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

1 - Os exercícios devem ser realizados INDIVIDUALMENTE

2 - AS resoluções dos exercícios devem estar em ARQUIVOS DIFERENTES. Uma pasta.zip com todos os arquivos deverá ser entregue até as 23:59 do dia 06/08 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 ser 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 progamação orientada a objeto. Logo utilize getters, setters, modificações de visibilidade e todos os demais conceitos estudados em sala de aula na medida que julgarem necessário para a resolução do problema.

8 - Qualquer dúvida ou inconsistência em relação ás questões, entrar em contato urgente com os 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 dois 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 dois tipos de ingresso disponível: ADULTO e CRIANCA
- Crie um a 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
- **Integer idade**
- **String nome**

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

> Métodos:
- **Integer compareTo(Person person)**

O método que deve ser implementado quando você criar uma classe que implementa a interface Comparable (use a idade para implementar esse método)

- **String getName()**

- **Integer getAge()**
- **void toString()**

O método retorna: name+“:”+age+“[“+getTicketType()+”]”;

- Crie duas classes que estemdem a classe Pessoa:
- **Adulto**
- **Criança**

Obs: cada classe deve retornar o tipo de ingresso correspondente com 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

- **ArrayList Pessoas**

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

> Métodos:

- **<T>push(T pessoas)**

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 a imagem abaixo

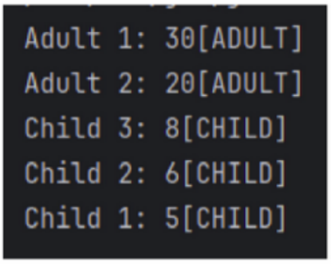
```
public class Main {
    public static void main(String[] args) {
        Queue<Person> queue = new Queue<>( capacity: 5);

        // A linha 12 deve dar erro ao tentar ser compilada, com uma mensagem semelhante a:
        // Type parameter 'java.lang.Integer' is not within its bound; should extend 'person.Person'
        // Queue<Integer> q = new Queue<>();

        queue.push(new Child( age: 5,  name: "Child 1"));
        queue.push(new Adult( age: 30,  name: "Adult 1"));
        queue.push(new Child( age: 6,  name: "Child 2"));
        queue.push(new Adult( age: 20,  name: "Adult 2"));
        queue.push(new Child( age: 8,  name: "Child 3"));

        while (!queue.isEmpty()) {
            Person p = queue.pop();
            System.out.println(p);
        }
    }
}
```

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 - Hamburgueria Premiada

Na Hamburgueria Big Food, o CEO Mr Big, aproveitou o mês da Comida Ogra em Recife e decidiu lançar uma promoção no seu restaurante: o mês do BigFats!

Ao longo de setembro, a cada compra de um refri, batata ou hamburguer, o cliente irá ganhar pontos. Ao final do mês, os 10 clientes com mais pontos ganham um voucher Ogro que pode ser trocado pelo combo Ogro do restaurente, que será revelado apenas no final do mês.

E um detalhe! Clientes VIPs da empresa com cadastro realizado até agosto terão um bônus nos pontos acumulados, dando mais chances de ganhar a competição. Outro detalhe: a cada 10 Reais em compras o cliente ganha 1 ponto extra. Esse ponto extra é inteiro, ou seja:

**R$ 35,00 em compras -> 3 pontos extras**

**6 compras de R$5,00 -> 0 pontos extras (pois individualmente nenhuma atinge o limite inferior de R$10,00)**

A ideia foi um sucesso e foi parar em todos os jornais de Recife! Porém há um problema, Mr Big não sabe programar, então não há possibilidade dele implementar o sistema descrito... até ele conhecer você! Em troca de 2 sachês de Ketchup e um aperto de mão, você aceitou o desafio: montar o sistema do Big Fats!

Usando interfaces e classes abstratas, modele os diferentes tipos de clientes (**Normal** e **VIP**) e crie um sistema com menu que represente os seguintes atributos:

- Preço do hamburguer
- Preço da batata
- Preco do refri
- Bônus VIP **Bônus VIP é um número entre (0,1) exclusivo**

E os seguintes métodos

- Adicionar cliente

Adicionando um cliente, envolve o tipo e nome do mesmo. Lembre-se que para todo e qualquer cliente a pontuação inicial é sempre zero!

- Adicionar compra

Adicionando uma compra, deve-se registrar o tipo do cliente, nome do mesmo e quantidade de produtos, os produtos são os atributos mencionados acima. Lembre-se de computar a pontuação parcial durante aquele mês para o cliente, como no exemplo abaixo:

Tipo: VIP

Nome: João

Hamburguer: 3

Batata: 2

Refri: 0

pontosSemBonus = (precoItem / 10)*qtdItem (divisão sem resto!)

pontosComBonus = Ceil(bonusVip * pontosSemBonus)*qtdItem **(Ceil é a função teto)**

E adicionar aos pontos obtidos em compras prévias.

- Finalizar promoção

Ao finalizar o mês, liste os usuários com mais pontos e imprima se são clientes VIP's ou normais.

## Questão 3 - Posto de Gasolina

Você é responsável pelo sistema de TI de um posto de combustível. O sistema é implementado seguindo o paradigma orientado a objetos e é composto por 3 classes, 1 interface, 1 enum e 2 exceções: **Automovel, Gasolina, Etanol, BombaDeCombustivel, TipoMotor, CombustivelNaoCompativel,CombustivelOverflow** respectivamente.

**TipoMotor** Um enumerador para definir os tipos de combústivel aceitos pelo automóvel, sendo eles:

- **GASOLINA**

suporta apenas gasolina

- **ETANOL**

suporta apenas etanol

- **FLEX**

suporta ambos os tipos de combústives.

**Automovel** classe representando os veículos a serem abastecidos

- Atributos

**double CombustivelAtual** representa a capacidade preenchida atualmente no tanque em litros.

**double capacidadeMaximaTanque** representa a capacidade do tanque em litros, assumindo que o mesmo esteja completo.

**TipoMotor motor** representa o tipo de combústivel suportado pelo motor

- Métodos

**gets e sets** para **combustivelAtual get** para **capacidadeMaximaTanque**

**Gasolina** classe que representa uma bomba de combustivel para gasolina

- Atributos

**double precoLitro** representa o valor do litro da gasolina

- Métodos

implemente interface de **BombaDeCombustivel**

**Etanol** classe que representa uma bomba de combústivel par etanol

- Atributos

**double precoLitro** representa o valor do litro do etanol

- Métodos

implementa a interface de **BombaDeCombustivel**

**BombaDeCombustivel** interface

- Métodos

**public void abastecer(Automovel automovel, double quantidadeLitros)** abastece o tanque de combustivel, aumentando o **combustivelAtual**.

**public void calcularCusto(double quantidadeLitros)** imprime informações do abastecimento como tipo do combústivel, quantidade abastecidae custo do abastecimento

**public void ajustarPreco(double novoPreco)** altera o valor do litro do combustivel

**CombustivelNaoCompativel** exception levantada quanto o automóvel abastecido não é compatível com o tipo de combústivel da bomba.

**CombustivelOverflow** exception levantada quando a quantidade a ser abastecida é maior que o volume vazio da capacidadeMaximaTanque.

**Main**

Implementar os seguintes testes

- Caso de combústivel incompatível
- Caso de abastecimento em excesso
- Abastecimento de gasolina bem sucedido, mas com limite do tanque excedido
- Abastecimento de gasolina após reajuste de preço.

## Questão 4 - Compra Maluca

Esta atividade tem como foco o **levantamento e tratamento de exceções personalizadas** em Java, simulando um carrinho de compras.

Você deverá implementar o código para tratar das seguintes situações por meio de exceções:

- **ItemAleradyExists**

lançada quando há tentativa de adicionar um item com o mesmo nome de outro que já esteja presente no carrinho de compras.

- **NotEnoughSpace**

lançadas quando não há mais espaço para adicionar novos itens (limite de 10 itens diferentes por carrinho)

- **NotFound**

lançada ao tentar remover ou buscar um item que **não existe** no carrinho

- **NullParameter**

lançada quando algum método recebe um **parâmetro nulo**

- A implementação das classes **Cart** e **Buy** estõ incompletas e precisam ser modificas para implementar corretamente o tratamento e levantamento das exceções descritas.
- A classe **Main** já está pronta e não deve ser modificada. Serva **apenas** para facilitar os testes do comportamento do sistema e verificar que as exeções estão sendo levantadas corretamente.

- Todas as exceções mencionadas já estão declaradas como classes.

**Buy** Classe reponsável pelos métodos de compra, adicionar, remover e buscar itens. Deve conter o tratamento das exceções.

**Cart** Classe que gerencia o estado do carrinho e a lógica de manipulação dos itens. Deve conter o levantamento das exceções.