Aula 19: Collections Framework

Professor(a): Virgínia Fernandes Mota
http://www.dcc.ufmg.br/∼virginiaferm

TECNOLOGIAS DE PROGRAMAÇÃO - SETOR DE INFORMÁTICA

Collections Framework

- Ao término desta aula, você será capaz de:
  - utilizar arrays, lists, sets ou maps dependendo da necessidade do programa;
  - iterar e ordenar listas e coleções;
  - usar mapas para inserção e busca de objetos.

Arrays são trabalhosos...

- Como vimos na aula de arrays, manipulá-las é bastante trabalhoso. Essa dificuldade aparece em diversos momentos:
  - não podemos redimensionar um array em Java;
  - é impossível buscar diretamente por um determinado elemento cujo índice não se sabe;
  - não conseguimos saber quantas posições do array já foram populadas sem criar, para isso, métodos auxiliares.

Arrays são trabalhosos...

- Supondo que os dados armazenados representem contas, o que acontece quando precisarmos inserir uma nova conta no banco? Precisaremos procurar por um espaço vazio? Guardaremos em alguma estrutura de dados externa, as posições vazias? E se não houver espaço vazio? Teríamos de criar um array maior e copiar os dados do antigo para ele?
- Como posso saber quantas posições estão sendo usadas no array? Vou precisar sempre percorrer o array inteiro para conseguir essa informação?
- **Solução**: um conjunto de classes e interfaces conhecido como **Collections Framework**.

Listas: java.util.List

- Um primeiro recurso que a API de Collections traz são **listas**.
- Uma lista é uma coleção que permite elementos duplicados e mantém uma ordenação específica entre os elementos.
- Você tem a garantia de que, quando percorrer a lista, os elementos serão encontrados em uma ordem pré-determinada, definida na hora da inserção dos mesmos.
- Ela resolve todos os problemas que levantamos em relação ao array!!

Listas: java.util.List

- A API de Collections traz a interface java.util.List, que especifica o que uma classe deve ser capaz de fazer para ser uma lista.
- Há diversas implementações disponíveis, cada uma com uma forma diferente de representar uma lista.
- Mais utilizada: ArrayList.

Listas: java.util.List

- Cuidado: ArrayLista não é um array!!
- O que ocorre é que, internamente, ela usa um array como estrutura para armazenar os dados, porém este atributo está propriamente encapsulado e você não tem como acessá-lo.
- Você não pode usar [] com uma ArrayList, nem acessar atributo length. Não há relação!

Listas: java.util.List

- Para criar um ArrayList, basta chamar o construtor:

```
1       ArrayList lista = new ArrayList();
```

- É sempre possível abstrair a lista a partir da interface List:

```
1       List lista = new ArrayList();
```

- Para criar uma lista de nomes (String), podemos fazer:

```
1       List lista = new ArrayList();
2       lista.add("Manoel");
3       lista.add("Joaquim");
4       lista.add("Maria");
```

Listas: java.util.List

- A interface List possui dois métodos add, um que recebe o objeto a ser inserido e o coloca no final da lista, e um segundo que permite adicionar o elemento em qualquer posição da mesma.
- E o tamanho da lista? Podemos acrescentar quantos elementos quisermos, que a lista cresce conforme for necessário.
- Toda lista (na verdade, toda Collection) trabalha do modo mais genérico possível: Todos os métodos trabalham com Object.

Listas: java.util.List

- Para saber quantos elementos há na lista, usamos o método size().
- Há ainda um método get(int) que recebe como argumento o índice do elemento que se quer recuperar.
- E para imprimir?

```
1     ContaCorrente c1 = new ContaCorrente();
2     c1.deposita(100);
3
4     ContaCorrente c2 = new ContaCorrente();
5     c2.deposita(200);
6
7     List contas = new ArrayList();
8     contas.add(c1);
9     contas.add(c3);
10
11    for (int i = 0; i < contas.size(); i++) {
12            ContaCorrente cc = (ContaCorrente) contas.
                  get(i);
13            System.out.println(cc.getSaldo());
14    }
```

Listas: java.util.List

- Há ainda outros métodos, como remove() que recebe um objeto que se deseja remover da lista; e contains(), que recebe um objeto como argumento e devolve true ou false, indicando se o elemento está ou não na lista.
- Uma lista é uma excelente alternativa a um array comum, já que temos todos os benefícios de arrays, sem a necessidade de tomar cuidado com remoções, falta de espaço etc.
- Outra implementação muito usada, a LinkedList, fornece métodos adicionais para obter e remover o primeiro e último elemento da lista.

Listas: java.util.List

- Geralmente, não nos interessa uma lista com vários tipos de objetos misturados...
- Podemos restringir as listas a um determinado tipo de objetos (e não qualquer Object):

```
1       List<ContaCorrente> contas = new ArrayList<
            ContaCorrente>();
2       //Ou ainda List<ContaCorrente> contas = new
            ArrayList<>();
3   contas.add(c1);
4   contas.add(c3);
5
6   for(int i = 0; i < contas.size(); i++) {
7     ContaCorrente cc = contas.get(i); // sem casting!
8     System.out.println(cc.getSaldo());
9   }
```

Listas: java.util.List

- Usando Lists em retornos de métodos:

```
1  class Agencia {
2
3    // modificação apenas no retorno:
4    public List<Conta> buscaTodasContas() {
5      ArrayList<Conta> contas = new ArrayList<>();
6
7      // para cada conta do banco de dados, contas.add
8
9      return contas;
10   }
11 }
```

- Usando Lists em parâmetros:

```
1  class Agencia {
2
3    public void atualizaContas(List<Conta> contas) {
4      // ...
5    }
6  }
```

Ordenação: Collections.sort

- Vimos que as listas são percorridas de maneira pré-determinada de acordo com a inclusão dos itens.
- Mas, muitas vezes, queremos percorrer a nossa lista de maneira ordenada.
- O método sort:

```
1  List<String> lista = new ArrayList<>();
2  lista.add("Pseudo");
3  lista.add("Pagagas");
4  lista.add("Canela");
5
6  // repare que o toString de ArrayList foi sobrescrito:
7  System.out.println(lista);
8
9  Collections.sort(lista);
10
11 System.out.println(lista);
```

Ordenação: Collections.sort

- E se quisermos ordenar uma lista de ContaCorrente? Em que ordem a classe Collections ordenará? Pelo saldo? Pelo nome do correntista?
- Precisamos pensar em um critério de ordenação, uma forma de determinar qual elemento vem antes de qual.
- É necessário instruir o sort sobre como comparar nossas ContaCorrente a fim de determinar uma ordem na lista.

Ordenação: Collections.sort

- Para isto, o método sort necessita que todos seus objetos da lista sejam comparáveis e possuam um método que se compara com outra ContaCorrente.
- Como é que o método sort terá a garantia de que a sua classe possui esse método?
- Através de uma interface!

Ordenação: Collections.sort

- Vamos fazer com que os elementos da nossa coleção implementem a interface java.lang.Comparable, que define o método int compareTo(Object).
- Este método deve retornar zero, se o objeto comparado for igual a este objeto, um número negativo, se este objeto for menor que o objeto dado, e um número positivo, se este objeto for maior que o objeto dado.

Ordenação: Collections.sort

```
1  public class ContaCorrente extends Conta
2              implements Comparable<ContaCorrente> {
3
4    // ... todo o código anterior fica aqui
5
6    public int compareTo(ContaCorrente outra) {
7      if (this.saldo < outra.saldo) {
8        return -1;
9      }
10
11     if (this.saldo > outra.saldo) {
12       return 1;
13     }
14
15     return 0;
16   }
17 }
```

- Nossa classe tornou-se "comparável"!
- Logo o critério de ordenação é totalmente aberto, definido pelo programador.

Outros métodos da classe Collections

- A classe Collections traz uma grande quantidade de métodos estáticos úteis na manipulação de coleções.
  - binarySearch(List, Object): Realiza uma busca binária por determinado elemento na lista ordenada e retorna sua posição ou um número negativo, caso não encontrado.
  - max(Collection): Retorna o maior elemento da coleção.
  - min(Collection): Retorna o menor elemento da coleção.
  - reverse(List): Inverte a lista.
  - e por aí vai...
- Existe uma classe análoga, a java.util.Arrays, que faz operações similares com arrays.

Exercícios: Ordenação

- 1. Abra sua classe Conta e veja se ela possui o atributo numero. Se não possuir, adicione-o:

```
1  protected int numero;
```

- 2. Faça sua classe ContaPoupanca implementar a interface Comparable<ContaPoupanca>.
- 3. Utilize o critério de ordenar pelo número da conta ou pelo seu saldo.
- 4. Crie uma classe TestaOrdenacao, onde você vai instanciar diversas contas e adicioná-las a uma List<ContaPoupanca>.

Exercícios: Ordenação

- **Observação**: repare que escrevemos um método compareTo em nossa classe e nosso código nunca o invoca!! Isto é muito comum. Reescrevemos (ou implementamos) um método e quem o invocará será um outro conjunto de classes (nesse caso, quem está chamando o compareTo é o Collections.sort, que o usa como base para o algoritmo de ordenação). Isso cria um sistema extremamente coeso e, ao mesmo tempo, com baixo acoplamento: a classe Collections nunca imaginou que ordenaria objetos do tipo ContaPoupanca, mas já que eles são Comparable, o seu método sort está satisfeito.

Exercícios: Ordenação

- 5. Como posso inverter a ordem de uma lista? Como posso embaralhar todos os elementos de uma lista? Como posso rotacionar os elementos de uma lista?
- 6. Mude o critério de comparação da sua ContaPoupanca. Adicione um atributo nomeDoCliente na sua classe (caso ainda não exista algo semelhante) e tente mudar o compareTo para que uma lista de ContaPoupanca seja ordenada alfabeticamente pelo atributo nomeDoCliente.
- 7. PESQUISE E RESPONDA: Estamos falando de coesão e acoplamento de classes todo o tempo. O que isso significa?

Conjunto: java.util.Set

- Um conjunto (Set) funciona de forma análoga aos conjuntos da matemática, ele é uma coleção que não permite elementos duplicados.
- Outra característica fundamental dele é o fato de que a ordem em que os elementos são armazenados pode não ser a ordem na qual eles foram inseridos no conjunto.
- A interface não define como deve ser este comportamento.
- Tal ordem varia de implementação para implementação.

Conjunto: java.util.Set

- Um conjunto é representado pela interface Set e tem como suas principais implementações as classes HashSet, LinkedHashSet e TreeSet.

```
1  Set<String> cargos = new HashSet<>();
2
3  cargos.add("Gerente");
4  cargos.add("Diretor");
5  cargos.add("Presidente");
6  cargos.add("Secretária");
7  cargos.add("Funcionário");
8  cargos.add("Diretor"); // repetido!
9
10 // imprime na tela todos os elementos
11 System.out.println(cargos);
12 //Aqui, o segundo Diretor não será adicionado e o
       método add lhe retornará false.
```

Conjunto: java.util.Set

- O uso de um Set pode parecer desvantajoso, já que ele não armazena a ordem, e não aceita elementos repetidos.
- Não há métodos que trabalham com índices, como o get(int) que as listas possuem.
- Vantagem: Existem implementações, como a HashSet, que possui uma performance incomparável com as Lists quando usado para pesquisa.

Principais interfaces: java.util.Collection

`boolean add(Object)`	Adiciona um elemento na coleção. Como algumas coleções não suportam elementos duplicados, este método retorna `true` ou `false` indicando se a adição foi efetuada com sucesso.
`boolean remove(Object)`	Remove determinado elemento da coleção. Se ele não existia, retorna `false`.
`int size()`	Retorna a quantidade de elementos existentes na coleção.
`boolean contains(Object)`	Procura por determinado elemento na coleção, e retorna verdadeiro caso ele exista. Esta comparação é feita baseando-se no método `equals()` do objeto, e não através do operador ==.
`Iterator iterator()`	Retorna um objeto que possibilita percorrer os elementos daquela coleção.

Percorrendo coleções

- Como percorrer os elementos de uma coleção? Se for uma lista, podemos sempre utilizar um laço for, invocando o método get para cada elemento. Mas e se a coleção não permitir indexação?
- Podemos usar o enhanced-for (o "foreach"):

```
1  Set<String> conjunto = new HashSet<>();
2
3  conjunto.add("java");
4  conjunto.add("vraptor");
5  conjunto.add("scala");
6
7  for (String palavra : conjunto) {
8    System.out.println(palavra);
9  }
```

Mapas - java.util.Map

- Muitas vezes queremos buscar rapidamente um objeto dado alguma informação sobre ele.
- Exemplo: dada a placa do carro, obter todos os dados do carro.
- Usar listas pode ser péssimo para a performance!
- Solução: Um mapa.

Mapas - java.util.Map

- Um mapa é composto por um conjunto de associações entre um objeto chave a um objeto valor.
- É equivalente ao conceito de dicionário!
- java.util.Map é um mapa, pois é possível usá-lo para mapear uma chave a um valor, por exemplo: mapeie à chave "turma"o valor "303A", ou então mapeie à chave "disciplina"ao valor "TP". Semelhante a associações de palavras que podemos fazer em um dicionário.

Mapas - java.util.Map

- Exemplo: criamos duas contas correntes e as colocamos em um mapa associando-as aos seus donos.

```
1  ContaCorrente c1 = new ContaCorrente();
2  c1.deposita(10000);
3
4  ContaCorrente c2 = new ContaCorrente();
5  c2.deposita(3000);
6
7  // cria o mapa
8  Map<String, ContaCorrente> mapaDeContas = new HashMap
       <>();
9
10 // adiciona duas chaves e seus respectivos valores
11 mapaDeContas.put("diretor", c1);
12 mapaDeContas.put("gerente", c2);
13
14 // qual a conta do diretor? (sem casting!)
15 ContaCorrente contaDoDiretor = mapaDeContas.get("
       diretor");
16 System.out.println(contaDoDiretor.getSaldo());
```

Mapas - java.util.Map

- Um mapa é muito usado para "indexar"objetos de acordo com determinado critério, para podermos buscar esse objetos rapidamente.
- Um mapa costuma aparecer juntamente com outras coleções, para poder realizar essas buscas!
- Suas principais implementações são o HashMap, o TreeMap e o Hashtable.

Boas práticas

- As coleções do Java oferecem grande flexibilidade ao usuário.
- A perda de performance em relação à utilização de arrays é "irrelevante", mas deve-se tomar algumas precauções:
  - Grande parte das coleções usam, internamente, um array para armazenar os seus dados. Quando esse array não é mais suficiente, é criada um maior e o conteúdo da antiga é copiado. Este processo pode acontecer muitas vezes, no caso de você ter uma coleção que cresce muito. Você deve, então, criar uma coleção já com uma capacidade grande, para evitar o excesso de redimensionamento.

Boas práticas

- Evite usar coleções que guardam os elementos pela sua ordem de comparação quando não há necessidade. Um TreeSet gasta computacionalmente O(log(n)) para inserir (ele utiliza uma árvore rubro-negra como implementação), enquanto o HashSet gasta apenas O(1).
- Não itere sobre uma List utilizando um for de 0 até list.size() e usando get(int) para receber os objetos. Enquanto isso parece atraente, algumas implementações da List não são de acesso aleatório como a LinkedList, fazendo esse código ter uma péssima performance computacional. (use Iterator)

Exercícios: 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:

```
1  long inicio = System.currentTimeMillis();
2
3  long fim = System.currentTimeMillis();
```

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

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

Exercícios: Collections

- 3. Faça testes com o Map. Crie um método no Banco *Conta buscaPorNome(String nome)* que procura por uma Conta cujo nome seja equals ao nome dado.
- 4. E se precisarmos ordernar uma lista com outro critério de comparação? Se precisarmos alterar a própria classe e mudar seu método compareTo, teremos apenas uma forma de comparação por vez. Precisamos de mais! Como definir outros critérios de ordenação?
- 5. Desafio: Definir uma Família usando a estrutura de dados Árvore. Imprima a árvore genealógica. (Defina uma classe Família que tem vários membros da classe Pessoa).

Na próxima aula...

Prova