

Utilização de Coleções em Java

Exercício: Resolução

Vinicius Takeo Friedrich Kuwaki

Universidade do Estado de Santa Catarina

Seções

Exemplo

Resolução

Métodos úteis

Exercício

Resolução

Exemplo

Crie uma função que armazene em um Map a tabuada do número 1 ao número 10 e exiba no console a tabuada do número n armazenada em um List.

Seções

Exemplo

Resolução

Métodos úteis

Exercício

Resolução

- Vamos utilizar um List no nosso exercício para armazenar os valores de 1 a 10 da tabuada de um número n ;
- Instanciaremos esse List como um ArrayList, pois ele tem melhor desempenho nas consultas aos dados armazenados;
- Vamos criar uma classe chamada **Principal** e importar a interface List e a classe ArrayList:

```
import java.util.List;  
import java.util.ArrayList;
```

- Vamos utilizar também um Map em nosso exercício;
- Como o próprio nome já diz, um Map é um mapa que associa um objeto a uma chave única de acesso/busca;
- Logo, os Maps são compostos de pares ordenados $\langle K, V \rangle$, onde K corresponde a chave (key) e V corresponde ao valor (value) associado a respectiva chave;
- Aqui, o Map será utilizado para armazenar as tabuadas de 1 a 10 dos números 1 a 10;
- Assim como o List, o Map é uma interface que necessita ser instaciado por uma classe que o implementa;
- Nesse exercício vamos instanciar o Map como um HashMap;

- Assim como para o List e ArrayList, vamos importar a interface Map e a classe HashMap:

```
import java.util.Map;  
import java.util.HashMap;
```

- Importadas as bibliotecas que utilizaremos, vamos construir a função que retorna a List contendo a tabuada de um número n ; //
- Faremos tudo dentro da classe Principal que também contém o método **main()**;
- Nosso método, que será estática, recebe como parâmetro o número inteiro n do qual queremos saber a tabuada e retorna um List de inteiros com os valores da tabuada de 1 a 10;
- Coleções utilizam apenas objetos, como inteiros são tipos primitivos, utilizaremos seu Wrapper **Integer**:

```
public static List<Integer> tabuada(int n){  
    ...  
}
```


- Vamos primeiro instanciar um objeto do tipo ArrayList;
- Através de polimorfismo, ele será um List:

```
public static List<Integer> tabuada(int n){  
    List<Integer> tabuada = new ArrayList<Integer>();  
}
```

- Vamos iterar sobre uma variável i de 1 até 10 e adicionar no List a multiplicação de i por n ;
- Para adicionar um valor no List, utilizamos o método **add()**:

```
public static List<Integer> tabuada(int n){  
    List<Integer> tabuada = new ArrayList<Integer>();  
  
    for (int i = 1; i <= 10; i++) {  
        tabuada.add(i * n);  
    }  
  
    return tabuada;  
}
```

Resolução

- Agora, vamos declarar um método **main()**;
- Dentro do método **main()** vamos instanciar o **HashMap**;
- Novamente, através de polimorfismo ele será um **Map**;
- A chave será um **Integer** e o valor será uma lista de inteiros, contendo a tabuada da respectiva chave;
- Logo o valor será: **List<Integer>**;

```
public static void main(String [] args) {  
    Map<Integer , List<Integer>> todasAsTabuadas = new HashMap<Integer , List<Integer>>();  
}
```

- Diferente do List que possui um método **add()**, o Map possui um método **put(chave, valor)** para inserir os pares $\langle K, V \rangle$;
- Vamos adicionar dentro do nosso Map a tabuada de 1 a 10;
- Para isso, utilizaremos o método **put()** e a nossa função **tabuada()**, iterando um valor i de 1 a 10:

```
for (int i = 1; i <= 10; i++) {  
    todasAsTabuadas.put(i, tabuada(i));  
}
```

- Agora vamos exibir os valores das tabuadas armazenadas no Map;
- O Map possui um método chamando **forEach()** que recebe um par ordenado (K, V) e um trecho de código;
- Esse trecho de código será executado para cada par ordenado (K, V) do Map;
- Vamos primeiro chamar o método e depois construir o código de manipulação do Map:

```
todasAsTabuadas.forEach(  
    (chave, tabuada) -> {  
        // Código a ser executado para cada par ordenado (chave, tabuada)  
    }  
);
```

- Para cada par ordenado vamos imprimir a chave e os valores da tabuada associados a essa chave e que estão armazenados no List;
- Vamos começar pela chave, que em nosso exemplo chamamos ela de “chave”:

```
todasAsTabudas.forEach(  
    (chave, tabuada) -> {  
        System.out.print("Tabuada de " + chave + ": ");  
    }  
);
```

- Agora vamos percorrer a lista que chamamos de “tabuada”, imprimindo o valor de cada posição da lista:

```
todasAsTabudas.forEach(  
    (chave, tabuada) -> {  
        System.out.print("Tabuada de " + chave + ": ");  
        for (int x : tabuada) {  
            System.out.print(x + " ");  
        }  
    }  
);
```

- Finalizado o **forEach()** vamos executar a **main()**:

```
Tabuada de 1: 1 2 3 4 5 6 7 8 9 10
Tabuada de 2: 2 4 6 8 10 12 14 16 18 20
Tabuada de 3: 3 6 9 12 15 18 21 24 27 30
Tabuada de 4: 4 8 12 16 20 24 28 32 36 40
Tabuada de 5: 5 10 15 20 25 30 35 40 45 50
Tabuada de 6: 6 12 18 24 30 36 42 48 54 60
Tabuada de 7: 7 14 21 28 35 42 49 56 63 70
Tabuada de 8: 8 16 24 32 40 48 56 64 72 80
Tabuada de 9: 9 18 27 36 45 54 63 72 81 90
Tabuada de 10: 10 20 30 40 50 60 70 80 90 100
```


Seções

Exemplo

Resolução

Métodos úteis

Exercício

Resolução

Métodos úteis - List

- **int size():** retorna o numero de objetos na lista;
- **boolean add(T objeto):** adiciona um objeto ao final da lista;
- **boolean add(int posicao, T objeto):** adiciona um objeto em um posição específica da lista;
- **boolean remove(T objeto):** remove um objeto da lista. A classe ao qual pertence o objeto deve implementar o método equals();
- **boolean remove(int posicao):** remove um objeto em uma posição específica da lista;
- **T get(int posicao):** retorna o objeto da posição x da lista;
- **boolean contains(T objeto):** retorna true caso o objeto esteja dentro da lista e false caso contrário. A classe ao qual pertence o objeto deve implementar o método equals();

- **int size():** retorna o numero de objetos na lista;
- **boolean containsKey(K chave):** retorna true caso essa chave exista dentro do map e false caso contrário. A classe ao qual pertence o objeto deve implementar o método equals();
- **boolean remove(T objeto):** remove um objeto (K ou V) do map. A classe ao qual pertence o objeto deve implementar o método equals();
- **V get(K chave):** retorna o valor associado a chave K;
- **boolean put(K chave, V valor):** adiciona um par ordenado (K,V) no Map;

Seções

Exemplo

Resolução

Métodos úteis

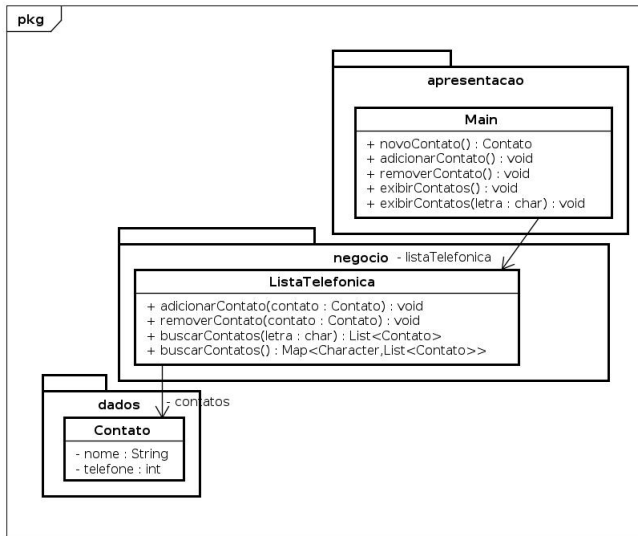
Exercício

Resolução

Exercício

Utilizando o framework de coleções, implemente o diagrama de classes a seguir:

Exercício



Exercício

- A classe ListaTelefonica deve manter uma lista de contatos telefonicos;
- O método **exibirContatos()** da classe Main deve exibir todos os contatos, ordenados de acordo com a primeira letra do nome;

- Por exemplo:

A:

- André: 983748574

- Ana: 97364985

B:

- Bianca: 947346543

C:

D:

- Daniel: 973648374

E:

F:

Exercício

G:

H:

...

Y:

Z:

- Os contatos não necessariamente precisam ser exibidos em ordem alfabética, apenas devem estar agrupados de acordo com a inicial do seu primeiro nome.
- Utilize HashMap para indexar o contatos pela inicial.
- O método **removerContato()** da classe Main deve requisitar ao usuário a inicial do contato que ele deseja remover, após o usuário entrar com ela, deve ser exibido uma lista contendo todos os contatos que possuem essa inicial. O usuário deverá então escolher um.

Seções

Exemplo

Resolução

Métodos úteis

Exercício

Resolução

Resolução

- Primeiro iremos definir a classe Contato;
- Esta possui dois atributos: nome e telefone;

```
public class Contato {  
    private String nome;  
    private int telefone;
```

- Também será definido um método **toString()** que será usado futuramente para exibir no console as informações do contato:

```
    public String toString() {  
        return this.nome + " — " + this.telefone;  
    }
```

- Agora vamos implementar a classe que representa a lista telefonica;
- Utilizaremos as coleções HashMap e LinkedList para mantermos os contatos em memória;

- Para isso precisamos importar as classes necessárias:

```
package negocio;  
  
import java.util.List;  
import java.util.LinkedList;  
  
import dados.Contato;  
  
import java.util.Map;  
import java.util.HashMap;
```

- Utilizaremos o HashMap onde a chave é inicial do nome e o valor é uma LinkedList de contatos, que obviamente possuem o nome que começa com essa inicial;
- HashMap utiliza apenas objetos, portanto teremos que utilizar os Wrappers. Logo, é necessário usar a classe Character ao invés do tipo primitivo `char`:

```
public class ListaTelefonica {  
    private Map<Character, List<Contato>> contatos = new HashMap<Character, List<  
        Contato>>();
```

- Nosso construtor irá inicializar o HashMap;
- Teremos um laço de repetição que criará 26 listas de contatos para cada letra do alfabeto;
- Como na tabela ASCII, as letras do alfabeto (maiúsculas) vão de 65 a 90, nosso for também irá;
- Utilizaremos o método **put()** do HashMap para adicionar um objeto <Chave,Valor>:

```
public ListaTelefonica() {  
    for (char i = 65; i < 91; i++) {  
        List<Contato> lista = new LinkedList<Contato>();  
        contatos.put(i, lista);  
    }  
}
```

- Para adicionar um contato, precisamos acessar a inicial do contato que é passado como parâmetro;
- Logo, usaremos o método **toUpperCase()** da classe String para obter o nome do contato com todas as suas letras em maiúsculo;
- Após isso, utilizaremos o método **charAt()** da classe String para obter a primeira letra do nome;

- Conhecendo a primeira letra do nome, passaremos ela como chave para o método **get(chave)** do HashMap para obter o objeto associado a essa chave. Em nosso caso, a lista de contatos cujo nome começa com essa letra;
- Como o método irá retornar uma List, utilizaremos o método **add()** para adicionar um contato nessa lista:

```
public void adicionaContato(Contato contato) {  
    String nome = contato.getNome().toUpperCase();  
    contatos.get(nome.charAt(0)).add(contato);  
}
```

- O método de remover contato segue o mesmo princípio, mas ao invés de utilizar o método **add()** utilizaremos o método **remove()**:

```
public void removeContato(Contato contato) {  
    String nome = contato.getNome().toUpperCase();  
    contatos.get(nome.charAt(0)).remove(contato);  
}
```

- Criaremos também dois métodos para buscar contatos, um deles irá retornar uma List e o outro irá retornar um Map:
- O que retorna a List pede como parâmetro a letra associada a essa lista:

```
public List<Contato> buscarContatos(char letra) {  
    return contatos.get(letra);  
}
```

- Já o que retorna o Map não pede nenhum parâmetro:

```
public Map<Character, List<Contato>> buscarContatos() {  
    return contatos;  
}
```

- Definidos os métodos da classe ListaTelefonica, vamos fazer a interface com o usuário agora;
- A classe Main terá uma instância de uma lista telefônica e uma da classe Scanner para realizar a leitura de dados do usuário:

```
package apresentacao;  
  
import java.util.Scanner;  
  
import dados.Contato;  
import negocio.ListaTelefonica;  
  
public class Main {  
  
    private static Scanner s = new Scanner(System.in);  
    private static ListaTelefonica lista = new ListaTelefonica();  
}
```


Resolução

- Criaremos um método apenas para instanciar um Contato de acordo com os dados passados pelo usuário:

```
public static Contato novoContato() {  
  
    System.out.println("Digite o nome do contato:");  
    String nome = s.nextLine();  
    nome = s.nextLine();  
  
    System.out.println("Digite o telefone do contato:");  
    int telefone = s.nextInt();  
  
    Contato c = new Contato();  
    c.setNome(nome);  
    c.setTelefone(telefone);  
  
    return c;  
}
```

- Agora criaremos o método exibir todos os contatos;
- Utilizando do método **forEach()** do HashMap;

Resolução

- Como o próprio nome já diz, para cada chave do HashMap é possível acessar o seu valor;
- Esse método utiliza uma função anônima. Isto é, a função é implementada na hora que é passada como parâmetro;
- O protótipo do parâmetro que deve ser passado é: $(k,v) \rightarrow \{ \}$ sendo a função implementada dentro das chaves;
 - **k** (key): é chave de cada item;
 - **v** (value): é o objeto associado a essa chave;
- O método irá executar a função anônima a cada iteração, ou seja, para cada chave do HashMap;
- Como queremos que para cada chave, em nosso caso, para cada letra do alfabeto, seja exibido no console a letra em questão e a lista de contatos associada a ela, iremos percorrer a LinkedList associada a chave:

```
public static void exibirContatos() {  
    lista.buscarContatos().forEach((chave, lista) -> {  
        System.out.println(chave + ":");  
        for (Contato contato : lista) {  
            System.out.println("  " + contato.toString());  
        }  
    });  
}
```

- Note que a chamada do método de buscar contatos do objeto lista (**lista.buscarContatos()**) retorna um `HashMap`;
- Por isso que é possível utilizar o **forEach()**;

Resolução

- Dentro do **forEach()** ainda é realizado um for, que percorre todos os objetos da lista e os exibe no console;
- Agora iremos implementar o método que exibe apenas os contatos relacionados a uma determina letra passada como parâmetro;
- Esse método na verdade é exatamente igual ao que ocorre dentro do **forEach()** do método anterior;
- A única diferença é que o for utiliza um index **i** para percorrer e associar um número ao objeto em questão:

```
public static void exibirContatos(char letra) {  
    for (int i = 0; i < lista.buscarContatos(letra).size(); i++) {  
        System.out.println("Codigo: " + i);  
        System.out.println(lista.buscarContatos(letra).get(i).toString());  
    }  
}
```

Resolução

- Iremos utilizar o método apresentado anteriormente, no método de remover contatos;
- Esse método é um pouco mais extenso, por isso é necessário detalhá-lo:
- Primeiro requisitamos ao usuário a letra do contato a ser removido:
- Iremos torná-la maiúscula para que se adeque ao nosso sistema de chaves, onde todos as chaves do HashMap são letras maiúsculas;

```
public static void removerContato() {  
    System.out.println("Escolha uma letra que deseja remover:");  
    String entrada = s.next().toUpperCase();
```

- Agora iremos verificar se para esta letra existem contatos associados;
- Isto é, se o tamanho da lista retornada pelo método de buscar contatos for maior do que zero, isso significa que há contatos nessa lista;

- Utilizaremos do método **size()** do LinkedList:

```
if ( lista . buscarContatos ( entrada . charAt ( 0 ) ) . size ( ) > 0 ) {
```

- Caso positivo, exibimos a lista de contatos e requisitamos ao usuário qual contato ele deseja remover:

```
    if ( lista . buscarContatos ( entrada . charAt ( 0 ) ) . size ( ) > 0 ) {  
        exibirContatos ( entrada . charAt ( 0 ) );  
  
        System . out . println ( "Escolha um contato para remover:" );  
        int index = s . nextInt ( );
```

- Caso o index escolhido pelo usuário seja um número menor que o tamanho de itens da lista, então é possível remover;
- Para isso passaremos o contato que está na posição "index" do LinkedList associado a chave (letra) que o usuário escolheu:

```
        if (index < lista.buscarContatos(entrada.charAt(0)).size()) {  
            lista.removeContato(lista.buscarContatos(entrada.charAt(0)).get(  
index));  
        }
```

- Toda o primeiro if ficou assim:

```
        if (lista.buscarContatos(entrada.charAt(0)).size() > 0) {  
            exhibirContatos(entrada.charAt(0));  
  
            System.out.println("Escolha um contato para remover:");  
            int index = s.nextInt();  
  
            if (index < lista.buscarContatos(entrada.charAt(0)).size()) {  
                lista.removeContato(lista.buscarContatos(entrada.charAt(0)).get(  
index));  
            }
```

```
} else {
```

- Resta apenas tratar o caso contrário, que é quando a lista está vazia:


```
} else {  
    System.out.println("N o existem contatos para serem removidos");  
}
```

- E por fim o método main:
- Ele possui um laço de repetição que só é quebrado quando o usuário digitar 0;


```
public static void main(String[] args) {  
  
    int opcao = -1;  
  
    while (opcao != 0) {  
  
        System.out.println("Escolha uma opção:");  
        System.out.println("0 - Sair");  
        System.out.println("1 - Cadastrar um contato");  
        System.out.println("2 - Remover um contato");  
        System.out.println("3 - Mostrar todos os contatos");  
  
        opcao = s.nextInt();  
    }  
}
```

- Após isso, um *switch case* determina qual método será chamado:

```
switch (opcao) {  
    case 0:  
        break;  
    case 1:  
        Contato contato = novoContato();  
        lista.adicionaContato(contato);  
        break;  
    case 2:  
        removerContato();  
        break;  
    case 3:  
        exibirContatos();  
        break;  
    default:  
        break;  
}
```

 KUWAKI, V. T. F. Modelo de slides udesc lattex. In: . [S.l.]: Disponível em: <<https://github.com/takeofriedrich/slidesUdescLattex>>. Acesso em: 24 jan. 2020.

Duvidas:
Vinicius Takeo Friedrich Kuwaki
vinicius.kuwaki@edu.udesc.br
github.com/takeofriedrich