Java Collections: Como utilizar Collections

Apresentar as interfaces, implementações e algoritmos da Collections Framework, assim como as estratégias para escolher a coleção mais adequada visando solucionar os requisitos de uma aplicação.

Por que eu devo ler este artigo:

Apresentar as interfaces, implementações e algoritmos da Collections Framework, assim como as estratégias para escc

Ver mais

Desde as primeiras versões, Java dispõe das estruturas de arrays e as classes `Vector` e `Hashtable`. No entanto, além da dificuldade em implementar estruturas de dados utilizando arrays, os desenvolvedores sentiam falta de classes que implementassem estruturas como listas ligadas e tabelas de espalhamento (hash). Para atender a essas necessidades, a partir de Java 1.2, foi criado um conjunto de interfaces e classes denominado Collections Framework, que faz parte do pacote `java.util`.

O que é Collections Framework?

Collections Framework é um conjunto bem definido de interfaces e classes para representar e tratar grupos de dados como uma única unidade, que pode ser chamada coleção, ou collection. A Collections Framework contém os seguintes elementos:

- **Interfaces**: tipos abstratos que representam as coleções. Permitem que coleções sejam manipuladas tendo como base o conceito "Programar para interfaces e não para implementações", desde que o acesso aos objetos se restrinja apenas ao uso de métodos definidos nas interfaces;
- **Implementações**: são as implementações concretas das interfaces;
- **Algoritmos**: são os métodos que realizam as operações sobre os objetos das coleções, tais como busca e ordenação.

A **Figura 1** mostra a árvore da hierarquia de interfaces e classes da Java Collections Framework que são derivadas da interface `Collection`. O diagrama usa a notação da UML, onde as linhas cheias representam `extends` e as linhas pontilhadas representam `implements`.

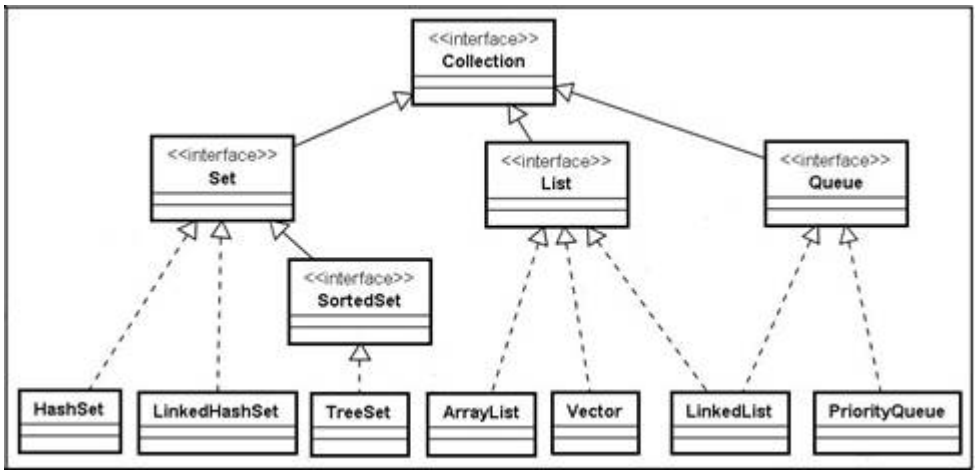

Figura 1. A hierarquia de interfaces e classes

mostra a **Figura 2.** Essas interfaces, mesmo não sendo consideradas coleções, podem ser manipuladas como tal.

Figura 2. Hierarquia de mapas

Interfaces

Neste momento vamos apresentar uma breve descrição de cada uma das interfaces da hierarquia:

- `Collection` – está no topo da hierarquia. Não existe implementação direta dessa interface, mas ela define as operações básicas para as coleções, como adicionar, remover, esvaziar, etc.;
- `Set` – interface que define uma coleção que não permite elementos duplicados. A interface `SortedSet`, que estende `Set`, possibilita a classificação natural dos elementos, tal como a ordem alfabética;
- `List` – define uma coleção ordenada, podendo conter elementos duplicados. Em geral, o usuário tem controle total sobre a posição onde cada elemento é inserido e pode recuperá-los através de seus índices. Prefira esta interface quando precisar de acesso aleatório, através do índice do elemento;
- `Queue` – um tipo de coleção para manter uma lista de prioridades, onde a ordem dos seus elementos, definida pela implementação de Comparable ou Comparator, determina essa prioridade. Com a interface fila pode-se criar filas e pilhas;

não. `SortedMap` é uma interface que estende `Map`, e permite classificação ascendente das chaves. Uma aplicação dessa interface é a classe Properties, que é usada para persistir propriedades/configurações de um sistema, por exemplo.

> **Nota**: Observe que usamos acima os termos `ordenação` e `classificação`. Dizemos que uma estrutura está ordenada se ela pode ser percorrida (iterada) em uma certa ordem, tal como os itens de um `ArrayList` podem ser percorridos através de seus índices. Por sua vez, a classificação diz respeito à ordenação na essência dos dados, tal como a classificação em ordem alfabética de `Strings` ou ordem numérica das classes `wrapper`, como `Integer` e `Double`, por exemplo. Podemos afirmar que uma estrutura classificada é uma estrutura ordenada, mas o inverso não é verdadeiro.

A API oferece também interfaces que permitem percorrer uma coleção derivada de Collection. Neste artigo falaremos de:

- `Iterator` – possibilita percorrer uma coleção e remover seus elementos;
- `ListIterator` – estende `Iterator` e suporta acesso bidirecional em uma lista, modificando e/ou removendo elementos.

Implementações

As interfaces apresentadas anteriormente possuem diversas implementações que são utilizadas para armazenar as coleções. Na **Tabela 1** estão resumidas as implementações mais comuns.

Tabela 1. Implementações de uso geral

Interfaces	Tabela de Espalhamento	Array Redimensionável	Árvore	Lista Ligada	Tabela de Espalhament Lista Ligad
Set	HashSet		TreeSet		LinkedHashS
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashM

A seguir apresentamos algumas características das implementações que podem ajudar a decidir qual delas utilizar em uma aplicação:

- `ArrayList` – é como um array cujo tamanho pode crescer. A busca de um elemento é rápida, mas inserções e exclusões não são. Podemos afirmar que as inserções e exclusões são lineares, o tempo cresce com o aumento do tamanho da estrutura. Esta implementação é preferível quando se deseja acesso mais rápido aos elementos. Por exemplo, se você quiser criar um catálogo com os livros de sua biblioteca pessoal e cada obra inserida receber um número sequencial (que será usado para acesso) a partir de zero;
- `LinkedList` – implementa uma lista ligada, ou seja, cada nó contém o dado e uma referência para o próximo nó. Ao contrário do `ArrayList`, a busca é linear e inserções e exclusões são rápidas. Portanto, prefira LinkedList quando a aplicação exigir grande quantidade de inserções e exclusões. Um pequeno sistema para gerenciar suas compras mensais de supermercado pode ser uma boa aplicação, dada a necessidade de constantes inclusões e exclusões de produtos;
- `HashSet` – o acesso aos dados é mais rápido que em um `TreeSet`, mas nada garante que os dados estarão ordenados. Escolha este conjunto quando a solução exigir elementos únicos e a ordem não for importante. Poderíamos usar esta implementação para criar um catálogo pessoal dos canções de

- `TreeSet` – os dados são classificados, mas o acesso é mais lento que em um `HashSet`. Se a necessidade for um conjunto com elementos não duplicados e acesso em ordem natural, prefira o TreeSet. É recomendado utilizar esta coleção para as mesmas aplicações de HashSet, com a vantagem dos objetos estarem em ordem natural;
- `LinkedHashSet` – é derivada de `HashSet`, mas mantém uma lista duplamente ligada através de seus itens. Seus elementos são iterados na ordem em que foram inseridos. Opcionalmente é possível criar um LinkedHashSet que seja percorrido na ordem em que os elementos foram acessados na última iteração. Poderíamos usar esta implementação para registrar a chegada dos corredores de uma maratona;
- `HashMap` – baseada em tabela de espalhamento, permite chaves e valores null. Não existe garantia que os dados ficarão ordenados. Escolha esta implementação se a ordenação não for importante e desejar uma estrutura onde seja necessário um ID (identificador). Um exemplo de aplicação é o catálogo da biblioteca pessoal, onde a chave poderia ser o ISBN (International Standard Book Number);
- `TreeMap` – implementa a interface `SortedMap`. Há garantia que o mapa estará classificado em ordem ascendente das chaves. Mas existe a opção de especificar uma ordem diferente. Use esta implementação para um mapa ordenado. Aplicação semelhante a HashMap, com a diferença que TreeMap perde no quesito desempenho;
- `LinkedHashMap` – mantém uma lista duplamente ligada através de seus itens. A ordem de iteração é a ordem em que as chaves são inseridas no mapa. Se for necessário um mapa onde os elementos são iterados na ordem em que foram inseridos, use esta implementação. O registro dos corredores de uma maratona, onde a chave seria o número que cada um recebe no ato da inscrição, é um exemplo de aplicação desta coleção.

Cada uma das implementações tem todos os métodos definidos em suas interfaces. Em qualquer uma delas é possível inserir elementos `null`. Em mapas, tanto chaves quanto valores podem ser `null`. Diferente de `Vector` e `Hashtable`, não são seguras para serem usadas com threads (não são Thread-safe). Ou seja, o acesso concorrente a esses objetos pode produzir resultados imprevisíveis. Além disso, são serializáveis – isto é, seus estados podem ser salvos – e suportam o método `clone()`, que cria uma cópia de um objeto.

> **Nota**: Thread-safe é o termo designado a objetos seguros para serem usados com threads.

Seguindo as boas práticas de orientação a objetos, você deve programar para interfaces e não para implementações. A recomendação é escolher uma implementação para instanciar o objeto e atribuir a nova coleção ao tipo de interface correspondente. Ou ainda, passar o objeto coleção para um método que espera um argumento do tipo interface. Seguindo essas práticas você conseguirá o que chamamos de baixo acoplamento, ou seja, poderá mudar facilmente de implementação sem que isso acarrete alteração no código da aplicação. Desta forma você fica livre para mudar a implementação sempre que questões relacionadas a desempenho ou detalhes de comportamento exigirem a mudança.

> **Nota**: Não confunda a interface `Collection` com a classe `Collections`. Essa classe oferece métodos estáticos utilitários que podem manipular coleções. Outra classe utilitária é `Arrays`, cujos métodos estáticos são aplicados a arrays.

Após uma visão geral da Collections Framework, vamos por as mãos na massa e

são analisadas as necessidades apresentadas pelo problema para decidir a interface a ser utilizada.

Como aplicar adequadamente a Collections Framework

Vimos então que temos sete interfaces: `Collection`, `List`, `Set`, `SortedSet`, `Map`, `SortedMap` e `Queue`. A pergunta que geralmente se faz é: Qual delas usar? Para selecionar adequadamente uma interface devemos analisar o problema e verificar como ele se enquadra nas características de cada interface. Somente após isso devemos decidir.

Iniciaremos com um problema simples. Queremos manter uma lista de nomes de alunos de uma escola que oferece cursos de Informática básica. Essa lista será percorrida na ordem em que os elementos são inseridos. Além disso, queremos poder acessar um nome de aluno aleatoriamente.

Analisando os requisitos do problema (lista na ordem de inserção e recuperação aleatória) e as características das interfaces disponíveis, optamos por utilizar `List`. Elementos não duplicados é um requisito que pode ser inferido na descrição do problema e isso poderia nos levar a escolher `Set`, mas vamos manter nossa decisão inicial por questões didáticas.

A interface List

`List` tem duas implementações – `ArrayList` e `LinkedList`. `ArrayList` oferece acesso aleatório rápido através do índice. Já em `LinkedList` o acesso aleatório é lento e necessita de um objeto nó para cada elemento, que é composto pelo dado propriamente dito e uma referência para o próximo nó, ou seja, consome mais

`LinkedList`. Para apoiar a decisão de usar uma ou outra implementação é melhor fazer testes de desempenho. Um teste simples é mostrado na **Listagem 1**. Execute o programa e anote o tempo. Substitua ArrayList por LinkedList e repita o teste. Ao final escolha a implementação mais eficiente.

Optamos então por usar `ArrayList`. Uma implementação básica pode ser vista na **Listagem 2**. Esta aplicação instancia um ArrayList e o atribui a uma referência do tipo List. Insere alguns nomes de alunos com o método `add()` e finalmente imprime a lista – as implementações de coleções sobrescrevem o método `toString()`, por isso podemos imprimir a lista passando apenas a referência para o método `println()`.

Listagem 1. Teste simples de desempenho

```
1  import java.util.*;
2  public class Teste {
3
4    public static void main(String[] args) {
5      final int MAX = 20000;
6      long tInicio = System.currentTimeMillis();
7      List<Integer> lista = new ArrayList<Integer>();
8      for (int i = 0; i < MAX; i++) {
9        lista.add(i);
10     }
11
12     for (int i = 0; i < MAX; i++) {
13       lista.contains(i);
14     }
15     long tFim = System.currentTimeMillis();
16     System.out.println("Tempo total: " + (tFim - tInicio));
17   }
18
19 }
```

Listagem 2. Classe ListaAluno utilizando uma implementação da interface List

```
1  import java.util.*;
2
3  public class ListaAluno {
4
5    public static void main(String[] args) {
6      List<String> lista = new ArrayList<String>();
7      lista.add("João da Silva");
8      lista.add("Antonio Sousa");
9      lista.add("Lúcia Ferreira");
10     System.out.println(lista);
11   }
12
13 }
```

Neste exemplo, a primeira consideração a fazer é que, tendo em mente a programação para interface, na declaração de `lista` foi usado o tipo `List`. Portanto, se decidirmos mudar a implementação para `LinkedList`, é necessário apenas substituir o tipo `ArrayList`.

A segunda consideração refere-se ao tipo de dado que uma lista pode adicionar. Normalmente é possível inserir qualquer Object em uma lista, ou seja, assim como poderíamos inserir uma String, poderíamos inserir `Aluno`, `Integer`, etc. Se a lista permite inserir Object, na hora de recuperar esses dados, é necessário fazer cast para o tipo desejado. Além disso, não se teria certeza do tipo de dado que foi inserido, e o cast poderia causar uma exceção. A partir de Java 5 foi introduzido o conceito de Generics, que nos permite escrever código reusável para qualquer tipo de objeto. Sob a ótica da utilização deste conceito em coleções, para definirmos o tipo que lista poderá adicionar, incluímos o parâmetro em sua declaração. Dessa forma o compilador gerará um erro caso se tente adicionar um

Adicionando novo requisito – Ordem ascendente

Vamos supor agora que desejamos que a lista seja classificada em ordem ascendente. Observando a documentação da implementação `ArrayList`, verificamos que não existe um método de ordenação. Para solucionar este requisito, uma opção seria mudar nossa aplicação para utilizar a interface `Set`, onde os elementos estariam classificados pela ordem natural, no entanto a inserção de novos elementos seria mais lenta. Sendo assim, vamos utilizar a classe utilitária `Collections`. Esta classe dispõe do método `sort()`, que pode classificar uma interface `List` em ordem natural ou classificar de acordo com a implementação da interface `Comparator`, que logo estudaremos ainda neste artigo. A aplicação deste método pode ser constatada na **Listagem 3**, onde podemos observar que o método `sort()` altera a lista original.

Listagem 3. Utilização do método sort() da classe Collections

```
1  import java.util.*;
2
3  public class ListaAluno {
4
5    public static void main(String[] args) {
6      List<String> lista = new ArrayList<String>();
7      lista.add("João da Silva");
8      lista.add("Antonio Sousa");
9      lista.add("Lúcia Ferreira");
10     System.out.println(lista);
11     Collections.sort(lista);
12     System.out.println(lista);
13   }
14
15 }
```

Adicionando novo requisito – Novos dados

Considere agora que as necessidades da nossa aplicação foram modificadas e que, precisamos, além do nome do aluno, o nome do curso que ele está fazendo e a sua nota. Definimos então a classe `Aluno` de acordo com a **Listagem 4**, e modificamos a classe `ListaAluno` conforme a **Listagem 5**, de maneira que a lista possa adicionar objetos `Aluno` ao invés de `String`.

Listagem 4. Classe Aluno

```
1  public class Aluno {
2    private String nome;
3    private String curso;
4    double nota;
5
6    Aluno(String nome, String curso, double nota) {
7      this.nome = nome;
8      this.curso = curso;
9      this.nota = nota;
10   }
11
12   public String toString() {
13     return this.nome;
14   }
15
16   // Métodos getters e setters
17 }
```

Listagem 5. Classe ListaAluno modificada

```
1  import java.util.*;
2
```

```
6       List<Aluno> lista = new ArrayList<Aluno>();
7
8       Aluno a = new Aluno("João da Silva", "Linux básico", 0);
9       Aluno b = new Aluno("Antonio Sousa", "OpenOffice", 0);
10      Aluno c = new Aluno("Lúcia Ferreira", "Internet", 0);
11      lista.add(a);
12      lista.add(b);
13      lista.add(c);
14      System.out.println(lista);
15    }
16  }
```

Classificação de objetos

Se incluirmos na **Listagem 5** a chamada ao método `sort()` veremos que o código não compila. O compilador retornará um erro informando que não encontrou o método `sort()`. Visto que apenas trocamos a classe `String` pela classe `Aluno`, parece razoável supor que o problema está na classe `Aluno`, e está correto.

A documentação da classe `Collections` nos informa que o método `sort()` aceita apenas listas cujos elementos sejam de tipos que implementem a interface `Comparable`, e Aluno não implementa `Comparable`.

Esta interface tem apenas um método a ser implementado, `compareTo()`. Sua implementação deve ser feita de forma a retornar um inteiro negativo, zero ou um inteiro positivo caso o objeto que execute o método seja menor, igual ou maior que o objeto passado como parâmetro. Cabe ao desenvolvedor decidir o critério que será adotado para comparar dois objetos.

Na classe `Aluno` consideraremos que a comparação entre dois objetos será determinada pela comparação entre seus nomes, que são do tipo String. Dessa forma a classe `Aluno` deve ser alterada para que fique de acordo com a **Listagem**

Listagem 6. A classe Aluno com implementação da interface Comparable

```
1  public class Aluno implements Comparable<Aluno>{
2       private String nome;
3       private String curso;
4       double nota;
5
6    Aluno(String nome, String curso, double nota) {
7      this.nome = nome;
8      this.curso = curso;
9      this.nota = nota;
10   }
11
12   public String toString() {
13     return this.nome;
14   }
15
16   public int compareTo(Aluno aluno) {
17     return this.nome.compareTo(aluno.getNome());
18   }
19
20   // Métodos getters e setters
21
22   public String getNome() {
23     return this.nome;
24   }
25 }
```

Note que no método `compareTo()` fizemos simplesmente uma chamada ao mesmo método, só que para o atributo `nome`, que é do tipo String. String é uma classe comparável, isto é, já implementa `Comparable`.

Agora podemos incluir uma chamada ao método `sort()` na classe `ListaAluno`. A ordenação implementada por `Comparable` é chamada ordenação natural. Por exemplo, em uma String a ordenação natural é a ordem alfabética, em uma classe

Em certas situações precisamos de uma ordenação diferente da natural ou temos uma coleção de objetos de uma classe de terceiros que não é comparável, ou seja, não implementa Comparable. Nesses casos usamos a interface `Comparator`. Para implementar esta ordenação é necessário escrever uma classe que implementa essa interface, definindo como os objetos da lista serão comparados. A interface possui apenas um método, `compare()`. Ele recebe dois objetos que são comparados e retorna um inteiro negativo, zero ou um inteiro positivo se o primeiro objeto é menor, igual ou maior que o segundo. Na **Listagem 7** temos um exemplo de implementação de `Comparator`.

Listagem 7. Implementação da interface Comparator

```
1  import java.util.Comparator;
2
3  public class ComparaAluno implements Comparator<Aluno> {
4    public int compare(Aluno a, Aluno b) {
5      return a.getNome().compareTo(b.getNome());
6    }
7  }
```

Para usar esta implementação chamamos o método sobrecarregado `sort()` da classe `Collections`. Ele recebe como argumentos a lista a ser ordenada e uma instância da implementação de `Comparator`, conforme a **Listagem 8**.

Listagem 8. Uso de Comparator para ordenar a lista

```
1  import java.util.*;
2
```

```
6       List<Aluno> lista = new ArrayList<Aluno>();
7       ComparaAluno ca = new ComparaAluno();
8
9       Aluno a = new Aluno("João da Silva", "Linux básico", 0);
10      Aluno b = new Aluno("Antonio Sousa", "OpenOffice", 0);
11      Aluno c = new Aluno("Lúcia Ferreira", "Internet", 0);
12      lista.add(a);
13      lista.add(b);
14      lista.add(c);
15      System.out.println(lista);
16      Collections.sort(lista, ca);
17      System.out.println(lista);
18    }
19  }
```

> **Nota**: A classe Arrays, cujos métodos estáticos se aplicam a arrays, também tem os métodos `sort()` que precisam das implementações de Comparable e Comparator, semelhante ao que foi estudado anteriormente para a classe **Collections**

A interface Iterator

As interfaces que estendem Collection herdam o método `iterator()`. Quando este método é chamado por um `collection` ele retorna uma interface `Iterator`. Após essa chamada, usamos os métodos de `Iterator` para percorrer um collection do início ao fim e até remover seus elementos. Na **Listagem 9** é mostrada uma aplicação desta interface em `ListaAluno`.

Listagem 9. Utilização da interface Iterator para percorrer uma lista

```
1  import java.util.*;
```

```
5     public static void main(String[] args) {
6       List<Aluno> lista = new ArrayList<Aluno>();
7
8       Aluno a = new Aluno("João da Silva", "Linux básico", 0);
9       Aluno b = new Aluno("Antonio Sousa", "OpenOffice", 0);
10      Aluno c = new Aluno("Lúcia Ferreira", "Internet", 0);
11      Aluno d = new Aluno("Antonio Sousa", "OpenOffice", 0);
12      lista.add(a);
13      lista.add(b);
14      lista.add(c);
15      lista.add(d);
16      System.out.println(lista);
17      Aluno aluno;
18      Iterator<Aluno> itr = lista.iterator();
19      while (itr.hasNext()) {
20        aluno = itr.next();
21        System.out.println(aluno.getNome());
22      }
23    }
24
25  }
```

Observe que é necessário informar o tipo que será retornado pelo `Iterator`. O método `hasNext()` retorna `true` se houver elemento a ser lido, e o método `next()` retorna o objeto, de acordo com o tipo informado na declaração da interface. A partir de Java 5 foi introduzido o enhanced-for, que facilita muito a iteração sobre collections e arrays. Mostraremos uma aplicação desse comando quando falarmos de Map.

> **Nota**: A interface `Iterator` pode ser usada também para percorrer um `Set`. O método `listIterator()` retorna uma interface `ListIterator`. Esta interface, além dos métodos `hasNext()` e `next()`, oferece o método `hasPrevious()`, que retorna `true` se existir um elemento anterior, e o método `previous()` que retorna o elemento anterior. Além desses,

A interface Set

Uma das características de `List` é que ela permite elementos duplicados, o que não é desejável em nossa lista de alunos. Analisando as interfaces, concluímos que `Set` é o que realmente precisamos, pois não permite elementos duplicados. Como `HashSet` tem desempenho superior a `TreeSet`, optamos por esta implementação.

Dessa forma, pode-se observar na **Listagem 10** a classe `ListaAluno` modificada para usar `Set`. Note que forçamos a inserção de um objeto duplicado, mas quando executamos a aplicação constatamos que o objeto foi inserido. Se um Set não permite elementos duplicados, onde está o erro? Como `HashSet` determina que dois objetos estão duplicados?

Listagem 10. ListaAluno usando a interface Set

```
1  import java.util.*;
2
3  public class ListaAluno {
4
5    public static void main(String[] args) {
6      Set<Aluno> conjunto = new HashSet<Aluno>();
7
8      Aluno a = new Aluno("João da Silva", "Linux básico", 0);
9      Aluno b = new Aluno("Antonio Sousa", "OpenOffice", 0);
10     Aluno c = new Aluno("Lúcia Ferreira", "Internet", 0);
11     Aluno d = new Aluno("Antonio Sousa", "OpenOffice", 0);
12     conjunto.add(a);
13     conjunto.add(b);
14     conjunto.add(c);
15     conjunto.add(d);
16     System.out.println(conjunto);
17   }
```

`HashSet` usa o código hash do objeto – dado pelo método `hashCode()` – para saber onde deve por e onde buscar o mesmo no conjunto (Set). Antes ele verifica se não existe outro objeto no Set com o mesmo código hash. Se não há código hash igual, então ele sabe que o objeto a ser inserido não está duplicado. Dessa forma, classes cujas instâncias são elementos de HashSet devem implementar o método `hashCode()`. Como consequência disso, a classe `Aluno`, no nosso exemplo, deve sobrescrever o método `hashCode()`.

Conforme o contrato geral de `hashCode()`, que consta na especificação da classe Object, se dois objetos são diferentes de acordo com `equals()` então não é obrigatório que seus códigos `hash` sejam diferentes.

Portanto, objetos que retornam o mesmo código hash não são necessariamente iguais. Assim, quando encontra no conjunto um objeto com o mesmo código hash do objeto a ser inserido, `HashSet` faz uma chamada ao método `equals()` para verificar se os dois objetos são iguais. Dessa forma, a classe `Aluno` deve sobrescrever o método `equals()` também. Veja a classe Aluno com esses métodos implementados na **Listagem 11**.

Criar código para `equals()` e `hashCode()` não é trivial, pois existem contratos definidos pela API de Java que devem ser rigorosamente seguidos. Por exemplo: se dois objetos são iguais, eles devem permanecer iguais durante toda a aplicação e devem resultar no mesmo `hashCode()`. Para facilitar essa tarefa, Eclipse e NetBeans têm opções para gerar esses métodos para as classes.

Listagem 11. Implementação de equals() e hashCode() na classe Aluno

```
1  public class Aluno implements Comparable<Aluno>{
2    private String nome;
```

```
6     Aluno(String nome, String curso, double nota) {
7       this.nome = nome;
8       this.curso = curso;
9       this.nota = nota;
10    }
11
12    public String toString() {
13      return this.nome;
14    }
15
16    public int compareTo(Aluno aluno) {
17      return this.nome.compareTo(aluno.getNome());
18    }
19
20    public boolean equals(Object o) {
21      Aluno a = (Aluno) o;
22      return this.nome.equals(a.getNome());
23    }
24
25    public int hashCode() {
26      return this.nome.hashCode();
27    }
28
29    // Métodos getters e setters
30    public String getNome() {
31      return this.nome;
32    }
33  }
```

A implementação de `hashCode()` e `equals()` foi simplificada devido a questões didáticas. Definimos que um aluno terá o código hash igual ao hash do seu nome – que é String. Sendo assim, precisamos apenas retornar o código hash do nome do aluno no método `hashCode()`. Ficou definido também que dois alunos são iguais quando têm nomes iguais, por isso no método `equals()` é retornada a comparação entre os nomes de dois alunos.

true ou false para indicar se o objeto foi inserido ou não no collection. Se for necessário, verifique o retorno do método para ter garantia da inclusão do objeto.

Se você programar pensando na interface e precisar de um conjunto (Set) classificado, use `TreeSet` em vez de `HashSet` sem necessidade de alterar o restante do código, pois tanto `TreeSet` como `HashSet` implementam exatamente os mesmos métodos de Set. No entanto, vale ressaltar que a classe dos elementos que são adicionados ao TreeSet deve implementar Comparable. Como Aluno já implementa esta interface não precisamos nos preocupar com isso.

> **Nota**: Todas as classes em Java são derivadas de `Object`, herdando assim métodos que, por padrão, devem ser sobrescritos, tais como: `clone()`, `equals()`, `hashCode()`, `toString()`, entre outros. Por padrão, o método `equals()` usa `==` para verificar se duas referências são iguais, enquanto `hashCode()` retorna um inteiro calculado a partir do endereço do objeto. Classes Java como `String` e `Date` já sobrescrevem tais métodos.

A interface Map

Vamos supor que agora queremos uma estrutura onde possamos recuperar os dados de um aluno passando apenas o seu nome como argumento de um método. Ou seja, informamos o nome do aluno e o objeto correspondente a esse nome é devolvido. Para isso vamos usar a interface Map, que não estende Collection. Isso causa uma mudança profunda na aplicação, visto que os métodos usados anteriormente não poderão ser usados. Map tem seus próprios métodos para inserir/buscar/remover elementos na estrutura.

Esta interface mapeia chaves para valores. Considerando a nova proposta do problema, a chave será o nome do aluno e o valor será o objeto aluno.

Para usar uma classe que implementa Map, quaisquer classes que forem utilizadas como chave devem sobrescrever os métodos `hashCode()` e `equals()`. Isso é necessário porque em um Map as chaves não podem ser duplicadas, apesar dos valores poderem ser. Para a implementação mostrada na **Listagem 12**, utilizamos um `TreeMap`, que garante que as chaves estarão em ordem ascendente.

Listagem 12. Implementação de estrutura usando Map

```
1  import java.util.*;
2
3  public class MapaAluno {
4
5    public static void main(String[] args) {
6      Map<String, Aluno> mapa = new TreeMap<String, Aluno>();
7
8      Aluno a = new Aluno("João da Silva", "Linux básico", 0);
9      Aluno b = new Aluno("Antonio Sousa", "OpenOffice", 0);
10     Aluno c = new Aluno("Lúcia Ferreira", "Internet", 0);
11     Aluno d = new Aluno("Benedito Silva", "OpenOffice", 0);
12     mapa.put("João da Silva", a);
13     mapa.put("Antonio Sousa", b);
14     mapa.put("Lúcia Ferreira", c);
15     mapa.put("Benedito Silva", d);
16     System.out.println(mapa);
17     System.out.println(mapa.get("Lúcia Ferreira"));
18
19     Collection<Aluno> alunos = mapa.values();
20     for (Aluno e : alunos) {
21       System.out.println(e);
22     }
23   }
24
25 }
```

Note que na declaração do collection informamos dois tipos: String e Aluno. O primeiro refere-se à chave e o segundo ao valor. O método para inserir na estrutura é `put()`, que recebe dois objetos (chave e valor). Para recuperar um objeto específico utilizamos o método `get()` passando a chave como parâmetro.

Como Map não estende Collection, não tem os métodos `iterator()` e `listIterator()`. Entretanto, existe o método `keySet()` que retorna um Set com as chaves do mapa, e o método `values()` que retorna um Collection com os valores associados às chaves. Assim, podemos percorrer o mapa partindo desses métodos e usando enhanced-for. A aplicação deste comando (for (Objeto obj: colecao) { ... }) para percorrer o mapa também é mostrada na **Listagem 12**.

Com tudo o que foi apresentado, podemos constatar que não existe a melhor implementação que resolve todos os problemas de estruturas de dados. Cada tipo de problema requer uma implementação diferente dependendo das características do mesmo. Escolher a implementação certa envolve saber o que sua interface oferece, quais as suas características e como ela será usada.

Na vídeo aula deste artigo apresentamos o Java Collections Framework, abordando as principais coleções e algumas boas práticas.

Conclusões

Java Collections Framework tem muito mais recursos do que aqueles que apresentamos neste artigo. É fundamental estudarmos a documentação da API para nos familiarizarmos com as opções que esta estrutura oferece. Falamos no texto que as interfaces não são thread-safe, no entanto a classe Collections possui um método synchronized para cada collection. Este método retorna objetos thread-safe, para o caso de você necessitar de acesso concorrente. O conhecimento dos contratos de `equals()` e `hashCode()` é muito importante para o

utilização adequada das interfaces aqui estudadas. A implementação errada desses métodos pode produzir resultados inesperados e errôneos.

Além das interfaces apresentadas, existem outras, tais como `NavigableSet`, `BlockingQueue`, `Deque`, `BlockingDeque`, `NavigableMap`, etc. É muito importante consultar sempre a documentação de Java SE para usar com eficiência a API.

Neste artigo, aprendemos que é fundamental conhecer a hierarquia das coleções de maneira a utilizar as interfaces para programar polimorficamente. Vimos também que, além da hierarquia, é fundamental conhecer os algoritmos – métodos – para manipular corretamente as estruturas.

Entender as características de cada interface e implementação fornece a base para a decisão de qual delas utilizar, visando solucionar, da melhor maneira possível, os problemas apresentados durante o desenvolvimento de aplicações.

Nota:

- Tutorial da Java Collections Framework no site da Oracle/Sun

Tecnologias:

Java UML

Perguntas Frequentes

Quem somos?

Por que a programação se tornou a profissão mais promissora da atualidade?

Como faço para começar a estudar?

Em quanto tempo de estudo vou me tornar um programador?

Sim, você pode se tornar um programador e não precisa ter diploma de curso superior!

Principais diferenciais da DevMedia

Qual o investimento financeiro que preciso fazer para me tornar um programador?

Como funciona a forma de pagamento da DevMedia?

Por **Carlos**

Em 2010

Comentários nesta publicação

Escrever um comentário sobre conteúdo

Rafael Silva

Nível 45

Tenho duas dúvidas com relação a ordenação:

1ª - quando eu devo usar Comparable e usar o Comparator. Não ficou claro para mim.

2ª - Eu tentei ordenar pela nota e pelo tipo primitivo double eu não consegui. Consegui somente com o Double que vem com compareTo. Como faço para ordenar com o tipo primitivo double?

há +1 ano

Ver comentários anteriores (7)

Marcio Souza
DevMedia

Por nada Rafael.

t+

há +1 ano

Luciano Oliveira
Nível 0

Muito melhor do que meus professores no meu tempo de faculdade.

Parabéns!

há +1 ano

Ricardo Crispim
Nível

Muito boa a didática usada neste artigo e apresentou conceitos essenciais. Parabéns pelo ótimo trabalho.

há +1 ano

Hilton Castro
Nível

gostei da comunicacao do artigo aida nao utilizei abplamenta mas parece simples

há +1 ano

Bruna Leal
Nível

Parabens pela edição.
O conteúdo é super facil de entender e não é cansativo.
Para quem é fã de Java é uma ótima escolha.

há +1 ano

Nossos casos de sucesso

Leonardo Carlos

Eu sabia pouquíssimas coisas de programação antes de começar a estudar com vocês, fui me especializando em várias áreas e ferramentas que tinham na plataforma, e com essa bagagem **consegui um estágio logo no início do meu primeiro período na faculdade.**

Lucas Rodrigues

Estudo aqui na Dev desde o meio do ano passado! Nesse período a Dev me ajudou a crescer muito aqui no trampo.
Fui o primeiro desenvolvedor contratado pela minha empresa. Hoje eu lidero um time de desenvolvimento!
Minha meta é continuar estudando e praticando para ser um Full-Stack Dev!

Heráclito Júnior

Economizei 3 meses para assinar a plataforma e sendo sincero valeu muito a pena, pois a **plataforma é bem intuitiva e muuuuito didática a metodologia**

Julio Cablen

Nossa! Plataforma maravilhosa. To amando o curso de desenvolvimento front-end, tinha coisas que eu ainda não tinha visto. **A didática é do jeito que qualquer pessoa consegue aprender.** Sério, to apaixonado, adorando demais.

Joelberth Sena

Adquiri o curso de vocês e logo percebi que são os melhores do Brasil. É um passo a passo incrível. **Só não aprende quem não quer. Foi o melhor investimento da minha vida!**

Felipe Nunes

Foi um dos melhores investimentos que já fiz na vida e tenho aprendido bastante com a plataforma. Vocês estão fazendo parte da minha jornada nesse mundo da programação, **irei assinar meu contrato como programador graças a plataforma.**

Wanderson Oliveira

Comprei a assinatura tem uma semana, aprendi mais do que 4 meses estudando outros cursos. Exercícios práticos que não tem como não aprender, estão de parabéns!

José Lucas

Obrigado DevMedia, nunca presenciei **uma plataforma de ensino tão presente na vida acadêmica de seus alunos**, parabéns!

Eduardo Dorneles

Aprendi React na plataforma da DevMedia há cerca de 1 ano e meio... **Hoje estou há 1 ano empregado** trabalhando 100% com React!

Adauto Junior

Já fiz alguns cursos na área e **nenhum é tão bom quanto o de vocês.** Estou aprendendo muito, muito obrigado por existirem. Estão de parabéns... Espero um dia conseguir um emprego na área.

Ver todos os casos de sucesso

Artigos:

Front-End Javascript Iniciantes Angular Dart Engenharia Mobile Node.js Python React Native Vue.js Android Banco de Dados Delphi Flutter Java Kotlin .Net PHP React Spring Gratuitos

DevCast:

HTML e CSS Javascript Angular Engenharia Mobile Node.js Python React Native Android Banco de Dados Delphi Flutter Java Automação .Net PHP React Spring Gratuitos Canal Mais

Guia:

Fundamentos .NET PHP Python Java Delphi HTML e CSS JavaScript Node React Native Flutter Banco de Dados Mobile Spring Arquitetura Automação Engenharia + Assuntos