Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática – CEEI
Unidade Acadêmica de Sistemas e Computação – UASC

Disciplina: Laboratório de Programação 2

Laboratório 04

Como usar esse guia:

- Leia atentamente cada etapa
- Quadros com dicas tem leitura opcional, use-os conforme achar necessário
- Preste atenção nos trechos marcados como importante (ou com uma exclamação)!

Sumário

Acompanhe o seu aprendizado

Conteúdo sendo exercitado

- Coleções como objetos que armazenam outros objetos
- Operações básicas sobre coleções: adicionar, remover, pesquisar e iterar
- Coleções em Java: listas, conjuntos, mapas
- O conceito de generics aplicado a coleções Java
- Implementações de coleções baseadas em tabelas hash
- O “contrato” hashcode+equals

Objetivos de aprendizagem

Ao final desse lab você deve conseguir:

- Entender coleções como uma estrutura de dados representada como um objeto de objetos;
- Conhecer o funcionamento e a implementação das operações básicas de estruturas de dados (adicionar, remover, pesquisar e iterar) no contexto de objetos;
- Usar a API de Java de coleções com generics, especialmente para as estruturas ArrayList, HashSet e HashMap.

Perguntas que você deveria saber responder após este lab

- Como se diferenciam as listas, conjuntos e mapas oferecidos em Java? Explique com exemplos.
- Qual a semântica de funcionamento de um ArrayList? Por exemplo, você entende o que acontece internamente quando chamamos a operação add() sobre um objeto do tipo ArrayList?
- Qual a semântica de funcionamento de um HashSet?
- Qual a semântica de funcionamento de um HashMap?
- Considerando as implementações de coleções de Java, em quais situações cada coleção é mais apropriada para ser usada?
- O que é generics? E qual a sua vantagem no contexto de coleções?
- Explique o “contrato” entre equals e hashcode e porque é importante.

Para se aprofundar mais...

- Referências bibliográficas incluem:
  - material de referência desenvolvido por professores de p2/lp2 em semestres anteriores (ONLINE)
  - o livro Use a cabeça, Java (LIVRO-UseCabecaJava)
  - o livro Java para Iniciantes (Livro-JavaIniciantes)
- Tudo que você precisava saber sobre o framework de coleções Java (https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html)
- Entenda a relação entre hashcode e equals (http://blog.algaworks.com/entendendo-o-equals-e-hashcode/)

Introdução

No laboratório anterior, a maioria dos alunos controlaram o número de itens a serem inseridos a partir de um array. Um array é uma estrutura de dados rápida e eficiente, fácil de ser trabalhada mas apresenta suas limitações.

Um problema do array é que, para o mesmo ser eficiente, ele não é capaz de ser expandido ou reduzido. Um array de 10 elementos sempre terá 10 elementos. Existem duas alternativas para essa situação, simplesmente não adicionar o elemento ou expandir o array. Essas duas soluções são descritas no código abaixo:

```
// Alternativa 1: Não adicionar o elemento
private Object[] elementos = new Object[10];

public boolean adicionarElemento(Object elemento) {
    for (int i = 0; i < this.elementos.length; i++) {
        if (this.elementos[i] == null) {
            this.elementos[i] = elemento;
            return true;
         }
    }
    return false;
}
```

```
// Alternativa 2: Expandir o array
private Object[] elementos = new Object[10];

public boolean adicionarElemento(Object elemento) {
    for (int i = 0; i < this.elementos.length; i++) {
        if (this.elementos[i] == null) {
            this.elementos[i] = elemento;
            return true;
        }
    }
    Object[] novoElementos = new Object[this.elementos.length * 2];
    novoElementos[this.elementos.length] = elemento;
    System.arraycopy(elementos, 0, novoElementos, 0, this.elementos.length);
    this.elementos = novoElementos;
}
```

A necessidade de ter estruturas de dados automaticamente expansíveis é tão comum que Java (e demais linguagens) oferece classes básicas para tais entidades. Estas entidades fazem parte da API (interface de programação) de coleções de Java, que fazem parte do pacote de utilitários de java (java.util).

Dizemos assim que tais coleções encapsulam os dados que as compõem e as operações a serem realizadas sobre essa estrutura. Em OO, o conceito de encapsulamento rege todo comportamento e estrutura de código.

Das diferentes estruturas de dados descritas nas coleções de Java, três são de maior importância:

- **Listas:** representa uma coleção de elementos (com possibilidade de repetição) e com ordem
- **Conjunto:** representa uma coleção de elementos (sem repetição) e sem ordem
- **Mapa:** representa uma associação entre dois conjuntos (chave e valor)

Abaixo, descreveremos uma classe de cada tipo de coleções.

1. Listas

Enquanto um array é também uma lista, as listas das coleções de Java oferecem métodos que facilitam a manipulação de tais listas. Um exemplo é o java.util.ArrayList que apresenta os métodos abaixo:

```
private ArrayList listaElementos = new ArrayList();

public boolean adicionarElemento(Object elemento) {
    return listaElementos.add(elemento);
}

public void removerElemento(Object elemento) {
    listaElementos.remove(elemento);
}

public Object pegarElemento(int posicao) {
    return listaElementos.get(posicao);
}

public boolean pertence(Object elemento) {
    return listaElementos.contains(elemento);
}

public int pegaPosicao(Object elemento) {
    return (Integer) listaElementos.indexOf(elemento);
}

public int tamanho() {
    return listaElementos.size();
}
```

Para detectar se um objeto pertence ou não a um ArrayList, java simplesmente faz uso da comparação de igualdade já existente em cada objeto, como mostra o código-fonte da classe ArrayList:

```
public int indexOf(Object o) {
    if (o == null) {
        for (int i = 0; i < size; i++)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = 0; i < size; i++)
            if (o.equals(elementData[i]))
                return i;
```

```
    }
    return -1;
}
```

Listas devem ser usadas quando a ordem dos elementos é importante. Existem três formas distintas de manipular tais coleções:

```
// Alternativa 1: For-each
private ArrayList elementos = new ArrayList();

public int somarElementos() {
    int soma = 0;
    for (Object o : elementos) {
        Integer i = (Integer) o;
        soma += i;
    }
    return soma;
}
```

```
// Alternativa 2: Utilizando um objeto Iterator
private ArrayList elementos = new ArrayList();

public int somarElementos() {
    int soma = 0;
    Iterator itr = elementos.iterator();
    while (itr.hasNext()) {
        Integer i = (Integer) itr.next();
        soma += i;
    }
    return soma
}
```

```
// Alternativa 3: Utilizando o índice
private ArrayList elementos = new ArrayList();

public int somarElementos() {
    int soma = 0;
    for (int i = 0; i < elementos.size(); i++) {
        Integer i = (Integer) elementos.get(i);
        soma += i;
    }
    return soma;
}
```

O iterável, apesar de ser mais complexo de ser usado, é útil especialmente nas situações em que é preciso remover um elemento da lista (itr.remove()).

Importante! GENERICS

Nos exemplos acima, o ArrayList manipulou a classe Object (ou seja, a classe mais básica de Java). Como todos os objetos de Java são do tipo Object, é possível inserir qualquer objeto nessa lista e recuperar tais objetos realizando uma operação de

conversão (cast).

Entretanto, para facilitar a manipulação de objetos compostos (como as listas, conjuntos e mapas), Java criou uma estrutura chamada GENERICS.

Generics é um artifício da linguagem java em que o código feito para classes de composição são feitos de forma genérica, mas especializadas durante a sua codificação. Por exemplo, para criar um ArrayList que apenas opere com Integer, devemos usar a notação descrita a seguir:

```
private ArrayList<Integer> elementos = new ArrayList<>();

public int somarElementos() {
    int soma = 0;
    for (Integer i : elementos) {
        soma += i;
    }
    return soma;
}
```

A importância do uso de generics é que ele permite evitar erros de tipagem. Uma classe com generics só vai permitir a inserção, captura e teste de objetos do tipo indicado no generics.

2. Conjunto

Um conjunto representa uma composição de elementos sem ordem e que evita a repetição. Ao contrário de listas, um conjunto é extremamente rápido para checar a relação de pertence. Vamos ver abaixo um exemplo de uso do conjunto básico de Java, o java.util.HashSet e já aplicando um exemplo de generics.

```
private HashSet<String> palavras = new HashSet<>();

public boolean adicionarElemento(String elemento) {
    return palavras.add(elemento);
}

public boolean pertence(String elemento) {
    return palavras.contains(elemento);
}

public boolean remove(String elemento) {
    return palavras.remove(elemento);
}

public int tamanho() {
    return palavras.size();
}
```

O conjunto é chamado de **HASH**SET pois faz uso da informação de hashcode para escolher onde e como armazenar um elemento dentro do conjunto. Por esse motivo, é extremamente importante que ao implementar um equals próprio para uma classe, o hashcode também seja implementado. Via de regra, é importante garantir que:

Se o1.equals(o2), então o1.hashcode() tem o mesmo valor de o2.hashcode()

Observe que dois objetos podem ter o mesmo hashcode e ser diferentes, no entanto, todo objeto igual deve ter o mesmo hashcode.

Dicas - Como funciona um hashset por baixo dos panos?

Imagine que você tem N posições na memória e quer armazenar um número X entre 0 e N. Uma maneira bem fácil de armazenar esse número, seria colocá-lo na posição X da memória. Assim, para checar se esse número está ou não na memória, basta olhar se a memória na posição X está ocupada.

Ou seja:

1. Pegue o número X, coloque o bit 1 na posição X da memória
2. Se a posição X da memória está ocupada com o bit 1, este número foi armazenado.

Agora imagine que você não quer armazenar números, mas objetos. O mesmo princípio pode ser usado para armazenar tal objeto:

1. Gere um número X entre 0 e N que representa o objeto O
2. Coloque uma referência ao objeto O na posição X da memória

Basicamente, o que o hashcode faz é gerar esse número que representa o objeto O. Entretanto, vários objetos diferentes podem ter o mesmo hashcode. Para resolver o problema de conflito, além do hashcode, o java usa também o equals para verificar se o objeto O que está na memória é igual ao objeto que está sendo passado como parâmetro do *contains*.

Conjuntos podem ser processados da mesma forma de que listas através do for-each ou de objetos Iterators. Um HashSet deve ser usado quando deseja-se armazenar elementos, ignorando a ordem que são inseridos e descartando repetições.

Existem duas formas de processar conjuntos. A primeira alternativa é através de um for-each e a segunda através de um iterator. Veja os exemplos de uso abaixo.

```
// Alternativa 1: For-each
private HashSet<String> palavras = new HashSet<>();

public int contarPalavrasComecandoEmVogais() {
    int conta = 0;
    for (String palavra : palavras) {
        if (palavra.startswith(“a”) || palavra.startswith(“e”) ||
palavra.startswith(“i”) || palavra.startswith(“o”) || palavra.startswith(“u”)) {
            conta += 1;
        }
    }
    return conta;
}
```

```
// Alternativa 2: Utilizando um objeto Iterable
private HashSet<String> palavras = new HashSet<>();

public int contarPalavrasComecandoEmVogais() {
    int conta = 0;
    Iterator<String> itr = palavras.iterator();
    while (itr.hasNext()) {
```

```
        String palavra = itr.next();
        if (palavra.startswith(“a”) || palavra.startswith(“e”) ||
palavra.startswith(“i”) || palavra.startswith(“o”) || palavra.startswith(“u”)) {
            conta += 1;
        }
    }
    return conta;
}
```

3. Mapas

Um mapa é uma estrutura de dados que associa elementos (chaves) a outros elementos (valores). Por exemplo, é possível associar uma string (como matrícula) a um objeto (como aluno). Veja o código abaixo que faz uso da classe mapa básica, o java.util.HashMap:

```
private HashMap<String, Aluno> mapaMatriculaAlunos = new HashMap<>();

public Aluno adicionaAluno(String matricula, Aluno aluno) {
    return this.mapaMatriculaAlunos.put(matricula, aluno);
    // retorna o aluno anteriormente associado a essa matricula, ou nulo se não existia
tal aluno
}

public boolean existeAluno(String matricula) {
    return this.mapaMatriculaAlunos.containsKey(matricula);
}

public boolean existeAluno(Aluno aluno) {
    return this.mapaMatriculaAlunos.containsValue(aluno);
}

public Aluno recuperaAluno(String matricula) {
    return this.mapaMatriculaAlunos.get(matricula);
}

public Aluno remove(String matricula) {
    return this.mapaMatriculaAlunos.remove(matricula);
}

public int numeroDeAlunos() {
    return this.mapaMatriculaAlunos.size();
}
```

Comumente, para processar um mapa existem três formas diferentes de fazer isto, dependendo do seu objetivo.

```
// Alternativa 1: Pelas chaves
private HashMap<String, Aluno> mapaMatriculaAlunos = new HashMap<>();

public void alterarTurma(String turma) {
    for (String matricula : this.mapaMatriculaAlunos.keySet()) {
        Aluno aluno = this.mapaMatriculaAlunos.get(matricula);
```

```
        aluno.setTurma(turma);
    }
}
```

```
// Alternativa 2: Pelos valores
private HashMap<String, Aluno> mapaMatriculaAlunos = new HashMap<>();

public void alterarTurma(String turma) {
    for (Aluno aluno : this.mapaMatriculaAlunos.values()) {
        aluno.setTurma(turma);
    }
}
```

```
// Alternativa 3: Pelas chaves e valores
private HashMap<String, Aluno> mapaMatriculaAlunos = new HashMap<>();

public void alterarTurma(String turma) {
    for (Entry<String, Aluno> entry : this.mapaMatriculaAlunos.entrySet()) {
        Aluno aluno = entry.getValue();
        aluno.setTurma(turma);
    }
}
```

Para o objetivo acima (setar a turma de todos os alunos) o uso do processamento dos valores (alternativa 2) teria sido a melhor alternativa.

Controle de Alunos

Neste laboratório, construiremos um sistema que será a base do controle de alunos de Programação II. Como sistema, deve ser possível primeiramente cadastrar e consultar alunos. Cada aluno tem uma matrícula, nome e curso (todos Strings). Cada consulta deve procurar o aluno a partir de sua matrícula. O aluno é identificado unicamente pela matrícula e não deve ser possível alterar os dados dos alunos uma vez cadastrado.

Os alunos podem se juntar em grupos de estudo. Cada grupo de estudo tem um tema (descrito por uma String, como por exemplo o tema “Coleções”) e é formado por um conjunto de alunos. O aluno pode estar em mais de um grupo de estudo. Deve ser possível colocar alunos em tais grupos a partir de sua matrícula. Cada grupo é identificado unicamente a partir do nome do grupo.

Por fim, durante as aulas, os alunos costumam fazer exercícios em quadro e responder perguntas feitas pelo professor. O professor deseja manter um registro dos alunos que responderam tais perguntas. É possível que o mesmo aluno seja questionado mais que uma vez e preciso registrar a ordem que os mesmos responderam.

O programa deve iniciar com um menu como indicado abaixo.

```
(C)adastrar Aluno
(E)xibir Aluno
(N)ovo Grupo
(A)locar Aluno no Grupo e Imprimir Grupos
(R)egistrar Aluno que Respondeu
(I)mprimir Alunos que Responderam
(O)ra, vamos fechar o programa!

Opção>
```

Ao longo da execução do sistema, caso o usuário passe uma entrada inválida (nula ou vazia) o sistema deve lançar uma exceção e ser encerrado. Após cada ação do sistema, o menu deve ser novamente impresso e uma opção deve ser novamente selecionada.

Abaixo, seguem mais detalhes do que deve ser feito neste laboratório.

1. Cadastrar Aluno

Ao selecionar a opção para cadastrar um aluno, o sistema deve pedir informações como a matrícula, nome e curso.

```
Matrícula: 250
Nome: Gabriel Reyes
Curso: Computação
CADASTRO REALIZADO!
```

Quando a matrícula cadastrada já existe, o aluno não deve ser cadastrado e uma mensagem deve ser exibida como mostra o exemplo abaixo.

```
Matrícula: 250
Nome: Mei-Ling Zhou
Curso: Computação
MATRÍCULA JÁ CADASTRADA!
```

2. Consultar Aluno

Para consultar alunos, apenas a matrícula deve ser pedida como entrada e a saída abaixo deve ser produzida em caso do aluno daquela matrícula existir:

```
Matrícula: 250

Aluno: 250 - Gabriel Reyes - Computação
```

Caso o aluno não exista, uma mensagem indicando que o aluno não foi cadastrado deve ser exibida:

```
Matrícula: 2500

Aluno não cadastrado.
```

3. Cadastrar Grupo

Para cadastrar um grupo basta colocar um nome associado a este grupo. Maiúsculas e minúsculas são indistintas (i.e. o grupo “Listas” é igual ao grupo “listas”). Uma mensagem de sucesso deve ser exibida como a indicada abaixo:

```
Grupo: Listas
CADASTRO REALIZADO!
```

Caso o usuário cadastre um grupo com um nome já existente, a mensagem “GRUPO JÁ CADASTRADO!” deve ser exibida.

4. Alocar Alunos em Grupos

Ao selecionar a opção (A) correspondente a alocação, o usuário é perguntado se ele deseja alocar ou imprimir um grupo.

```
(A)locar Aluno ou (I)mprimir Grupo? A
```

Caso o usuário selecione alocar, ele segue para o pedido de entrada apresentado abaixo.

Para alocar alunos em grupos, basta informar uma matrícula e nome do grupo para realizar a alocação. Em caso de sucesso, a mensagem "ALUNO ALOCADO" deve ser exibida em seguida. Caso o aluno já esteja no grupo, a mesma mensagem pode ser repetida e nada é alterado na sua alocação.

```
Matricula: 250
Grupo: LISTAS
ALUNO ALOCADO!
```

Caso o aluno não esteja cadastrado ou o grupo não esteja cadastrado, as mensagens: “Aluno não cadastrado.” e “Grupo não cadastrado.” devem ser impressas, respectivamente.

5. Imprimir Grupos

Caso o usuário peça para imprimir um grupo, o sistema deve pedir como entrada o nome do grupo a ser impresso e exibir o nome do grupo (como inserido originalmente) e os alunos que fazem parte desse grupo. A ordem que os alunos são impressos não é importante para o sistema.

```
Grupo: listas

Alunos do grupo Listas:
* 250 - Gabriel Reyes - Computação
* 200 - Angela Ziegler - Medicina
```

Caso o grupo de estudo não esteja cadastrado, a mensagem “Grupo não cadastrado.” deve ser exibida em tela.

6. Cadastrar Alunos que Respondem Questões no Quadro

Para cadastrar os alunos que respondem questões, o usuário basta apenas colocar a matrícula do aluno que respondeu uma questão. Novamente a mensagem “Aluno não cadastrado.” deve ser exibida caso uma matrícula não tenha aluno associado. Abaixo um exemplo de uma alocação de sucesso.

```
Matricula: 250
ALUNO REGISTRADO!
```

7. Imprimir Alunos que Respondem Questões no Quadro

Por fim, ao imprimir os alunos que responderam questões, a lista de alunos deve ser impressa juntamente com a ordem que os alunos responderam questões, como indicado no exemplo abaixo.

```
Alunos:
1. 250 - Gabriel Reyes - Computação
2. 200 - Angela Ziegler - Medicina
3. 250 - Gabriel Reyes - Computação
4. 201 - Torbjorn Lindholm - Engenharia Mecanica
5. 201 - Torbjorn Lindholm - Engenharia Mecanica
```

Entrega

Faça o programa de Controle de Alunos que atenda as funcionalidades descritas anteriormente. Lembre-se de lançar as exceções apropriadas e considerar todos os fluxos alternativos de execução (ex.: o que acontece quando tenta-se imprimir um grupo que não existe, etc.).

É importante que o código esteja devidamente documentado (com javadoc).

Ainda, você deve entregar um programa com testes para as classes com lógica testável (isso exclui a classe do main(...)). Tente fazer um programa que possa ser devidamente testado e que explore as condições limite.

Para a entrega, faça um **zip** da pasta do seu projeto. Coloque o nome do projeto para: LAB4_SEU_NOME e o nome do zip para LAB4_SEU_NOME.ZIP. Exemplo de projeto: LAB4_MATHEUS_GAUDENCIO.ZIP. Este zip deve ser submetido pelo Canvas.

Seu programa será avaliado pela corretude e, principalmente, pelo DESIGN do sistema. É importante:

- Usar nomes adequados de variáveis, classes, métodos e parâmetros.
- Fazer um design simples, legível e que funciona. É importante saber, apenas olhando o nome das classes e o nome dos métodos existentes, identificar quem faz o que no código.