UNIVERSIDADE PRESBITERIANA MACKENZIE
- Faculdade de Computação e Informática –

Ciência da Computação
Paradigmas de Linguagens de Programação – 05N
Prova 1 – 24 de abril 2020
Professor: Fabio Lubacheski

Orientações para realização da prova 1

Esta prova pode ser feita em dupla ou individualmente, basta que somente um dos integrantes entregue um arquivo zipado (.zip) com o código fonte da solução do problema, o arquivo fonte deve conter o seguinte cabeçalho no início do arquivo.

```
/*
Entrega da prova 1 - Paradigmas de Linguagens de Programação – 05N

Nós,

Nome completo e TIA (1º integrante)
Nome completo e TIA (2º integrante)

declaramos que

todas as respostas são fruto de nosso próprio trabalho,
não copiamos respostas de colegas externos à equipe,
não disponibilizamos nossas respostas para colegas externos ao grupo e
não realizamos quaisquer outras atividades desonestas para nos beneficiar ou prejudicar
outros.
*/
```

O programa deve estar bem documentado e implementado na linguagem **Java**, aplicando os conhecimentos sobre o paradigma orientado a objetos vistos na disciplina, tais como: atributos encapsulados (privados), passagem de parâmetro implícita e explicita.

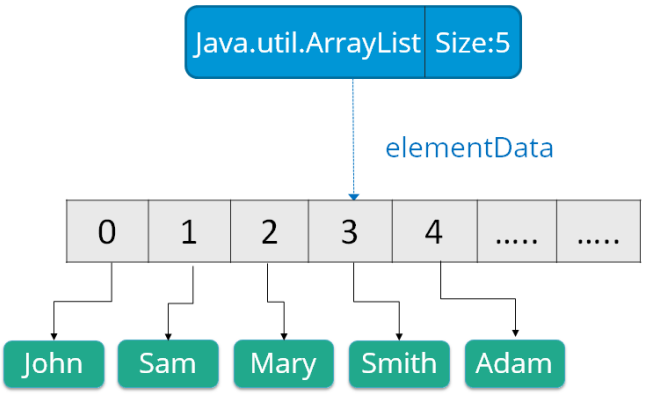
A entrega da prova deve ser feita pelo Moodle (não serão aceitas entregas via e-mail) e será avaliado de acordo com os seguintes critérios:

* Funcionamento do programa;
* O quão fiel é o programa quanto à descrição do enunciado;
* Comentários, legibilidade do código e clareza no nome das variáveis;

Como esta prova pode ser em grupo (até 2 integrantes), evidentemente você pode “discutir”' o problema dado com outros grupos, inclusive as “dicas”' para chegar às soluções, mas você deve ser responsável pela solução final e pelo desenvolvimento do seu programa. Ou seja, qualquer tentativa de fraude será punida com a nota zero. Para maiores esclarecimentos leiam o documento **“Orientações para Desenvolvimento de Trabalhos Práticos”.**

Introdução da prova

No paradigma orientado a objetos é comum trabalharmos não apenas com um objeto, mas com um conjunto de objetos. Uma ***collection*** (coleção) é um objeto que agrupa múltiplos objetos ou tipos primitivos dentro de uma única unidade.



Na linguagem Java, para armazenar uma coleção de objetos, temos a **classe `ArrayList`** (implementada utilizando um vetor), cada objeto no vetor é acessado através de um índice e são chamados **elementos** ou **itens**. A classe `ArrayList` também disponibiliza uma série de métodos para inserção, remoção e consulta dos elementos da coleção, por exemplo:

- **`boolean add(Object elemento)`**: método adiciona um elemento no final da coleção.
- **`void add(int index, Object elemento)`**: método insere um elemento antes do índice informado pelo parâmetro index.
- **`int indexOf(Object element):`** método retorna a posição da primeira ocorrência do elemento contido na coleção.
- **`Object remove(int index)`**: Remove o elemento no índice informado pelo parâmetro index.

Objetivo

O objetivo desta prova é implementar um programa para administrar uma coleção elementos no **Paradigma Orientada a Objetos** utilizando a **linguagem Java**, além disso, você deve implementar duas classes um para **representar os objetos** da coleção de elementos (classe **Ponto**) e outra para armazenar a **coleção de pontos** (classe **listaPonto**). Na implementa dos requisitos dessa prova **não é permitida o uso de classes já prontas** da linguagem Java, tais como: **Point, Point2D, Point3D** ou **ArrayList** (e variações).

A classe **Ponto** , que será usado na prova, é descrita abaixo, e é claro, você pode acrescentar outros métodos na classe para atender a especificação da prova.

```
public class Ponto{
   private int x,y;
   public Ponto(int x, int y) {
        this.x = x;
        this.y = y;
   }
   . . . . . .
}
```

A classe `listaPonto` deve ser implementado de forma que para cada instância da classe `listaPonto` teremos **N** instâncias da classe `Ponto`, para tanto na classe `listaPonto` alocaremos um *container* (=vetor) de objetos da classe `Ponto` com **N** posições, observe que nem sempre teremos **N** objetos no vetor, dessa forma a classe `listaPontos` deverá controlar a quantidade de objetos armazenados (**válidos**) no *container*.

```
public class listaPonto{
    private Ponto pontos[];
    private int validos;
    public listaPonto( int N ){
        this.pontos =new Ponto[N];
        this.validos = 0;
    }
   . . . . . .
}
```

Os elementos no *container* são armazenados de forma **continua e estável**, ou seja, não pode existir **"buracos"** no meio do *container* e as operações para adicionar e de remover um elemento não **alteram a posição relativa** dos elementos no *container*, e caso tenham **sucesso**, deve ser **atualizada a quantidade de elementos válidos** na coleção. No programa que administra a coleção de elementos o usuário pode solicitar as seguintes operações.

1. Adicionar um elemento no final da coleção;
2. Adicionar um elemento em uma posição da coleção;
3. Retornar o índice da primeira ocorrência de um elemento especificado na coleção.
4. Remover um elemento em uma posição na coleção.
5. Calcular a distância dos dois pontos mais distantes na coleção;
6. Retornar uma coleção de pontos contido em uma circunferência.

Para o usuário escolher uma das operações, o seu programa deverá mostrar um **menu de opções**, sendo que para cada opção no menu deverá ser **executada a operação correspondente**, por exemplo, para adicionar um elemento no final da coleção, o usuário deve escolher a opção **1-Adicionar no final**, digitar os valores para x e y do ponto, que em seguida será alocado e passado por parâmetro para método que implementa essa a operação na classe `listaPonto`. O programa **deverá executar continuamente** até o momento que usuário solicite finalizar a aplicação, ou seja, o usuário pode escolher outras operações do **menu de opção** várias vezes. Ao final da execução de cada operação **é impresso** os elementos **válidos na coleção**. A seguir são descritas detalhadamente as operações, para **cada operação** deverá ser implementado **um método** correspondente na classe `listaPonto` e **não é permitido o uso de variáveis globais**:

1. **Adicionar um elemento no final da coleção.**
   Adiciona um novo elemento na primeira posição disponível no final do vetor. Caso o *container* já esteja no limite, ou seja, com **N** elementos, o elemento não é adicionado e a quantidade de elementos válidos não é atualizada, caso seja inserido o elemento, a quantidade de elementos válidos é incrementada. O método responsável em implementar essa operação receberá como parâmetro o novo elemento, ou seja, um objeto da classe `Ponto`.

2. **Adicionar um elemento em uma posição da coleção.**
   A operação de adicionar um elemento em uma determinada posição é mais delicada. Primeiro, precisamos verificar se a posição faz sentido ou não, ou seja, só podemos adicionar um elemento em alguma posição que já estava ocupada ou na primeira posição disponível no final do *container*. Caso a posição seja válida, devemos tomar cuidado para não colocar um elemento sobre outro. É preciso deslocar todos os elementos a "direita" da posição onde vamos inserir o elemento uma vez para a "frente". Isso abrirá um espaço para guardar o novo elemento. Se conseguirmos inserir a elemento a quantidade de elementos válidos no *container* é atualizada. O método responsável em implementar essa operação receberá como parâmetro o novo elemento, ou seja, um objeto da classe **`Ponto`** e o índice onde será inserido o elemento.

3. **Retornar o índice da primeira ocorrência de um elemento especificado na coleção.**

   Nessa operação será informado para método o elemento (`Ponto`) que será buscado no vetor, e em seguida, o método o índice onde o elemento foi encontrado. Caso o elemento não esteja no vetor o método devolve -1.

4. **Remover um elemento em uma posição na coleção;**

   Para essa operação precisamos verificar se a posição está ocupada ou não, se a posição estiver ocupada, então podemos remover o elemento. Para isso basta deslocar os elementos que estavam à direita daquele que removemos uma vez para esquerda e fechamos o "buraco" aberto pela remoção e a quantidade de elementos no vetor é atualizada. O método responsável em implementar essa operação receberá como parâmetro o índice do elemento que será removido.

5. **Calcular a distância dos dois pontos mais distantes na coleção;**

   O método que implementa essa operação deverá retornar o valor da distância entre os dois pontos mais distantes armazenados na coleção, não sendo necessário passar nenhum parâmetro para o método. Considere que para essa função funcionar precisamos ter pelo menos 2 pontos armazenados no *container*.

6. **Retornar uma coleção de pontos contido em uma circunferência.**

   E por fim, temos a operação que encontra pontos que estão contidos dentro de uma circunferência, o método receberá por parâmetro o valor do raio (`double`) e o centro da circunfência (`Ponto`), e devolverá uma nova coleção de pontos, ou seja, um objeto da classe `listaPonto` contendo os pontos cuja distância do centro da circunferência é menor ou igual ao comprimento do raio. No programa cliente é impresso os pontos retornados.