativa de venda de um produto inexistente.
  - Uma tentativa de venda com quantidade inválida.
  - Uma tentativa de venda com estoque insuficiente.

Utilize blocos try-catch para capturar e tratar as exceções lançadas, exibindo mensagens apropriadas para o usuário.

Questão 3

Você vem pesquisando cada vez mais sobre casas inteligentes e decide fazer um sistema que precisa gerenciar diferentes dispositivos. Para tal, deve criar uma interface **SmartDevice** com os métodos **void turnOn()**, **void turnOff()**, **String getStatus()** e **boolean isOn()**. Em seguida, você compra uma lâmpada (**SmartLamp**), uma fechadura eletrônica (**SmartLock**) e uma televisão (**SmartTV**) que implementam essa interface. Cada dispositivo possui o atributo boolean on. Além disso, a TV também possibilita a mudança de canal **void changeChannel(int channel)**. A lâmpada permite que troque de cor e intensidade **void changeColor(String color, int intensity)** alterando os atributos String color e int intensity.

Por fim, você deve criar um hub central (**SmartHomeHub**) que gerencia os múltiplos dispositivos utilizando uma lista, onde será possível adicionar um dispositivo **void addDevice()**, ligar todos os dispositivos **void turnAllOn()** e **void turnAllOff()**, além de um **void showStatus()**.

Segue o código para a SmartHomeSystem, que testa todo o seu sistema:

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

```
    SmartTv tv = new SmartTv();
    SmartLamp lamp = new SmartLamp();
    SmartLock lock = new SmartLock();

    SmartHomeHub homeHub = new SmartHomeHub();
    homeHub.addDevice(tv);
    homeHub.addDevice(lamp);
    homeHub.addDevice(lock);

    System.out.println("Status inicial:");
    homeHub.showStatus();

    System.out.println("\nLigando todos os dispositivos...");
    homeHub.turnAllOn();
    tv.changeChannel(5);
    lamp.changeColor("blue");

    homeHub.showStatus();

    System.out.println("\nDesligando todos os dispositivos...");
    homeHub.turnAllOff();
    homeHub.showStatus();
  }
}
```

————————————————————

Parte discursiva da questão 3

Por que você utilizou uma interface em SmartDevice? Por que não uma classe ou uma classe abstrata?

Questão 4

Nossa que dia! Hoje é seu primeiro dia como estagiário na empresa Quadrado da Fênix, que decidiu fazer uma parceira com o Porto Digital para elaboração do próximo projeto. Após uma seleção que parecia não acabar nunca e de um teste técnico que fez você duvidar se merecia passar em Algoritmo e estrutura de dados você obteve a excelente notícia que foi aprovad(o/a).

Após tomar seu café da manhã, você abre seu email corporativo e nota um email do seu supervisor, Vicente.

> Olá, parabéns pela aprovação pessoal!
>
> Como vocês devem saber a Quadrado da Fênix está no processo de desenvolver o próximo grande hit, Os heróis do cristal 3! A sua equipe ficou responsável por implementar o comportamento dos monstros de um dos estágios do início do jogo, imagino que a tarefa será simples para vocês e noss(o/a) nov(o/a) estagiári(o/a). Segue o link do repositório do projeto como anexo e lembrem-se de consultar esse excelente manual que um dos nossos colaboradores desenvolveu para ajudar os novatos, recomendo **fortmente darem uma olhada na seção de classe abstrata e interface**: https://docs.google.com/document/d/1z_N_yhuYAdfTeN9b9l5tnPuedni1XrPweu9ThajA-pw/edit?usp=sharing

Você clica no link e é redirecionado para a lista de issues, Vicente aparentemente não bate bem da cabeça, pois só para você há mais de 10 issues para esse ciclo e você é um estagiári(o/a), bem ningúem disse que vida de GameDev seria fácil...

Após analisar cada um dos issues associados você tem a seguinte lista de tarefas misturada com suas anotações:

- Há 3 classes de monstros, as classes são: Dragão, Rato e Lobo.
- Cada turno consiste em checar se o monstro está morto, caso contrário o mesmo ataca.
- Cada criatura deverá alterar seu estado antes de utilizar um ataque que envenene, aterrorize ou sangre um dos membros da equipe do jogador.

Você suspira, isso vai ser mais complicado do que parece...

Cada classe pode vir numa variante distinta identificada pela cor da criatura:

- Azul:

  Dragão: incrementa vida da criatura em 25%.

  Rato: capaz de envenenar a equipe no próximo ataque.

  Lobo: nos dois primeiros turnos realizam investida.

- Verde:

  Dragão: incrementa poder de ataque em 20%

  Rato: primeiro a atacar no início do combate

  Lobo: nos dois primeiros turnos realizam mordida

- Amarelo:

  Dragão: ataca duas vezes por turno

  Rato: capaz de aterrorizar a equipe no próximo ataque.

  Lobo: nos dois primeiros turnos realizam arranhão

- Roxo:

  Dragão: incrementa poder de ataque em 50% e reduz vida em 25%

  Rato: ataca 3 vezes por turno

  Lobo: nos primeiros 3 turnos realizam mordida, invetida e arranhão nessa ordem.

Cada classe tambem tem um conjunto de ataques comuns, mas com comportamento distinto

- Dragões:
  - mordidas de dragões deixam um dos membros da equipe sangrando após ataque.
  - arranhões de dragões removem 50 pontos de vida vezes o nível da criatura.
  - investidas de dragões incrementam poder de ataque em 10%, cumulativo até 50%
- Ratos:
  - mordidas de ratos deixam um dos membros da equipe envenendado.
  - arranhões de rato removem 10 pontos de vida vezes o nível da criatura.
  - investidas incrementam a velocidade da criatura de forma que tal criatura pode atacar duas vezes por turno.
- Lobos:
  - mordidas de lobo deixam um dos membros da equipe aterrorizado.
  - arranhões de lobo removem 20% da vida de um dos membros da equipe.
  - investidas aumentam o poder de ataque da criatura em 10%, cumulativo até 100%.

Após uma longa e árdua inspecção das issues você conseguiu elaborar o design necessário para satisfazer os requisitos, **nome dos atributos e métodos ficam a sua escolha, abaixo apenas sugestão**:

Monstro :

- atributos:
  - vida: Double // ao chegar em zero o monstro morre.
  - estado: String // venenoso, sangramento, aterrorizante, atacando
  - poderDeAtaque: Double // dano causado aos membros da equipe
  - nível: Integer // atributo que escala dano da criatura
- métodos:
  - aindaVive(): Boolean // deve ser chamado no loop principal para determinar fim do combate
  - mordida(): void // ataque comum a todos os monstros, ataca e aplica qualquer efeito do estado anterior, entra no estado de ataque, sobrescreve estado anterior.
  - arranhao(): void // ataque comum a todos os monstros, entra no estado de ataque, sobrescreve estado anterior.
  - investida(): void // ataque comum a todos os monstros, entra no estado de ataque, sobrescreve estado anterior.
  - sangra(): void // altera estado do monstro para sangramento, sobrescreve estado anterior.
  - aterroriza(): void // altera estado do mostro para aterrorizante, sobrescreve estado anterior.
  - enevenena(): void // altera estado do monstro para venenoso, sobrescreve estado anterior.

Colorido:

- métodos:
- alaranjado(): void
- amarelado(): void
- azulado(): void
- esverdeado(): void

Para simular a equipe do jogador, que ainda não foi implementada, utilize a classe Dummy

Dummy

- atributos:
  - estado : String // normal, envenenado, sangrando, aterrorizado, estado padrão é normal
  - vida: Double // utilize java.lang.Double.MAX_VALUE
- métodos:
  - reset(): void // retorna estado de Dummy para os valores setados pelo construtor.

Faltando a implementação das classes de Dragão, Rato e Lobo

————————————————————————————-

Sugestão de parâmetros de combate para os monstros:

- Dragão
  - vida: 9999 + nível/2
  - poderDeAtaque: 999 * $\sqrt{\text{nível}}$
- Rato
  - vida: 1000 + nível
  - poderDeAtaque: 100 + nível^2
- Lobo
  - vida: 5000 + nível^2
  - poderDeAtaque: 500 + nível

Sugestão de como implementar o loop principal:

```
class Game{
  private static Integer turno;

  public static void main(String[] args){
    // inicie uma nova instância de Dummy
    // gere um novo monstro, certifique-se de o monstro ter uma cor aleatória

    // para simular uma batalha execute um laço de até 10 turnos
      // cheque que nessa ordem: monstro está vivo, monstro ataca estando vivo,
lembre-se que ataque deve respeitar as restrições mencionadas previamente, note que
monstro pode atacar mais de uma vez por turno depedendo da cor e classe do mesmo.
      // remova 10% da vida do monstro
      // se monstro morrer a partir desse ponto, saia do laço e encerre a aplicação
      // resete o estado da instância de Dummy

  }
}
```

- os monstros optam por ataque aleatórios, caso não haja conflito com os requisitos mencionados previamentes, utilize a classe Random.
- utilize a classe Random para determinar a cor do monstro, todo monstro gerado deve ter pelo menos uma cor.

Sua nota será dada pelo que foi mencionado acima dos "––", no entanto a sugestão é para facilitar você mesmo testar seu código.

————————————————————

Parte discursiva da questão 4

Existe mais de uma maneira de implementar monstros coloridos, um monstro colorido é qualquer instância da classe Monstro que pode chamar os métodos definidos em Colorido, comente sobre sua escolha de implementação mencionando prós e contras.