UFOP (Universidade Federal de Ouro Preto)

PROFESSOR:

Guillermo Cámara Chávez

RELATÓRIO - BCC 221:

Grupo:

Gabriel Catizani Faria Oliveira (20.1.4004)

Lucas Gomes dos Santos (20.1.4108)

Rafael Augusto Freitas Oliveira (20.1.4038)

Robson Novato Lobão (20.1.4018)

Trabalho Prático 2 - Imobiliária (em Java)

Ouro Preto, Minas Gerais

30 de dezembro de 2021

1) Introdução

1.1) Objetivo

Implementar uma corretora de imóveis que tem um sistema que gerencia imóveis de 3 tipos: casa, apartamento ou chácara.

1.2) Implementação das classes

Faremos uma classe base Imóveis que possui os atributos id, valor do imovel, o proprietário, a rua, bairro e cidade onde se encontram, o numero do imovel, quantidade de quartos e quantidade de banheiro e assim irei implementar getters e setter para cada um deles.

Além disso, iremos fazer as classes derivadas que vão herdar da classe base imóveis que são as classes Casa, Apartamento ou Chácara:

- Casa: terão os atributos específicos que são quantos andares a casa tem e se tem ou não sala de jantar.
- Apartamento: terão os atributos específicos que é qual andar do prédio, a taxa de condomínio, se tem ou não elevador e se tem ou não sacada.
- Chácara: terão os atributos específicos que é se tem ou não salão de festa, se tem ou não salão de jogos, se tem ou não campo de futebol, se tem ou não churrasqueira e se tem ou não tem piscina.

E por fim faço os respectivos getters e setters de cada classe e o sobrecarga do operador toString() para imprimir os atributos (dados).

1.3) Funções

Fizemos as funções que realizam os métodos pedidos em uma classe separada chamada Func, que em sua maioria retornam Vectores da classe Imovel com o que queremos. Além disso, temos funções que apenas retornam true ou false e funções que retornam um iterator

1.5) Telas

Nas telas é onde é implementada a nossa interface gráfica e suas funcionalidades e onde é rodado o main, ou seja, onde lançamos a execução do programa

2) Desenvolvimento

2.1) Coleção implementadas

A única coleção usada nesse TP foi a <*Vector*>:

Inicialmente, usamos o <*Vector*> por meio da **importação da biblioteca "java.util.Vector"** para fazer um vetor de referência a classe base Imóveis, que será usado para guardar cada um dos objetos em cada uma das posições desse vetor. Dessa maneira, vamos lendo os arquivos linha por linha usando um vetor de strings, tirando os ";" e guardando os objetos com seus respectivos construtores, onde para colocar os construtores criamos primeiro 3 ponteiros para Imovel que receberá ou os dados para o construtor da classe Casa, para o construtor da classe Apartamento ou para o construtor da classe Chácara.

OBS: Vamos inserir esses objetos no final do vetor de ponteiros a Imovel a medida que vamos lendo linhas por linhas usando a função da <vector> **add()**.

Usamos Vector nessa parte e praticamente em todo código, porque ele é facilmente implementado e facilmente manipulado. Com essa facilidade de manipulação, podemos (no vetor de referência a classe Imovel):

- Inserir objetos da classe Imovel no final do vetor com muita facilidade apenas utilizando o método "add()";
- Ver o tamanho do vetor com size() e também para fazer comparações;
- Usar o método isEmpty() para testarmos se o vetor é vazio para imprimirmos mensagens de vetor vazio;
- Percorrê-lo para imprimir os dados cada um dos objetos, ou usando a sobrecarga do operador toString(), ou fazendo a impressão de dados próprios de cada classe que herda da classe base por meio de getters;
- Também usamos a coleção Vector para retornar outros vetores das funções que foram criadas de forma fácil;
- Usamos o vetor pela facilidade de fazer o método de ordenação para ordenar os objetos de acordo com o dado que queremos ordenar, como nesse caso de acordo com o valor do imóvel;

- ☐ **OBSERVAÇÃO:** Para realizar o uso do **método sort()** para ordenar os dados de uma coleção a partir do valor, **implementamos o método compareTo() da interface Comparable** na classe Imovel

- Por fim, também pela facilidade de locomoção dos iteradores dentro dos vetores para procurar elementos e fazer comparações com o **hasNext() e next()** que são os métodos da classe Iterator

Além disso, a memória é alocada automaticamente e quando temos necessidade de mais espaço também é alocado mais memória. Assim, isso facilita a manipulação.

IMPORTANTE: Quando precisávamos pegar métodos específicos de cada classe que herda da classe Imovel, como getters e setters próprios, usamos downcasting. Isso acontece, pois o objeto que já foi uma derivada, agora está sendo apontado por uma base, ou seja, apenas conseguimos usar os métodos da classe base Imovel. Logo, para usarmos os métodos próprios das classes que herdam de Imovel, temos que fazer o downcasting e, assim, conseguimos usar getters que irão pegar dados próprios dessas outras classes.

2.2) Quais as melhores coleções?

• Acessar uma posição específica de uma coleção;

<Vector> : É propício, pois o vetor é facilmente acessado por meio do método get e add, onde junto deles coloco a posição que desejo acessar **get(int index)** e adicionar **add(int index, Object element)** um elemento, respectivamente.

• Adicionar um elemento e manter somente elementos únicos na coleção;

<HashSet>: É usado para armazenar o elemento na coleção por meio de uma tabela hash e não permite repetição de elementos

• Inserção no final;

<Stack> porque usa vetor como padrão em sua implementação

• Remoção no final;

<Stack> porque usa vetor como padrão em sua implementação

• Retornar um valor baseado em uma chave específica (não necessariamente inteiros);

<TreeMap> Pois manipula os **valores associados com chaves** e ainda, para acelerar o processo de pesquisa tanto para remoção e inserção, armazenamos os elementos em árvores

• Inserção no início;

<ArrayList>: Inserção eficiente na frente, já que usa um vetor em sua implementação e por conta disso é possível usar iteradores para facilitar a inserção

• Remoção no início;

<PriorityQueue>: Remoção eficiente na frente, já que usa o conceito de FIFO (first-in first-out).

• Busca por um elemento;

<TreeMap>, já que usa as chaves para pesquisa e não o registro em si. Além disso, com o elementos em um árvore de pesquisa o processo fica ainda mais rápido, já que uma árvore binária tem uma ordem de complexidade O(log n), ou seja, a pesquisa é muito rápida

• Coleção com o comportamento de primeiro a entrar é o último a sair

<Stack>: Implementa o conceito de FILO (First-in, Last-Out)

• Coleção com o comportamento de primeiro a entrar é o primeiro a sair.

<PriorityQueue: A queue propriamente dita, é uma estrutura de dado do tipo FIFO (first-in first-out).

2.3) Interface Gráfica

Para construção da interface gráfica, utilizamos o framework swing que possui uma coleção de elementos gráficos para ser utilizado no java.

O swing renderiza por conta própria todos os seus componentes, ao contrário da maioria das outras APIs, que atribui essa tarefa ao sistema operacional.