DESAFIO AULA 4

Java Platform - Parte 2

2019

Prof. Danilo Vitoriano
profdanilo.vitoriano@fiap.com.br

MBA FULL STACK DEVELOPER
MICROSERVICES, CLOUD & IoT

1. Ordenação

Vamos ordenar o campo de destino da tela de detalhes da conta para que as contas apareçam em ordem alfabética de titular.

1. Faça sua classe Conta implementar a interface Comparable<Conta>. Utilize o critério de ordenar pelo titular da conta.

```
public class Conta implements Comparable<Conta> {
    ...
}
```

Deixe o seu método compareTo parecido com este:

```
public class Conta implements Comparable<Conta> {

    // ... todo o código anterior fica aqui

    public int compareTo(Conta outraConta) {
              return this.titular.compareTo(outraConta.titular);
    }
}
```

2. Queremos que as contas apareçam ordenadas pelo titular. Use o Collections.sort() para ordenar a lista adicionando as contas criadas:

```
List<ContaCorrente> contas = new ArrayList<>();
contas.add(c1);
contas.add(c2);
contas.add(c3);

Collections.sort(contas);
```

3. (opcional) Crie uma nova classe TestaLista que cria uma ArrayList e insere novas contas com saldos aleatórios usando um laço (for). Adivinhe o nome da classe para colocar saldos

aleatórios? Random. Do pacote java.util. Consulte sua documentação para usá-la (utilize o método nextInt() passando o número máximo a ser sorteado).

4. Modifique a classe TestaLista para utilizar uma LinkedList em vez de ArrayList:

```
    List<Conta> contas = new LinkedList<Conta>();
```

Precisamos alterar mais algum código para que essa substituição funcione? Rode o programa. Alguma diferença?

5. (opcional) Imprima a referência para essa lista. O toString de uma ArrayList/LinkedList é reescrito?

```
    System.out.println(contas);
```

2. Collections

1. Crie um código que insira 30 mil números numa ArrayList e pesquise-os. Vamos usar um método de System para cronometrar o tempo gasto:

```
public class TestaPerformance {

    public static void main(String[] args) {
        System.out.println("Iniciando...");
        Collection<Integer> teste = new ArrayList<>();
        long inicio = System.currentTimeMillis();

        int total = 30000;

        for (int i = 0; i < total; i++) {
            teste.add(i);
        }

        for (int i = 0; i < total; i++) {
            teste.contains(i);
        }
```

```
        long fim = System.currentTimeMillis();
        long tempo = fim - inicio;
        System.out.println("Tempo gasto: " + tempo);
    }
}
```

Troque a ArrayList por um HashSet e verifique o tempo que vai demorar:

```
    Collection<Integer> teste = new HashSet<>();
```

O que é lento? A inserção de 30 mil elementos ou as 30 mil buscas? Descubra computando o tempo gasto em cada for separadamente.

A diferença é mais que gritante. Se você passar de 30 mil para um número maior, como 50 ou 100 mil, verá que isso inviabiliza por completo o uso de uma List, no caso em que queremos utilizá-la essencialmente em pesquisas.

2. (conceitual, importante) Repare que, se você declarar a coleção e der new assim:

```
    Collection<Integer> teste = new ArrayList<>();
```

em vez de:

```
    ArrayList<Integer> teste = new ArrayList<>();
```

É garantido que vai ter de alterar só essa linha para substituir a implementação por HashSet. Estamos aqui usando o polimorfismo para nos proteger que mudanças de implementação venham nos obrigar a alterar muito código. Mais uma vez: programe voltado a interface, e não à implementação!

Esse é um excelente exemplo de bom uso de interfaces, afinal, de que importa como a coleção funciona? O que queremos é uma coleção qualquer, isso é suficiente para os nossos propósitos! Nosso código está com baixo acoplamento em relação a estrutura de dados utilizada: podemos trocá-la facilmente.

Esse é um código extremamente elegante e flexível. Com o tempo você vai reparar que as pessoas tentam programar sempre se referindo a essas interfaces menos específicas, na medida do possível: métodos costumam receber e devolver Collections, Lists e Sets em vez

de referenciar diretamente uma implementação. É o mesmo que ocorre com o uso de InputStream e OutputStream: eles são o suficiente, não há porque forçar o uso de algo mais específico.

Obviamente, algumas vezes não conseguimos trabalhar dessa forma e precisamos usar uma interface mais específica ou mesmo nos referir ao objeto pela sua implementação para poder chamar alguns métodos. Por exemplo, TreeSet tem mais métodos que os definidos em Set, assim como LinkedList em relação à List.

Dê um exemplo de um caso em que não poderíamos nos referir a uma coleção de elementos como Collection, mas necessariamente por interfaces mais específicas como List ou Set.

3. Faça testes com o Map, como visto nesse capítulo:

```
public class TestaMapa {

    public static void main(String[] args) {
        Conta c1 = new ContaCorrente();
        c1.deposita(10000);

        Conta c2 = new ContaCorrente();
        c2.deposita(3000);

        // cria o mapa
        Map mapaDeContas = new HashMap();

        // adiciona duas chaves e seus valores
        mapaDeContas.put("diretor", c1);
        mapaDeContas.put("gerente", c2);

        // qual a conta do diretor?
        Conta contaDoDiretor = (Conta)
mapaDeContas.get("diretor");
        System.out.println(contaDoDiretor.getSaldo());
    }
}
```

Depois, altere o código para usar o generics e não haver a necessidade do casting, além da garantia de que nosso mapa estará seguro em relação a tipagem usada.

Você pode utilizar o quickfix do Eclipse para que ele conserte isso para você: na linha em que você está chamando o put, use o ctrl + 1. Depois de mais um quickfix (descubra!) seu código deve ficar como segue:

```
// cria o mapa
Map<String, Conta> mapaDeContas = new HashMap<>();
```

Que opção do ctrl + 1 você escolheu para que ele adicionasse o generics para você?

4. (opcional) Assim como no exercício 1, crie uma comparação entre ArrayList e LinkedList, para ver qual é a mais rápida para se adicionar elementos na primeira posição (list.add(0, elemento)), como por exemplo:

```
public class TestaPerformanceDeAdicionarNaPrimeiraPosicao {
    public static void main(String[] args) {
        System.out.println("Iniciando...");
        long inicio = System.currentTimeMillis();

        // trocar depois por ArrayList
        List<Integer> teste = new LinkedList<>();

        for (int i = 0; i < 30000; i++) {
            teste.add(0, i);
        }

        long fim = System.currentTimeMillis();
        double tempo = (fim - inicio) / 1000.0;
        System.out.println("Tempo gasto: " + tempo);
    }
}
```

Seguindo o mesmo raciocínio, você pode ver qual é a mais rápida para se percorrer usando o get(indice) (sabemos que o correto seria utilizar o enhanced for ou o Iterator). Para isso, insira 30 mil elementos e depois percorra-os usando cada implementação de List.

Perceba que aqui o nosso intuito não é que você aprenda qual é o mais rápido, o importante é perceber que podemos tirar proveito do polimorfismo para nos comprometer apenas com a interface. Depois, quando necessário, podemos trocar e escolher uma implementação mais adequada as nossas necessidades.

Qual das duas listas foi mais rápida para adicionar elementos à primeira posição?

5. (opcional) Crie a classe Banco (caso não tenha sido criada anteriormente) no pacote br.com.fiap.contas.modelo que possui uma List de Conta chamada contas. Repare que numa lista de Conta, você pode colocar tanto ContaCorrente quanto ContaPoupanca por causa do polimorfismo.

Crie um método void adiciona(Conta c), um método Conta pega(int x) e outro int pegaQuantidadeDeContas(). Basta usar a sua lista e delegar essas chamadas para os métodos e coleções que estudamos.

Como ficou a classe Banco?

6. (opcional) No Banco, crie um método Conta buscaPorTitular(String nome) que procura por uma Conta cujo titular seja equals ao nomeDoTitular dado.

Você pode implementar esse método com um for na sua lista de Conta, porém não tem uma performance eficiente.

Adicionando um atributo privado do tipo Map<String, Conta> terá um impacto significativo. Toda vez que o método adiciona(Conta c) for invocado, você deve invocar .put(c.getTitular(), c) no seu mapa. Dessa maneira, quando alguém invocar o método Conta buscaPorTitular(String nomeDoTitular), basta você fazer o get no seu mapa, passando nomeDoTitular como argumento!

Note, apenas, que isso é só um exercício! Dessa forma você não poderá ter dois clientes com o mesmo nome nesse banco, o que sabemos que não é legal.

Referências:

Apostila Caelum - Java Orientação a Objetos
Apostila J2SE - Wincomp