Trabalho
de
Liguagem de Programação 2

Collection Frameworks Java

Aluno : Marcos Felipe Souza Pinto
Matricula: 20122104419

Universidade Veiga de Almeida

Collection Frameworks Java

COLLECTION FRAMEWORKS JAVA

O Java Collections Framework (JCF) é um conjunto de classes e interfaces que implementam comumente estruturas de dados de coleta reutilizáveis.

Embora referido como um quadro, ele funciona de uma forma de uma biblioteca. O JCF fornece ambas as interfaces que definem várias coleções e classes que os implementam.

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.

Em outras palavras, 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 (busca, remoção, tamanho "infinito",...)

LIST - JAVA.UTIL.LIST

A implementação mais utilizada da interface `List` é a `ArrayList`, que trabalha com um array interno para gerar uma lista. Portanto, ela é mais rápida na pesquisa do que sua concorrente, a `LinkedList`, que é mais rápida na inserção e remoção de itens nas pontas.

É comum confundirem uma `ArrayList` com um array, porém ela 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. Repare, também, que você não pode usar `[]` com uma `ArrayList`, nem acessar atributo `length`. Não há relação!

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.

Não há uma `ArrayList` específica para `Strings`, outra para Números, outra para Datas etc. Todos os métodos trabalham com `Object`.

Criando um ArrayList e atribuindo elementos nela :

```
List lista = new ArrayList();
lista.add("Manoel");
lista.add("Joaquim");
lista.add("Maria");
```

Para saber o tamanho é necessario um .size() e ainda tem um método `get(int)` que recebe como argumento o índice do elemento que se quer recuperar.

```
for (int i = 0; i < contas.size(); i++) {
  lista.get(i); // código não muito útil....
}
```

Em qualquer lista, é possível colocar qualquer **Object**. Com isso, é possível misturar objetos:

```
ContaCorrente cc = new ContaCorrente();

List lista = new ArrayList();
lista.add("Uma string");
lista.add(cc);
```

Como o método **`get`** devolve um **`Object`**, precisamos fazer o cast. Mas com uma lista com vários objetos de tipos diferentes, isso pode não ser tão simples

No Java 5.0, podemos usar o recurso de Generics para restringir as listas a um determinado tipo de objetos (e não qualquer **`Object`**). O uso de Generics também elimina a necessidade de casting, já que, seguramente, todos os objetos inseridos na lista serão do tipo **`ContaCorrente`**:

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

A partir do Java 7, se você instancia um tipo genérico na mesma linha de sua declaração, não é necessário passar os tipos novamente, basta usar **`new ArrayList<>().`** É conhecido como *operador diamante*:

```
List<ContaCorrente> contas = new ArrayList<>();
```

A classe **`Collections`** traz um método estático `sort` que recebe um **`List`** como argumento e o ordena por ordem crescente. Por exemplo:

```
List<String> lista = new ArrayList<>();
lista.add("Sérgio");
lista.add("Paulo");
lista.add("Guilherme");

// repare que o toString de ArrayList foi sobrescrito:
System.out.println(lista);

Collections.sort(lista);

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

Sempre que falamos em ordenação, 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. 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? Isso será feito, novamente, através de um contrato, de uma interface!

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.

Para ordenar as `ContaCorrente`s por saldo, basta implementar o `Comparable`:

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

  // ... todo o código anterior fica aqui

  public int compareTo(ContaCorrente outra) {
    if (this.saldo < outra.saldo) {
      return -1;
    }

    if (this.saldo > outra.saldo) {
      return 1;
    }

    return 0;
  }
}
```

Com o código anterior, nossa classe tornou-se "**comparável**": dados dois objetos da classe, conseguimos dizer se um objeto é maior, menor ou igual ao outro, segundo algum critério por nós definido. No nosso caso, a comparação será feita baseando-se no saldo da conta.

Repare que o critério de ordenação é totalmente aberto, definido pelo programador. Se quisermos ordenar por outro atributo (ou até por uma combinação de atributos), basta modificar a implementação do método `compareTo` na classe.

Quando chamarmos o método `sort` de `Collections`, ele saberá como fazer a ordenação da lista; ele usará o critério que definimos no método `compareTo`.

Mas, e o exemplo anterior, com uma lista de Strings? Por que a ordenação funcionou, naquele caso, sem precisarmos fazer nada? Simples: quem escreveu a classe `String`(lembre que ela é uma classe como qualquer outra) implementou a interface `Comparable` e o método `compareTo` para `String`s, fazendo comparação em ordem alfabética. (Consulte a documentação da classe `String` e veja o método `compareTo` lá). O mesmo acontece com outras classes como `Integer`, `BigDecimal`, `Date`, entre outras.

No Java 8 muitas dessas funcionalidades da `Collections` podem ser feitas através dos chamados `Streams`, que fica um pouco fora do escopo de um curso inicial de Java.
Existe uma classe análoga, a `java.util.Arrays`, que faz operações similares com arrays.

É importante conhecê-las para evitar escrever código já existente.

SET - 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.

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

O código a seguir cria um conjunto e adiciona diversos elementos, e alguns repetidos:

```
Set<String> cargos = new HashSet<>();

cargos.add("Gerente");
cargos.add("Diretor");
cargos.add("Presidente");
cargos.add("Secretária");
cargos.add("Funcionário");
cargos.add("Diretor"); // repetido!

// imprime na tela todos os elementos
System.out.println(cargos);
```

Aqui, o segundo **`Diretor`** não será adicionado e o método **`add`** lhe retornará **`false`**.

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. A grande vantagem do **`Set`** é que existem implementações, como a **`HashSet`**, que possui uma performance incomparável com as **`Lists`** quando usado para pesquisa (método **`contains`** por exemplo).

As coleções têm como base a interface `Collection`, que define métodos para adicionar e remover um elemento, e verificar se ele está na coleção, entre outras operações, como mostra a tabela a seguir:

`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.

Uma coleção pode implementar diretamente a interface **`Collection`**, porém normalmente se usa uma das duas subinterfaces mais famosas:
justamente **`Set`** e **`List`**.

A interface **`Set`**, como previamente vista, define um conjunto de elementos únicos enquanto a interface **`List`** permite elementos duplicados, além de manter a ordem a qual eles foram adicionados.

A busca em um **`Set`** pode ser mais rápida do que em um objeto do tipo **`List`**, pois diversas implementações utilizam-se de tabelas de espalhamento (*hash tables*), realizando a busca para tempo linear (**O(1)**).

INTERATOR - JAVA.UTIL.INTERATOR

Antes do Java 5 introduzir o novo enhanced-for, iterações em coleções eram feitas com o **`Iterator`**. Toda coleção fornece acesso a um *iterator*, um objeto que implementa a interface **`Iterator`**, que conhece internamente a coleção e dá acesso a todos os seus elementos, como a figura abaixo mostra.

Primeiro criamos um `Iterator` que entra na coleção. A cada chamada do método `next`, o`Iterator` retorna o próximo objeto do conjunto.
Um `iterator` pode ser obtido com o método `iterator()` de `Collection`, por exemplo numa lista de `String`:

```
Iterator<String> i = lista.iterator();
```

A interface `Iterator` possui dois métodos principais: `hasNext()` (com retorno booleano), indica se ainda existe um elemento a ser percorrido; `next()`, retorna o próximo objeto.

Voltando ao exemplo do conjunto de strings, vamos percorrer o conjunto:

```
Set<String> conjunto = new HashSet<>();
conjunto.add("item 1");
conjunto.add("item 2");
conjunto.add("item 3");

// retorna o iterator
Iterator<String> i = conjunto.iterator();
while (i.hasNext()) {
  // recebe a palavra
  String palavra = i.next();
  System.out.println(palavra);
}
```

O **`while`** anterior só termina quando todos os elementos do conjunto forem percorridos, isto é, quando o método **`hasNext`** mencionar que não existem mais itens.

MAPAS - JAVA.UTIL.MAP

Muitas vezes queremos buscar rapidamente um objeto dado alguma informação sobre ele. Um exemplo seria, dada a placa do carro, obter todos os dados do carro. Poderíamos utilizar uma lista para isso e percorrer todos os seus elementos, mas isso pode ser péssimo para a performance, mesmo para listas não muito grandes. Aqui entra o mapa.

Um mapa é composto por um conjunto de associações entre um objeto chave a um objeto valor. É equivalente ao conceito de dicionário, usado em várias linguagens. Algumas linguagens, como Perl ou PHP, possuem um suporte mais direto a mapas, onde são conhecidos como matrizes/arrays associativas.

`java.util.Map` é um mapa, pois é possível usá-lo para mapear uma chave a um valor, por exemplo: mapeie à chave "empresa" o valor "Marcos", ou então mapeie à chave "rua" ao valor "Vergueiro". Semelhante a associações de palavras que podemos fazer em um dicionário.

O método `put(Object, Object)` da interface `Map` recebe a chave e o valor de uma nova associação. Para saber o que está associado a um determinado objeto-chave, passa-se esse objeto no método `get(Object)`. Sem dúvida essas são as duas operações principais e mais frequentes realizadas sobre um mapa.

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

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(10000);

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

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

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

// qual a conta do diretor? (sem casting!)
ContaCorrente contaDoDiretor = mapaDeContas.get("diretor");
System.out.println(contaDoDiretor.getSaldo());
```

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!

Ele, assim como as coleções, trabalha diretamente com `Objects` (tanto na chave quanto no valor), o que tornaria necessário o casting no momento que recuperar elementos. Usando os generics, como fizemos aqui, não precisamos mais do casting.

Suas principais implementações são o `HashMap`, o `TreeMap` e o `Hashtable`.

Bibliografia :

Blog Caelum

https://en.wikipedia.org/wiki/Java_collections_framework

Dicionarios em Wiki

http://www.caelum.com.br/apostila-java-orientacao-objetos/collections-framework