Coleções e Serialização

Tiago T. Wirtti

27 de junho de 2014

Resumo

Neste texto entenderemos os conceitos fundamentais de coleções em Java e aplicaremos tais conceitos à solução de problemas que aparecem frequentemente em programação. Você aprenderá o que significa "serializar"um objeto e entenderá a ideia por trás do tipo genérico (conhecido como *generics*), aplicando tais recursos na solução de problemas práticos de programação.

1 Abstrações de coleções

As coleções podem ser descritas como grupamentos de objetos que seguem determinadas regras e possuem determinadas características. As principais abstrações para uma coleção são:

- Lista
- Conjunto
- Fila
- Pilha
- Mapa

Cada uma dessas abstrações possui características específicas, que descreveremos a seguir.

1.1 Lista

A lista é uma sequencia dinâmica de elementos indexados (Figura 1). Suas principais características são:

- Aceita repetição.
- Utiliza índices que nunca se repetem.
- A ordem de armazenamento é a ordem de inserção.
- É presentada pela interface *java.util.List*[1] em Java.

[1] Não confundir com a classe de interface gráfica `java.awt.List`

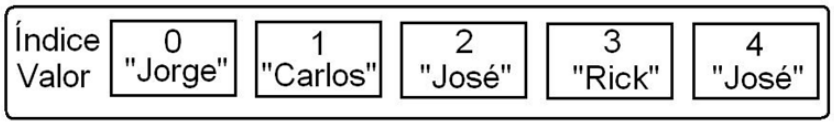

Figura 1: Esquema de uma lista.

1.2 Conjunto

O conjunto possui o mesmo conceito dos conjuntos numéricos em matemática. É um grupo de elementos sem repetição (Figura 2). Suas principais características são:

- Coleção de itens exclusivos, ou seja, sem repetição.
- Não há índices.
- A ordem de armazenamento não tem relação com a ordem de inserção.
- É presentada pela interface *java.util.Set* em Java.

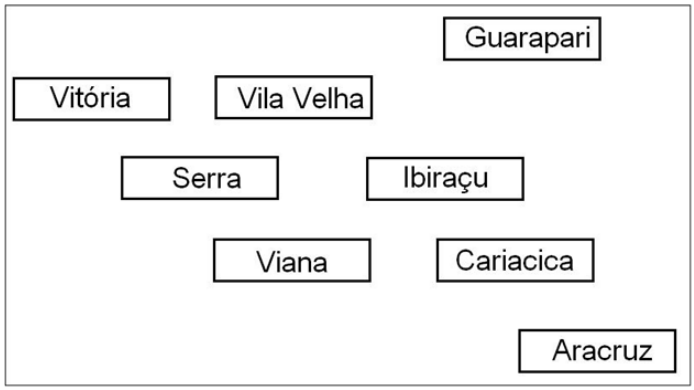

Figura 2: Esquema de um conjunto.

1.3 Fila

Estrutura que organiza os seus elementos em sequencia de forma que o primeiro elemento a entrar é o primeiro a sair (FIFO, *First Input, First Out*) (Figura 3). Suas principais características são:

- Coleção de itens que aceita repetição.
- Implementa FIFO.

- A ordem de armazenamento tem relação com a ordem de inserção.
- É presentada pela interface *java.util.Queue* em Java.

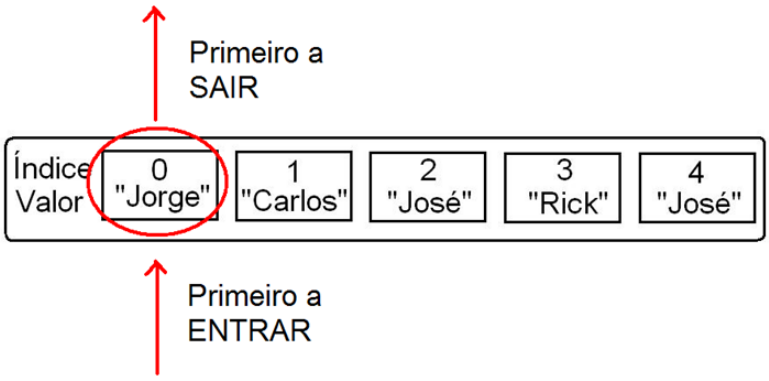

Figura 3: Esquema de uma fila.

1.4 Mapa

Estrutura que organiza seus elementos associando a cada um deles um valor, ou chave. Ou seja, cada entrada de um mapa (Figura 4) é uma associação entre um objeto chave a um objeto valor. Assim, o conjunto de associações entre objeto e valor é conhecido como mapa, ou tabela associativa. O mapa é equivalente ao conceito de dicionário, encontrado em muitas linguagens. Suas principais características são:

- Acesso através de uma chave.
- A chave é definida através de um código de *hashing*.
- A ordem de armazenamento não tem relação com a ordem de inserção.
- É presentada pela interface *java.util.Map* em Java.

2 Principais coleções em Java

Como observamos nas Figuras 5 e 6, a API Java implementa várias estruturas de dados prontas que colecionam objetos (coleções) para atender diversas aplicações. Essas interfaces, cujas abstrações foram apresentadas na seção anterior, se encontram, respectivamente, nos pacotes `java.util.Collection`[1] e `java.util.Map`[2].

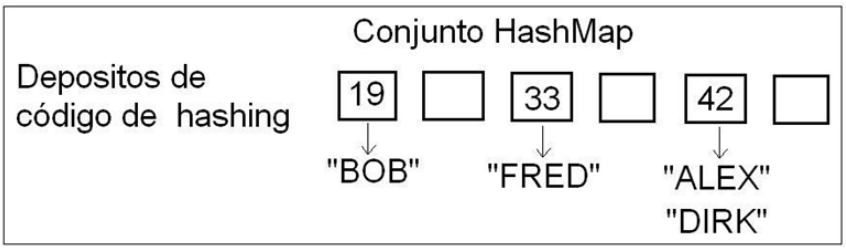

Figura 4: Representação conceitual de um mapa.

2.1 Interfaces de `java.util.Collection`

- `Collection`: Superinterface a partir da qual as interfaces Set, SortedSet, Queue e List são criadas.
- `List`: Coleção (lista) ordenada por índice e que pode conter elementos duplicados (mas não índices) [4].
- `Set`: Coleção (conjunto) sem duplicatas [5].
- `SortedSet`: Coleção (conjunto) ordenada sem duplicatas[6].
- `Queue`: Coleção que modela uma fila (FIFO) [7].

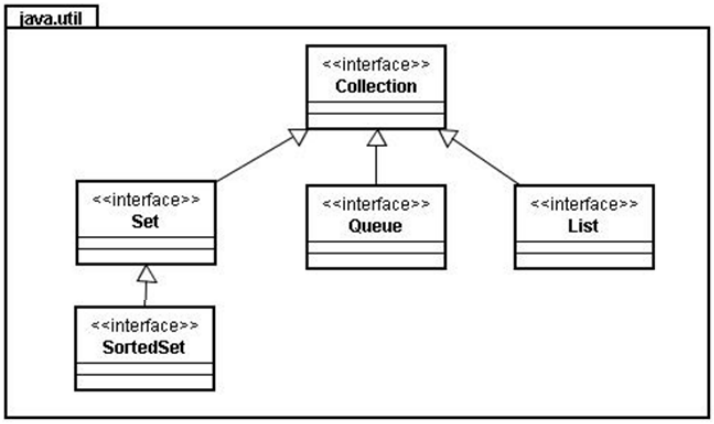

Figura 5: Hierarquia da interface `Collection`.

2.2 Interfaces de `java.util.Map`

- `Map`: Superinterface a partir da qual as interfaces `ConcurrentMap` e `SortedMap` são criadas. Associa chaves a valores e não pode conter chaves duplicadas.

- `SortedMap`: Mapa com chaves ordenadas em ordem natural [8].
- `ConcurrentMap`: Mapa que suporta acesso concorrente (várias *threads*) [9].
- Fornece a implementação de algumas classes: `Hashtable`, `HashMap` e `LinkedHashMap`.

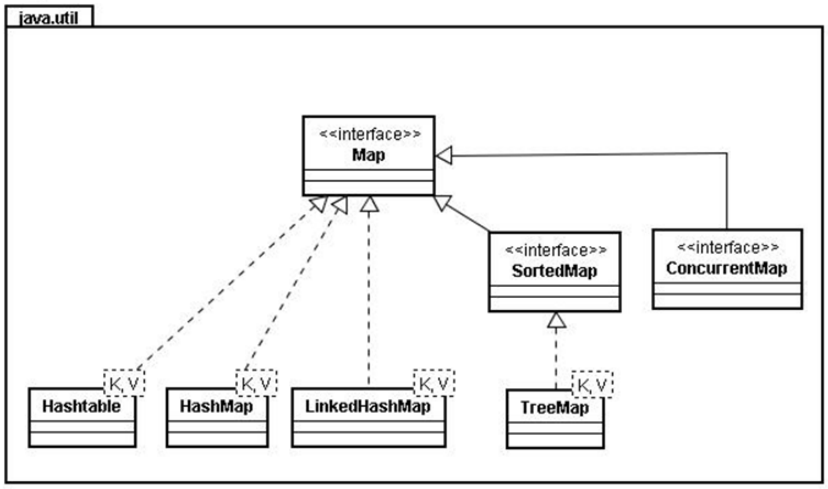

Figura 6: Hierarquia da interface `Map`.

Antes de prosseguir, é importante entender os conceitos de ordenação e classificação.

Coleção ordenada:

- É a ordem em que os elementos do conjunto serão mostrados quando percorridos (iterados, lidos).
- A ordem da iteração é conhecida (ordem de inserção, ordem de indexação...).

Coleção classificada:

- Significa que a ordem do conjunto está determinada por alguma(s) regra(s), conhecida(s) como ordem de classificação.
- Os elementos estão classificados por alguma ordem natural (alfabética, numérica, ou definida por um comparador[2]).

2.3 Interface `java.util.List`

As características da estrutura lista já foram apresentadas. A interface `List` é implementada pelas classes `LinkedList`, `ArrayList` e `Vector`, conforme mostrado na Figura 7. Toda classe da interface `List` pode ser percorrida através de

[2] O fornecimento de um comparador ocorre através da implementação da interface `Comparator` [3].

um iterador, implementado pela interface `java.util.Iterator`. As principais características do objeto de iteração são:

- Possibilita iteração sobre uma coleção sem utilizar índices.
- Possibilita a remoção de elementos durante a iteração.
- Os principais métodos são `hasNext()`, `next()` e `remove()`.
- O `Iterator` é obtido pelo método `interator()` da interface `Collection`.

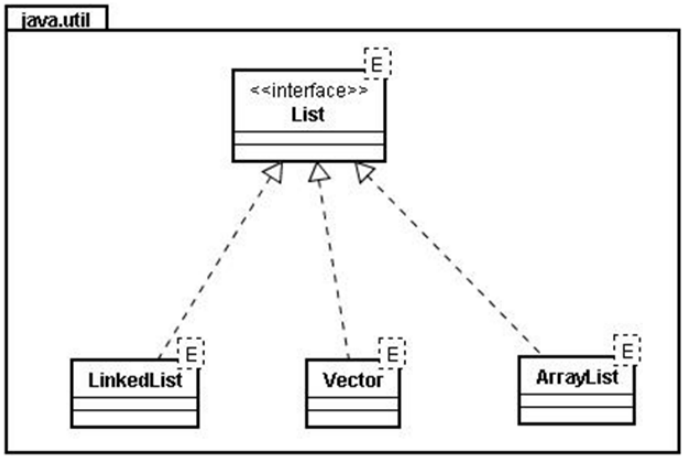

Figura 7: Hierarquia da interface `List`.

Vamos apresentar cada classe da interface `List`. Principais características:

- É um array que pode crescer dinâmicamente.
- É um conjunto ordenado por índices, mas não-classificado.
- Proporciona iteração e acesso aleatório com rapidez, mas com baixo desempenho para inserções e exclusões.

Agora vamos estudar um exemplo com `ArrayList` e `Iterator` (Figura 8). No marcador "1"da Figura 8, temos a declaração de duas listas dinâmicas, `ListaDeCores` e `ListaDeCoresARemover`, que são do tipo `ArrayList`, com referências do tipo `List`. A expressão `<String>` significa que as listas suportam o tipo `String` e suas subclasses[3]. O marcador "2"mostra a carga da lista a partir de um vetor usando um comando de repetição `for` sem contador explícito. O marcador "3"mostra o uso do iterador, que simplifica a manipulação da lista dinâmica, abstraindo para o programador o controle da iteração. Observe que o programador itera pela lista

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class ExemploArrayList {
   private static final String[] cores =
      { "VERMELHO", "VERDE", "AZUL", "PURPURA", "CIANO", "ROXO",
        "ROSA", "LARANJA" };
   private static final String[] removerCores = { "VERMELHO",
        "BRANCO", "AZUL", "ROXO" };

   public ExemploArrayList(){
1     List <String> listaDeCores = new ArrayList <String>();
      List <String> listaDeCoresARemover = new ArrayList <String>();

2     for ( String cor : cores )
         listaDeCores.add( cor );

      for ( String cor : removerCores )
         listaDeCoresARemover.add( cor );

      System.out.println( "ArrayList lista de cores: " );
      System.out.println(listaDeCores.toString());

      System.out.println( "\nArrayList lista de cores a REMOVER: " );
      System.out.println(listaDeCoresARemover.toString());
      removeCores( listaDeCores, listaDeCoresARemover );

      System.out.println( "\n\nArrayList depois da remoção: " );

      for ( String cor : listaDeCores )
         System.out.print( cor + "   " );
   }

   // remove de colecao1 as cores especificadas em colecao2
   private void removeCores( List <String> colecao1,
                             List <String> colecao2)
   {
3     Iterator <String> iterator = colecao1.iterator();
      while (iterator.hasNext()) {
          if (colecao2.contains( iterator.next() ))
              iterator.remove();// remove Color atual
      }
   }
   public static void main( String args[] )
   {
      new ExemploArrayList();
   }
}
```

Figura 8: Exemplo de utilização de `ArrayList`.

usando a função `hasNext()`, para saber se há um próximo elemento, `next()`,

para recuperar este elemento e `remove()` para excluir o elemento da lista.

Outra estrutura importante é a `LinkedList`, que é também uma lista dinâmica (com as mesmas características do `ArrayList`), mas implementada como uma lista duplamente encadeada. A implementação com lista duplamente encadeada (que para o programador Java é transparente), torna o desempenho da lista melhor quando há a necessidade de muita inserção e exclusão (em elementos intermediários da lista). Entretanto, vale ressaltar que se a sua aplicação precisa de uma lista em que a maioria das operações é de adição e remoção (sobre o final da lista), então a `LinkedList` terá desempenho inferior à `ArrayList`.

A Figura 9 mostra a implementação de um método (`removeItens(...)`) que recebe uma referência do tipo `List` e um intervalo de valores, devolvendo a lista sem os elementos do intervalo especificado. Observe que os elementos são selecionados pelo método `subList()` e removidos pelo método `clear()`.

```
private void removeItens(List <String> lista, int inicio, int fim ) {
      lista.subList( inicio, fim ).clear();
}
```

Figura 9: Exemplo de exclusão em `LinkedList`.

Outra classe (pouco conhecida e utilizada) da interface `List` é `Vector`. A principal diferença de `Vector` para `ArrayList` é que `Vector` suporta acesso concorrente, e `ArrayList` não. As principais características da classe `Vector` são:

- Versão equivalente e mais antiga de `ArrayList`, mantida por questões de compatibilidade.
- É uma versão sincronizada de `ArrayList`.
- Deve ser utilizada apenas quando houver a necessidade de programação concorrente (uso de *threads*).
- Por ter suporte nativo a programação concorrente, ela é mais lenta que `ArrayList`.

2.4 Interface `java.util.Set`

Como já mencionado neste texto, a interface `Set` implementa o conceito de conjunto, no qual não á repetição de elemento e a ordem de inserção não necessariamente coincide com a ordem de armazenamento. A interface `Set` é implementada pelas classes `HashSet`, `LinkedHashSet` e `TreeSet`, conforme mostrado na Figura 10. Assim como a as classes da interface `List`, as classes de `Set` podem ser percorridas através de um iterador, implementado pela interface `java.util.Iterator`.

A classe `HashSet` apresenta as seguintes características:

[3] O tipo `String` é definido como `final`, por questões de segurança, para evitar que ele possa ser estendido.

- Conjunto não-ordenado e não-classificado.
- Armazena seus elementos em uma tabela de *hash*.
- Indicado quando for necessário um conjunto sem duplicadas e sem ordem na iteração.

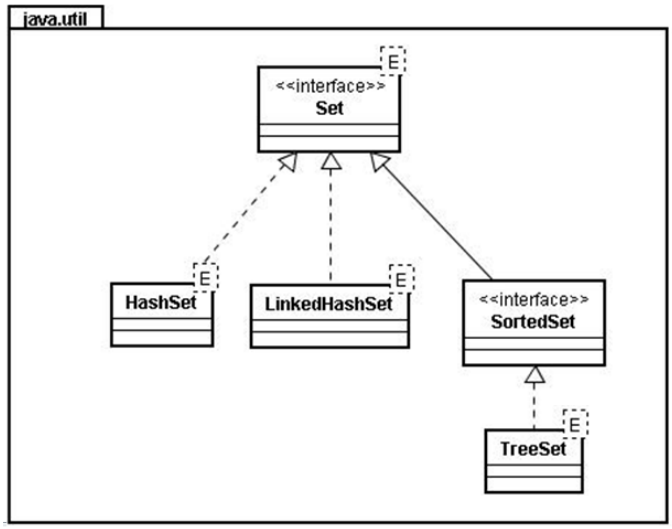

Figura 10: Interface `Set` e suas subclasses.

A seguir (Figura 11) mostramos um exemplo de conversão de um objeto, que pode ser de qualquer subtipo de `Collection`, sendo convertido a um conjunto sem duplicatas e não ordenado. O programador não precisa se preocupar em eliminar duplicatas, pois isso acontece automaticamente na criação do objeto `HashSet`!

```
private void imprimeNaoDuplicados(Collection <String> colecao) {

    Set <String> conjunto = new HashSet <String>(colecao);

    System.out.println( "\nExibe elementos não duplicados:" );
    System.out.println(conjunto);
}
```

Figura 11: Exemplo de criação de um objeto `HashSet`.

A classe `LinkedHashSet`, que também implementa `Set`, possui as seguintes características:

- Conjunto ordenado (por ordem de inserção ou de último acesso) e não-classificado.

- É uma versão ordenada do `HashSet`.
- Indicado quando for necessário um conjunto sem duplicadas em que haja necessidade de uma ordem na iteração

A classe `TreeSet` fornece uma implementação de `Set` com as seguintes características:

- Conjunto ordenado e classificado.
- É implementado internamente com uma árvore *red-black* (ordenada e balanceada).
- Indicado quando for necessário um conjunto sem duplicadas em que haja necessidade de classificação dos elementos.

A Figura 12 mostra um exemplo de criação e utilização de um objeto `TreeSet`. O marcador "1"mostra a criação de um objeto `TreeSet` a partir de um vetor `cores`. Repare que `cores` é convertido para uma lista (`List`, de tamanho fixo [10]) antes de ser passado ao construtor `TreeSet(...)`! Isso é possível porque `Arrays` é uma classe utilitária que fornece vários métodos estáticos para operações com vetores [10]. O marcador "2"mostra o método `headSet(...)`, que obtém os elementos que são estritamente[4] menores que `"orange"`. O marcador "3"mostra o método `tailSet(...)`, que obtém os elementos que são estritamente maiores que `"orange"`. O marcador "4"mostra como obter, respectivamente, o primeiro e o último elementos do `TreeSet`.

2.5 Classes da interface `java.util.Map`

A interface `Map`, mencionada na seção 2.2, fornece algumas classes muito interessantes para trabalhar com mapas. A classe `Hashtable` é a versão mais antiga de `HashMap`. Ambas possuem basicamente as mesmas funcionalidades. As características principais de `HashMap` são:

- Tem ótimo desempenho, pois pode ser percorrido sem ordem de iteração
- Fornece um mapa não-ordenado e não-classificado.
- Permite chaves com diversos valores nulos.

A seguir (Figura 13) mostra um programa separa as palavras de um texto, as conta e armazena em uma estrutura `HashMap`. O marcador "1"mostra a criação de um objeto mapa, informando a dupla `<chave, valor>` de tipos genéricos. O marcador "2"mostra a recuperação de um valor pela sua chave (`mapa.get(word)`) e a atualização do valor associado à chave (`mapa.put(word, count+1)`). O marcador "3"destaca a extração das chaves do mapa `mapa.keySet()`. Posteriormente, as chaves são ordenadas e o seu conteúdo é lido (marcador "4") e impresso.

A seguir apresentamos as características da classe `HashMap`:

[4] Há uma versão com dois parâmetros (`headSet(E toElement, boolean inclusive)`) que recebe o booleano `"inclusive"`, que, quando `true`, inclui `toElement` no resultado retornado.

```
import java.util.Arrays;
import java.util.SortedSet;
import java.util.TreeSet;

public class ExemploTreeSet
{
   private static final String cores[] = { "yellow", "green",
       "red", "green", "red", "white", "orange", "red", "green" };

   public ExemploTreeSet() {
1     SortedSet <String> c =
                 new TreeSet <String>(Arrays.asList(cores));

      System.out.println( "Conjunto (Set) ordenado: " );
      imprimeConjunto( c );

      // obtém o "headSet" (o que vem antes de) "orange"
2     System.out.print( "\nheadSet (\"orange\"):  " );
      imprimeConjunto(c.headSet("orange"));

      // obtém o "tailSet" (o que vem depois de) "orange"
3     System.out.print( "tailSet (\"orange\"):  " );
      imprimeConjunto(c.tailSet("orange"));

      // obtém primeiro e último elementos
4     System.out.println( "primeiro: "+ c.first());
      System.out.println( "último : "+ c.last());
   }

   private void imprimeConjunto( SortedSet <String> conjunto ){
      for (String s : conjunto)
         System.out.print ( s + "  " );
   }

   public static void main(String args[]){
      new ExemploTreeSet();
   }
}
```

Figura 12: Utilizando a classe `TreeSet`.

- Fornece um mapa não-ordenado e não-classificado.
- É uma versão mais antiga de `HashMap` e permite sincronização de seus métodos.
- Possui desempenho inferior a `HashMap`.
- Não permite que qualquer chave contenha um ou mais valores nulos.

A classe `LinkedHashMap` se caracteriza por;

- Ser ordenada por ordem de inserção ou por ordem de acesso e não-classificado

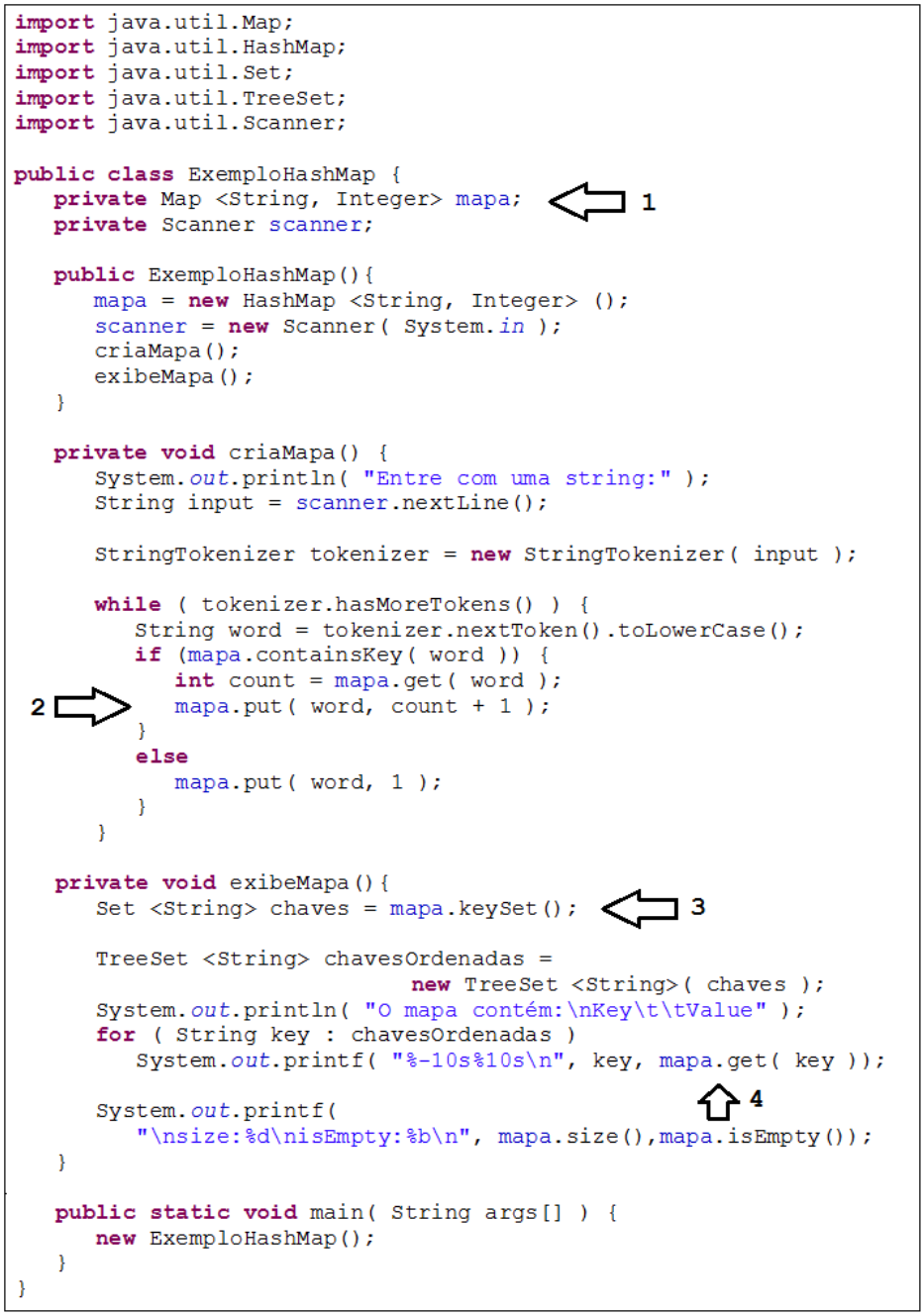

Figura 13: Utilizando a classe `HashMap`.

- Para inserção e exclusão é mais lenta que HashMap, mas é mais eficiente em iterações do que HashMap.

A classe TreeMap tem as seguintes características:

- É ordenada e classificada
- Permite que o programador utilize critérios arbitrários de comparação através da interface `Comparator`.

3 Classes utilitária `Collections`

Além das estruturas de dados mostradas até agora, a API Java oferece um conjunto de algoritmos através da classe `Collections`. Esses algoritmos são fornecidos ao programador através de métodos estáticos. Entre estes, destacamos:

- `sort`: Classifica elementos de uma `List`.
- `binarySearch`: Localiza um elemento em uma `List`.
- `reverse`: Inverte os elementos de uma `List`.
- `shuffle`: Ordena aleatoriamente os elementos de uma `List`.
- `fill`: Sobrescreve todos os elementos de uma `List` para um valor especificado.
- `copy`: Copia referências de uma `List` em outra.
- `min`: Retorna o menor elemento em uma `Collection`.
- `max`:Retorna o maior elemento em uma `Collection`.
- `addAll`: Acrescenta todos os elementos de um array a uma coleção.
- `frequency`: Calcula quantos elementos na coleção são iguais ao elemento especificado.
- `disjoint`: Determina se duas coleções são disjuntas.

Vale ressaltar que:

- A classe `Collections` fornece algoritmos implementados para ótimo desempenho.
- Estes algoritmos são fornecedos através de métodos estáticos listados anteriormente.
- A classe `Collections` é herança direta de `Object`.
- A inteface `Collection`, que define estruturas de dados, é herança direta de `Iterable` e não deve ser confundida com a classe `Collections`.

O exemplo da Figura 14, a seguir, ilustra a utilização de alguns métodos de `Collections`.

```java
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class ExemploOrdenacao {
   private static final String cores[] =
   { "black", "yellow", "green", "blue", "violet", "silver" };

   public void imprimeElementos(){
      List <String> lista = Arrays.asList( cores );
      System.out.println( "Não ordenada:\n" + lista + "\n");

      // ordena lista
      Collections.sort( lista );
      System.out.println( "Ordenada:\n" + lista+ "\n");

      // "embaralha" lista
      Collections.shuffle(lista);
      System.out.println( "Misturada:\n" + lista+ "\n");

      // ordena lista em ordem decrescente
      Collections.sort( lista, Collections.reverseOrder() );
      System.out.println( "Ordenada decrescente:\n" + lista+ "\n");
   }

   public static void main( String args[] ){
      ExemploOrdenacao sort1 = new ExemploOrdenacao();
      sort1.imprimeElementos();
   }
}
```

Figura 14: Utilizando a classe utilitária `Collections`.

4 Comparando objetos

Para ordenar objetos de uma determinada classe em uma coleção é preciso que se tenha um critério de comparação bem definido. A LP Java nos fornece uma interface que dá suporte à comparação de objetos. Trata-se da interface `Comparator`.

Entre as caraterísticas da interface `Comparator` destacam-se:

- Fornece um meio arbitrário de impor ordem a uma coleção através do método `compareTo(T o1, T o2)`.
- A implementação do método `compareTo()` precisa ser consistente com `equals()`, ou seja:

$$objA.equals(objB)) \Leftrightarrow compareTo(objA, objB) == 0 \qquad (1)$$

A seguir vemos um exemplo de criação de uma classe de comparação `TimeComparator` (Figura 15) para uma classe `Time` (omitida neste texto) e que armazena o horário em horas, minutos e segundos. A classe que testa o critério de comparação, chamada `TesteTime` é mostrada na Figura 16.

```java
import java.util.Comparator;

public class TimeComparator implements Comparator <Time> {
   public int compare(Time time1, Time time2) {
      int hComp = time1.getHour() - time2.getHour();

      if (hComp!= 0 ) return hComp;

      int mComp = time1.getMinute() - time2.getMinute();

      if (mComp!= 0 ) return mComp;

      return time1.getSecond() - time2.getSecond();
   }
}
```

Figura 15: Implementação de um comparador para a classe `Time`.

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class TesteTime {
   public void imprimeElementos() {
      List <Time> lista = new ArrayList <Time>();

      lista.add( new Time(  6, 24, 34 ) );
      lista.add( new Time( 18, 14, 58 ) );
      lista.add( new Time(  6, 05, 34 ) );
      lista.add( new Time( 12, 14, 58 ) );
      lista.add( new Time(  6, 24, 22 ) );

      System.out.println( "Elementos desordenados:\n"+ lista );

      Collections.sort( lista, new TimeComparator() );

      System.out.println( "Elementos ordenados:\n" + lista );
   }

   public static void main( String args[] ) {
      TesteTime compTime = new TesteTime();
      compTime.imprimeElementos();
   }
}
```

Figura 16: Testando o critério de comparação em `TimeComparator`.

5 Serialização

Serialização é o processo de salvar um objeto em um meio de armazenamento (como um arquivo de computador ou um buffer de memória) ou transmiti-lo por

uma conexão de rede, seja em forma binária ou em formato de texto como o XML. Esta série de bytes pode ser usada para recriar um objeto com o mesmo estado interno que o original [11].

Para serializar um objeto em Java é necessário que ele implemente a interface `java.io.Serializable` [12].

Mostraremos a seguir (Figura 17) um exemplo de implementação de serialização em arquivo de uma lista de qualquer objeto (serializável). A classe `SuporteArquivo` possui dois métodos: `gravar(...)` e `abrir(...)`. No método gravar o objeto é serializado na marcação "1". No método `abrir`, o objeto é serializado na marcação "2".

```
public class SuporteArquivo {

	public static <T> void gravar(File file, List <T> lista){
		ObjectOutputStream arq;
		try {
			arq = new ObjectOutputStream(new FileOutputStream(file));
	1 ⇨		arq.writeObject(lista); // grava um objeto
			arq.flush();
			arq.close();
			System.out.println("Gravou: "+file.getAbsolutePath());
		} catch (FileNotFoundException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
	public static <T> List<T> abrir(File file){
		List <T> lista = new ArrayList<T>();
		try {
			ObjectInputStream arq =
				new ObjectInputStream(new FileInputStream(file));
	2 ⇨		lista = (List<T>)arq.readObject();
			arq.close();
		} catch (IOException e) {
			e.printStackTrace();
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		}
		System.out.println("Leu arquivo "+file.getAbsolutePath());
		return lista;
	}
}
```

Figura 17: Serializando uma lista de <T>.

Para testar a serialização, o programa da Figura 18 pode ser utilizado.

```
public class TesteSerializa {
	public static void main(String[] args) {
		List <String> lista = new ArrayList<String>();

		lista.add("teste-01");
		lista.add("teste-02");
		lista.add("teste-03");
		lista.add("teste-04");
		lista.add("teste-05");
		File f = new File("teste_ser.txt");

		SuporteArquivo.gravar(f, lista);

		List <String> outra = new ArrayList<String>();
		System.out.println(outra);
		outra = SuporteArquivo.abrir(f);
		System.out.println(outra);
	}
}
```

Figura 18: Testando a serialização.

Referências

[1] Interface Collection<E>. Java™ Platform Standard Ed. 7. <http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html>. Acessado em: 23 junho 2014.

[2] Interface Map<E>. Java™ Platform Standard Ed. 7. <http://docs.oracle.com/javase/7/docs/api/java/util/Map.html>. Acessado em: 23 junho 2014.

[3] Interface Comparator<E>. Java™ Platform Standard Ed. 7. <http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html>. Acessado em: 23 junho 2014.

[4] Interface List<E>. Java™ Platform Standard Ed. 7. <http://docs.oracle.com/javase/7/docs/api/java/util/List.html>. Acessado em: 23 junho 2014.

[5] Interface Set<E>. Java™ Platform Standard Ed. 7. <http://docs.oracle.com/javase/7/docs/api/java/util/Set.html>. Acessado em: 23 junho 2014.

[6] Interface SortedSet<E>. Java™ Platform Standard Ed. 7. <http://docs.oracle.com/javase/7/docs/api/java/util/SortedSet.html>. Acessado em: 23 junho 2014.

[7] Interface Queue<E>. Java™ Platform Standard Ed. 7. <http://docs.oracle.com/javase/7/docs/api/java/util/Queue.html>. Acessado em: 23 junho 2014.

[8] Interface SortedMap<E>. Java™ Platform Standard Ed. 7. <http://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html>. Acessado em: 23 junho 2014.

[9] Interface ConcurrentMap<E>. Java™ Platform Standard Ed. 7. <http://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentMap.html>. Acessado em: 23 junho 2014.

[10] Interface Arrays>. Java™ Platform Standard Ed. 7. <http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html>. Acessado em: 26 junho 2014.

[11] Serialização>. Wikipedia. <http://pt.wikipedia.org/wiki/Serializa%C3%A7%C3%A3o>. Acessado em: 26 junho 2014.

[12] Serializable>. Java™ Platform Standard Ed. 7. <http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html>. Acessado em: 26 junho 2014.